ESCUELA POLITÉCNICA DEL EJÉRCITO

DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN ELECTRÓNICA Y TELECOMUNICACIONES

PROYECTO DE GRADO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERÍA

DESARROLLO DE UNA GUÍA DE PRÁCTICAS DE LABORATORIO PARA LA MATERIA DE PROCESAMIENTO DIGITAL DE SEÑALES UTILIZANDO LA TARJETA "ADZS-TS201S" DE ANALOG DEVICES

DANILO MARCELO CARVAJAL RAMÍREZ

PAÚL FERNANDO PÁRAMO VACA

SANGOLQUÍ – ECUADOR

2009

CERTIFICACIÓN

Certificamos que el presente Proyecto de Grado fue realizado por los señores Danilo Marcelo Carvajal Ramírez y Paúl Fernando Páramo Vaca bajo nuestra dirección

_____

Ing. Derlin Morocho
DIRECTOR

_____

Ing. Rubén León
CODIRECTOR

RESUMEN

En el presente proyecto se realizó el estudio de la tarjeta ADZS-TS201S de la familia de procesadores TigerSHARC tanto en hardware como en software incluida su programación mediante la utilización de unidades de cálculo, registros y demás herramientas.

Se verifica el funcionamiento de la tarjeta mediante la manipulación de la misma a través de aplicaciones como: manipulación de leds, operaciones lógicas, operaciones aritméticas, accesos hacia y desde memoria y utilización de archivos externos.

Las aplicaciones fueron implementadas en el software de desarrollo VisualDSP++ 5.0, herramienta cuya instalación y aprendizaje es parte del estudio. Además se realiza rutinas para la grabación de voz y comunicación entre tarjetas, desarrolladas en lenguaje de código C, C++ y ensamblador.

Se elaboran guías prácticas de laboratorio para sistemas discretos (eco, reverberación), Transformada Discreta de Fourier y filtros. Los filtros desarrollados son de tipo: FIR, IIR y Filtros adaptativos. Los filtros deben ser diseñados a una frecuencia de muestro de 48 KHz y en cada práctica se especifica el procedimiento que se debe seguir para su implementación y verificación.

DEDICATORIA

A mis padres que con gran esmero me han sabido guiar con principios de honestidad, constancia y respeto, y por haberme dado todo su apoyo para poder brindarme una educación superior. A la memoria de mi hermano el cuál esta presente siempre en mi corazón. Y en especial está dedicada para mi hermano Kevin que con cariño y amor a su familia demuestra la calidad de persona en la que se está convirtiendo mientras sigue en su etapa de crecimiento.

Danilo Marcelo Carvajal Ramírez

A mis padres quienes me apoyaron y me dieron las fuerzas necesarias para poder levantarme en todo momento ante cualquier tropiezo, a mi hermano que aunque está lejos lo llevo siempre en mi corazón y a mi novia, mis primos, tíos, abuelos y leales amigos porque son mi razón de vivir.

Paúl Fernando Páramo Vaca

AGRADECIMIENTO

A mi familia por todo el apoyo y aliento, a mis amigos con los cuales compartimos muchos momentos buenos durante mi paso por la universidad. A mi Director y Codirector por compartir sus conocimientos, por haberme dado su respaldo y su guía para la realización del proyecto.

Danilo Marcelo Carvajal Ramírez

A mi madre que con su infinito amor y trabajo hizo posible que mi camino haya sido siempre el correcto, a mi padre porque sin su apoyo incondicional nada de esto hubiese sido posible, a mi hermano por ser mi compañero en los momentos difíciles y a mi director y codirector del proyecto de grado por su guía y aliento permanente.

Paúl Fernando Páramo Vaca

PRÓLOGO

Los circuitos analógicos presentan el inconveniente de causar distorsión a las señales de entrada que se requiere sean procesadas; esto debido a las tolerancias propias de los componentes utilizados. Por su parte, el procesamiento digital de señales ofrece la ventaja de que esta distorsión sea disminuida de sobremanera al no existir componentes analógicos que causen distorsión por sus tolerancias de fabricación. Otras de las ventajas que presenta un procesador digital de señales es que, debido a la tecnología de estado sólido, permite un consumo de energía pequeño, un procesamiento de las señales de entrada en tiempo real, un control sobre el comportamiento del hardware con tan solo modificar el software (versatilidad), un rango de frecuencias de operación del procesador que abarca con facilidad el de las comunicaciones de voz (lo cual lo hace apto para estas aplicaciones) y un robusto código de instrucciones que satisfacen cualquier aplicación de procesamiento digital de señales.

Dadas las ventajas y facilidades que un DSP ofrece al diseñador de software, sobre los circuitos analógicos de bajo desempeño, se hace necesario el aprendizaje del funcionamiento de una tarjeta DSP como la ADZS-TS201S, familia TigerSHARC, no sólo para la solución de problemas reales en el ámbito industrial y de las investigaciones modernas sino además para aportar en los estudiantes conocimientos de Procesamiento Digital de Señales enfocados al desarrollo de aplicaciones prácticas.

Explicado lo anterior, se plantea realizar el estudio de la tarjeta ADZS-TS201S, del funcionamiento y del lenguaje de programación de su procesador ADSP-TS201, que es uno de los más recientes miembros de la familia de procesadores TigerSHARC. Además se pretende encaminar este estudio para el desarrollo de numerosas aplicaciones de procesamiento de señales.

Finalmente, a través de éstas múltiples aplicaciones que presenta el procesador ADSP-TS201, se elaborará una guía de prácticas de laboratorio para la materia de Procesamiento Digital de Señales con lo cual se busca contribuir con el aprendizaje de los estudiantes del Departamento de Ingeniería Eléctrica y Electrónica permitiéndoles contar con herramientas prácticas necesarias que puedan facilitar la comprensión de la parte teórica y reforzar su conocimiento en lo que respecta a dicha materia.

ÍNDICE

CAPITULO III

CAPITULO IV

INDICE DE TABLAS

INDICE DE FIGURAS

GLOSARIO

ASIC Circuito Integrado para Aplicaciones Específicas, o ASIC por sus siglas en inglés, es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general.

Buffer Memoria de almacenamiento temporal de información. Se utiliza para mejorar el rendimiento o también para compensar la diferencia de tiempos y velocidades que manejan los distintos dispositivos.

Butterfly Cada una de las estructuras elementales que se dan en la implementación del algoritmo FFT.

CPLD Dispositivo Logico Programable Complejo. Permite implementar sistemas más eficaces, ya que utilizan menor espacio, mejoran la fiabilidad del diseño, y reducen costos.

Debug Herramienta que proporciona un conjunto de métodos y propiedades que ayudan a depurar el código

Emulador *Software* que permite ejecutar programas en una plataforma diferente de aquella para la cual fueron escritos originalmente. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar de forma precisa el dispositivo que se está emulando.

Handshake Especifica el protocolo de control utilizado para establecer una comunicación con un puerto serial.

Host	Ordenador que funciona como el punto de inicio y final de las transferencias de datos.
IC	Circuito integrado.
IRQ	Cada dispositivo que desea comunicarse con el procesador por interrupciones debe tener asignada una línea única capaz de avisar a éste de que le requiere para una operación. Esta línea es la llamada IRQ ("Interrupt ReQuest", petición de interrupción).
ISR	Rutina de Servicio de Interrupción
Latch	Circuito electrónico usado para almacenar información en sistemas lógicos asíncronos. Puede almacenar un bit de información.
Marshalled messages	Para poder enviar mensajes por la red, el servidor debe serializar (marshal) los mensajes. Al serializar (marshalling) se transforma la representación de memoria de un objeto a un formato de datos adecuado para almacenamiento o transmisión.
Memory pools	Son bloques de memoria de tamaño fijo que permiten ubicaciones de memoria dinámica comparables a nuevos operadores de C++ permitiendo que su rendimiento se adapte a sistemas de tiempo real.
Pipeline	Conjunto de elementos procesadores de datos conectados en serie, en donde la salida de un elemento es la entrada del siguiente. Los elementos del pipeline son generalmente ejecutados en paralelo, en esos casos, debe haber un almacenamiento tipo buffer insertado entre elementos.

Reberveración	Fenómeno derivado de la reflexión del sonido consistente en una ligera prolongación del sonido una vez que se ha extinguido el original, debido a las ondas reflejadas.
SOC	System-on-a-chip o System on Chip. Se refiere a integrar todos los componentes de una computadora o sistema electrónico en un circuito integrado simple. Puede contener funciones digitales, análogas y de radio frecuencia. Una típica aplicación se encuentra en los sistemas embebidos.
SPORT	Puerto serial
Thread	Característica que permite a una aplicación realizar varias tareas concurrentemente. Los distintos *threads* comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.
Twiddle	Factor usado para el cálculo de la Transformada Rápida de Fourier (FFT).
Xilinx	Empresa líder en el mercado de investigación y desarrollo de dispositivos lógicos programables digitales (PLD).

INTRODUCCION

El propósito del presente proyecto de grado es el de realizar el estudio de la tarjeta "ADZS-TS201s" de Analog Devices, familia TigerSHARC, desarrollar sus aplicaciones y elaborar una guía de prácticas de laboratorio para la materia de procesamiento digital de señales.

A continuación se resume el contenido por capítulos del presente proyecto de grado.

CAPITULO 1: Introducción al Procesador ADSP-TS201.

CAPITULO 2: Estudio y análisis del procesador ADSP-TS201.

En el presente capítulo se realiza un estudio en hardware y software de la tarjeta ADSP-TS201. El estudio en hardware abarca los principales elementos de la arquitectura de la tarjeta como son los bloques de cálculo, memoria, registros, etc. El estudio en software hace referencia a las características de la sintaxis para utilización de registros, bloques de cálculo y tipo de datos (punto fijo y punto flotante).

CAPITULO 3: Características de la tarjeta ADSP-TS201 EZ-KIT LITE.

Se detalla las características de la arquitectura del sistema de la tarjeta EZ-KIT Lite, se describe el tipo de interfaces (audio, expansión) que contiene la tarjeta. Se presenta la configuración de los dip switches para señales de audio, habilitación de interrupciones, ancho del puerto de enlace, requerimientos de interrupción y banderas.

Contiene la presentación grafica de la ubicación de leds, pulsadores, fuente de alimentación, jacks de audio, conectores R45 para comunicación con su respectiva descripción. Además se describe el procedimiento para la instalación de los drivers de la tarjeta y el proceso de registro y validación para la instalación del software de desarrollo VisualDSP++ 5.0.

CAPITULO 4: Descripción y manejo del Software de Desarrollo "VisualDSP++ 5.0".

En este capítulo se hace una introducción al manejo del software de desarrollo VisualDSP++ 5.0, se describe el entorno y las herramientas con las que cuenta el programa, elementos de depuración, tipos de lenguaje de programación que soporta VisualDSP++ 5.0. Descripción de pasos para la creación de proyectos, manipulación de proyectos (edición, archivos de origen), construcción y corrida de los programas.

CAPITULO 5: Programación del Procesador Digital de Señales ADSP-TS201 y desarrollo de aplicaciones.

Se realiza la descripción de forma general de las instrucciones de lenguaje ensamblador de cada uno de los bloques de cálculo, definición de registros y bits. Además, una revisión del set de instrucciones que consta de una amplia variedad de instrucciones que han sido optimizadas para aplicaciones de DSP. Utilización de archivos externos y manipulación de accesos desde y hacia memoria para posteriores aplicaciones desarrolladas en el proyecto.

CAPITULO 6: Manipulación y realización de rutinas de aplicación con la tarjeta ADZS-TS201S.

En este capitulo se realiza la creación de rutinas para la comprobación del funcionamiento y desempeño de la tarjeta ADZS-TS201S. Las rutinas implementadas son grabación de voz y comunicación punto a punto entre tarjetas. En cada uno de las

rutinas se realiza la explicación de la programación como por ejemplo activación del puerto de audio, configuración de interrupciones externas, manejo del puerto de enlace, activación de los pulsadores y leds de la tarjeta.

CAPITULO 7: Desarrollo de prácticas de Procesamiento Digital de Señales ADSP-TS201.

Se realiza la implementación de algoritmos para sistemas discretos como eco y reverberación, además de algoritmos de filtros digitales que van hacer utilizados para el tratamiento de señales. Los tipos de filtros digitales implementados son FIR, IIR y filtros adaptativos. En base a estas implementaciones, se elaboran cinco guías de prácticas de laboratorio. Los algoritmos implementados para cada una de las prácticas se encuentran desarrollados con su debida explicación para que se puedan seguir

CAPITULO 8: Conclusiones y Recomendaciones.

Se detalla las conclusiones y recomendaciones del estudio de la tarjeta ADZS-TS201S y de las prácticas realizadas.

CAPITULO I

INTRODUCCIÓN AL PROCESADOR TIGERSHARC ADSP-TS201S

1.1 INTRODUCCIÓN AL PROCESADOR TIGERSHARC ADSP-TS201S

El procesador TigerSHARC ADSP-TS201 representa un hito en la integración de tecnología DRAM embebida y en el diseño de una arquitectura que alcance un balance en todos los frentes: velocidad, costo, potencia y ancho de banda.

El procesador TigerSHARC ADSP-TS201S es un procesador superescalar estático de súper alto rendimiento optimizado para tareas de procesamiento de señales largas e infraestructura de comunicaciones. El DSP combina un gran ancho de memoria con dos bloques de cálculo, que soportan procesamiento de punto flotante (IEEE 32-bit y precisión extendida 40-bit) y punto fijo (8-, 16-, 32-, and 64-bit), para fijar un nuevo estándar de rendimiento para procesadores digitales de señales. La arquitectura superescalar estática TigerSHARC permite al DSP ejecutar hasta cuatro instrucciones por cada ciclo, realizando 24 operaciones de punto fijo (16 bits) o 6 operaciones de punto flotante.

Cuatro buses de datos internos independientes de 128 bits de ancho, cada uno conectando a los seis bancos de memoria de 4M bits, habilitan datos de palabra cuádruple y accesos de I/O y proveen 33,6 Gbytes por segundo de ancho de banda de memoria interna. Operando a 600 MHz, el núcleo del procesador ADSP-TS201S tiene

un tiempo de ciclo de instrucción de 1.67 ns. Usando su característica de instrucción-simple, datos-múltiples (SIMD), el procesador ADSP-TS201S puede realizar 4.8 billones, 40 bits MACS o 1.2 billones, 80 bits MACS por segundo.

El núcleo de este Procesador TigerSHARC es único entre los procesadores de señales de alto rendimiento debido a que soporta directamente ambos tipos de datos fijos y flotantes. El núcleo además soporta un único conjunto de instrucciones orientadas a aplicaciones comunicaciones, permitiendo implementaciones de software de funciones que estaban previamente solo disponibles a través de ASICs. La familia de procesadores TigerSHARC presenta dos tipos de interfaces que directamente soportan sistemas de multiprocesamiento escalable sin la necesidad de lógica externa costosa. El bus común permite hasta ocho procesadores TigerSHARC, un host y memoria externa para compartir un bus y un mapa de memoria global, permitiendo un modelo de programación de multiprocesamiento muy simple. Los puertos de enlace son interfaces con un gran ancho de banda que permite comunicaciones punto a punto entre TigerSHARCs u otros dispositivos. Por lo tanto, el procesador ADSP-TS201S es compatible en código con los otros procesadores TigerSHARC.

Las innovaciones claves del procesador TigerSHARC ADSP-TS201 se basan en su especial estructura de memoria y balanceada arquitectura, a través de las cuales el ADSP-TS201 entrega más altos niveles de rendimiento que sus antecesores operando en hasta el doble de la frecuencia.

Entre sus innovaciones claves se encuentra:

- Provee operaciones DSP súper escalar estáticas de alto rendimiento, optimizado para infraestructura de telecomunicaciones y otras infraestructuras que demanden aplicaciones DSP de multiprocesamiento.

- Opera excepcionalmente bien sobre algoritmos DSP y puntos de referencia de I/O.

- Soporta bajas transferencias DMA entre memoria interna, memoria externa, periféricos del mapa de memoria, puertos de enlace, procesadores host y con otros DSPs (multiprocesamiento).

- Facilita la programación DSP a través de un conjunto de instrucciones extremadamente flexibles y una arquitectura DSP con un lenguaje en un alto nivel amigable.

Es decir, ningún otro procesador provee tal riqueza en sus características que directamente soporte aplicaciones de tan alto rendimiento o que requieran múltiples procesadores, como el ADSP-TS201s de la Familia TigerSHARC.

CAPITULO II

ESTUDIO Y ANÁLISIS DEL PROCESADOR ADSP-TS201

Este capítulo provee una descripción general del núcleo del procesador y su arquitectura periférica.

2.1 HARDWARE ADSP-TS201 TIGERSHARC

La referencia de Hardware del procesador ADSP-TS201 TigerSHARC provee de la información requerida para diseñar y configurar los sistemas del procesador TigerSHARC.

2.1.1 Descripción general de características

El procesador ADSP-TS201 es un procesador TigerSHARC de alto rendimiento de 128 bits que establece un nuevo estándar de rendimiento para procesadores digitales de señales, combinando unidades de cálculo múltiples para punto-flotante y punto–fijo y procesando también palabras de gran tamaño.

Este procesador mantiene una filosofía de diseño de computación escalable "system on chip", incluyendo 24 Mbits de memoria interna, divididos en seis bloques de memoria de 4K, periféricos de entrada y salida integrados, una interfaz del procesador host, controladores DMA, puertos de enlace LVDS y conectividad de bus compartida para multiprocesamientos independientes.

2.1.2 Arquitectura del ADSP-TS201

La arquitectura del procesador ADSP-TS201 consiste de dos divisiones, el núcleo del procesador (donde se ejecutan las instrucciones) y los periféricos de I/O (donde los datos son guardados y el off-chip I/O es procesado).

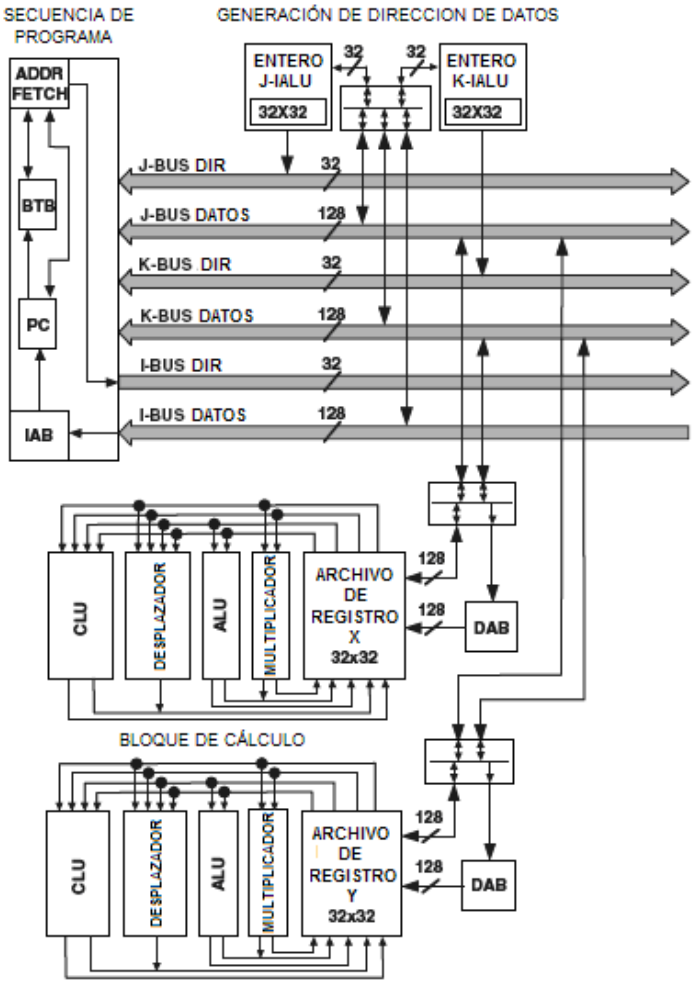

Figura. 2.1. Diagrama del núcleo del procesador TigerSHARC ADSP-TS201

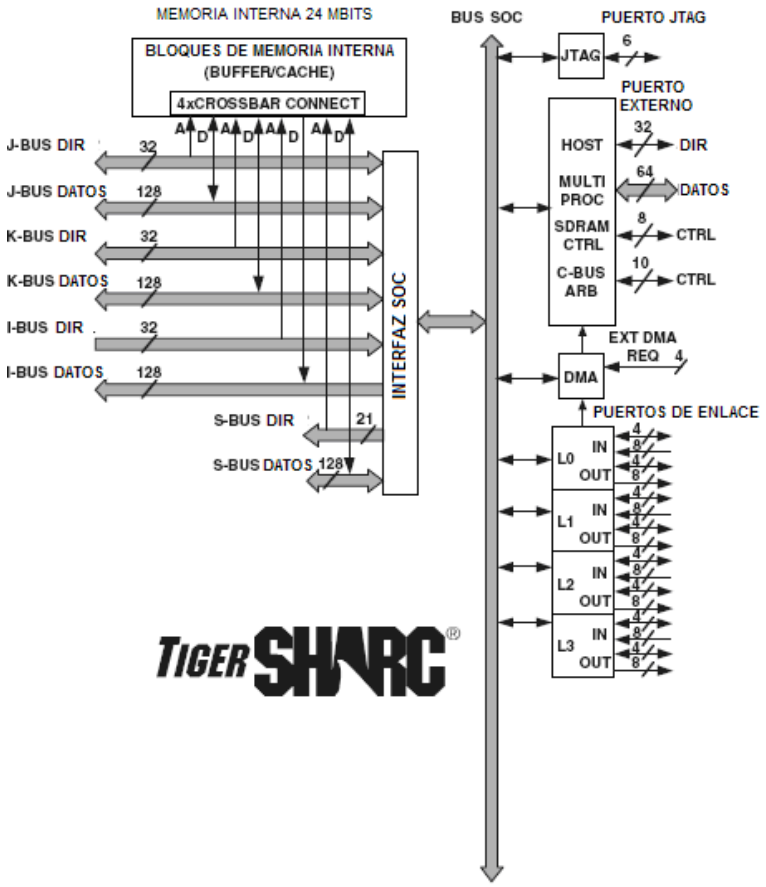

Figura. 2. 2. Diagrama Periférico del procesador TigerSHARC ADSP-TS201

Como se muestra en la Figura 2.1 y 2.2, el procesador tiene las siguientes características en su arquitectura.

- Dos bloques de cálculo: X e Y. Cada una consiste de un multiplicador, ALU, desplazador y un archivo de registro de 32 palabras.
- Dos ALU´s enteras (IALU´s): J y K. Cada una contiene un IALU de 32 bits y un archivo de registro de 32 palabras.
- Secuenciador de programa. Controla el flujo del programa y contiene un buffer de alineamiento de instrucción (IAB).
- Tres buses de 128 bits proveyendo una conectividad con un gran ancho de banda entre la memoria interna y el resto del núcleo del procesador.
- Un bus de 128 bits que provee una conectividad con un gran ancho de banda entre la memoria interna y los periféricos externos de I/O.

- Interfaz de puerto externo.
- Características de depuración.
- Puerto de acceso de pruebas JTAG.

2.1.3 Bloques de cálculo

El núcleo del procesador TigerSHARC contiene dos unidades de cálculo llamadas bloques de cálculo. Cada bloque de cálculo contiene un archivo de registros y cuatro unidades de cálculo independientes (ALU, CLU, multiplicador y desplazador). Para abarcar una gran variedad de necesidades de procesamiento las unidades de computación procesan datos en formatos punto fijo y punto flotante.

- **Unidad Aritmética Lógica (ALU)**

La ALU realiza operaciones aritméticas sobre datos de punto-fijo y punto flotante y operaciones lógicas sobre datos de punto-fijo. Además, ejecuta operaciones de conversión de datos tales como expansión y compresión sobre datos en formatos punto-fijo.

- **Acumulador de Multiplicación (Multiplicador)**

El multiplicador realiza multiplicaciones con punto-fijo o punto-flotante y operaciones de multiplicación/acumulación solamente con punto-fijo. Esta unidad además realiza todas las operaciones de multiplicación y acumulación complejas sobre datos de punto-fijo. Adicionalmente el multiplicador ejecuta operaciones de compresión de datos sobre resultados acumulados cuando se mueve datos al archivo de registro en formatos punto-fijo.

- **Desplazador**

El desplazador realiza desplazamientos aritméticos y lógicos, manipulación de bits, inserción de campos y extracción de campos. El desplazador trabaja sobre operandos de punto-fijo de 64 bits, uno o dos de 32 bits, dos o cuatro de 16 bits y cuatro u ocho de 8 bits.

- **Unidad Aritmética Lógica Entera (IALU)**

Las IALU's pueden ejecutar operaciones ALU estándar sobre archivos de registro IALU. Las IALU´s además ejecutan operaciones de carga, almacenamiento y transferencia de registros, proveyendo direcciones de memoria cuando los datos son transferidos entre memoria y registros. El procesador tiene dos IALU´s (la J-IALU y la K-IALU) que habilitan direcciones simultáneas para dos transacciones de hasta 128 bits en paralelo. Las IALU´s permiten ejecutar operaciones de cálculo con máxima eficiencia debido a que las unidades de cálculo pueden ser dedicadas exclusivamente para procesar datos.

2.1.4 Memoria

El procesador TigerSHARC ADSP-TS201 posee seis bloques de memoria interna. Cada bloque de memoria consiste de 4 Mbits de espacio de memoria, y está configurado como palabras de 128 K, cada una con 32 bits de ancho. Además, existen separados cuatro buses de datos internos de 128 bits; los cuales, pueden acceder a cualquiera de los bloques de memoria. Los bloques de memoria pueden guardar instrucciones y datos indistintamente, con un acceso por ciclo por bloque de memoria.

El bus de direcciones de 32 bits del procesador TigerSHARC provee un espacio de dirección de 4 gigapalabras. Este espacio de dirección es común para un grupo de procesadores TigerSHARC que comparten el mismo bus.

Para mayor información, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_*Reference contenido en el CD de EZ-KITLite.

2.1.5 Registros

- **Espacio de Registro Universal (Ureg)**

En el mapa de memoria del procesador, los registros universales están agrupados de acuerdo a su relación con la arquitectura del procesador.

El término registro universal indica importantes características acerca de un registro. Primero, el registro está en el mapa de memoria, indicando que el registro puede ser accedido por otros procesadores en un sistema multiprocesador a través de la dirección del registro. Segundo, los contenidos del registro pueden ser cargados desde memoria, grabados en la memoria o transferidos hacia o desde otros registros *Ureg* en el procesador. Tercero, el registro puede ser accedido como un registro simple doble cuádruple.

Los accesos para cargar, almacenar o transferir registros *Ureg* dentro de un procesador usan los nombres del registro y no su dirección del mapa de memoria.

El espacio de registro está compuesto de 64 grupos de registros de hasta 32 registros en cada grupo. Debido a que los grupos de registro tienen diferentes restricciones de acceso, es siempre importante conocer el grupo al cual pertenece el registro. Estos registros no deben ser usados para aplicaciones porque podrían causar un comportamiento inesperado del procesador TigerSHARC.

Los grupos de registros son:

Tabla. 2. 1. Grupos de registros

Registros	*Grupos*
Archivo de registro de bloque de cálculo.	0x00-0x09
IALU	0x0C-0x0F
Características de depuración	0x0A, 0x1B
Secuenciador de programa	0x1A
Controlador de memoria	0x1E-0x1F
DMA	0x20-0x23
Puerto externo de control/estado	0x24
Puerto de enlace	0x25-0x27
Controlador de interrupción	0x38, 0x39, 0x3A
JTAG	0x3D
AutoDMA	0x3F
Reservados	*

- **Grupos de registros del bloque de cálculo**

Cada archivo de registro del bloque de cálculo es un registro de 32 palabras. Existen dos archivos de registros, uno en cada bloque de cálculo (X e Y). El archivo de registro está duplicado en varios grupos, algunos de los cuales son aplicables únicamente para instrucciones de transferencia de datos. Algunos grupos apuntan al bloque de cálculo X y algunos al bloque Y; aquellos que apuntan a ambos bloques de cálculo en paralelo; de hecho, apuntan al mismo registro. Los registros X e Y son registros Universales *Ureg* del mapa de memoria.

- **Registros de estado del bloque de cálculo (XSTAT/YSTAT)**

Los registros XSTAT y YSTAT son registros de 32 bits que muestran el estado del bloque de cálculo a través de las banderas de estado del bloque de cálculo. Cada bandera se actualiza cuando una instrucción perteneciente a su unidad de cálculo es completada.

Para mayor información acerca de estos registros, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_Reference.*

- **Grupo de Registros IALU**

Cada IALU tiene dos grupos *Ureg*. El primer grupo es un archivo de registro de propósito general el cual incluye registros de 32 palabras normales cada uno. Estos registros son:

- Registros J0 – J31
- Registros K0 – K31

El segundo grupo es el registro de buffer circular el cual es usado para especificar los parámetros para almacenamiento circular *(circular buffering).* Los registros de buffer circular contienen los siguientes registros:

- 3 – 0 Registro de base de buffer circular JB3-0 y KB3-0
- 7 – 4 Registros de longitud de buffer circular JL3-0 y KL3-0
- 31 – 8 Reservados

- **Registro de control de banderas (FLAGREG)**

El registros FLAGREG es un registro de 32 bits que controla la dirección del pin de las banderas (entrada o salida) y proporciona controles a los valores de salida (1 o 0) para pines de bandera de salida. El registro FLAGREG aparece en la Figura 2.3.

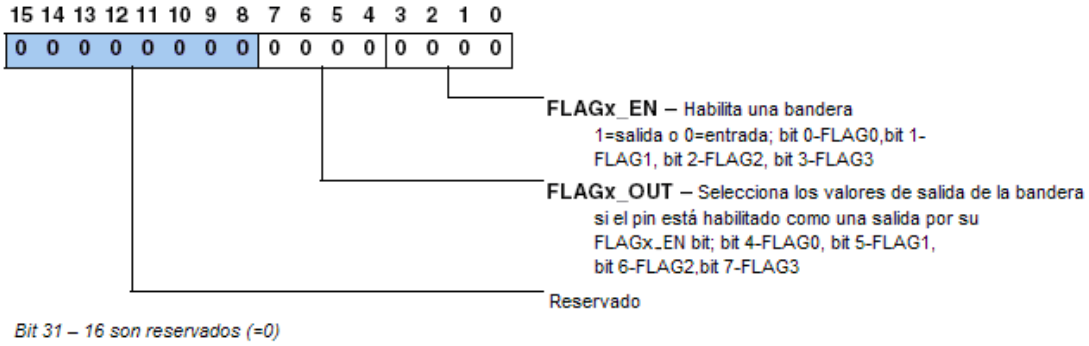

Figura. 2. 3. Descripción de bits (inferiores) del registro FLAGREG

- **Registro de Control del Secuenciador (SQCTL)**

En el registro SQCTL (Figura 2.4) usuario puede habilitar o deshabilitar la interrupción global a través del pin GIE.

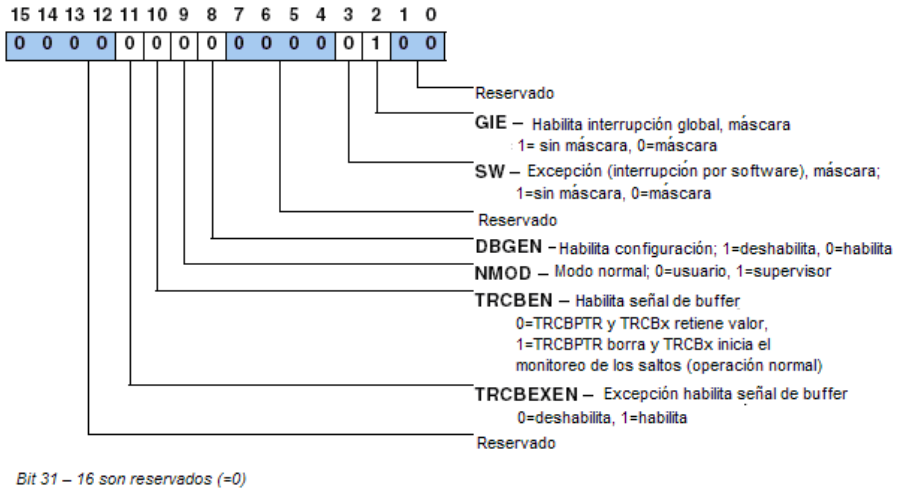

Figura. 2. 4. Descripción de bits (inferiores) del registro SQCTL

- **Grupos de Registros de Interrupciones**

El registro INTCTL es un registro de 32 bits que controla la sensibilidad de los pines de interrupción (nivel o borde de sensibilidad) y proporciona control de inicio/parada para los temporizadores. Ver la Figura 2.5.

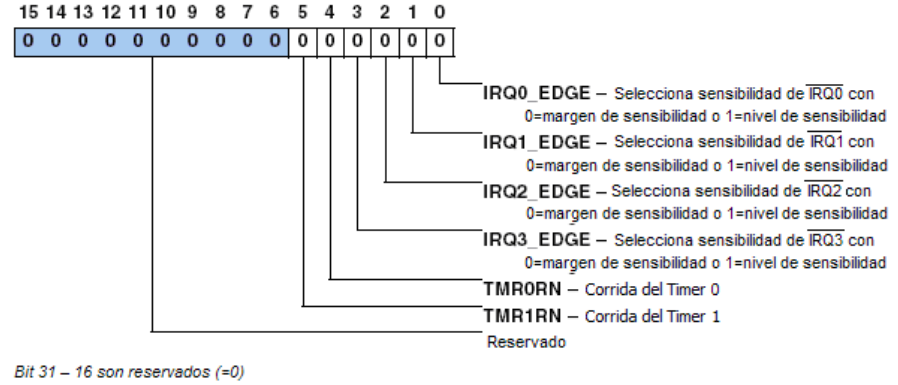

Figura. 2. 5. Descripción de bits (inferiores) del registro INTCTL

El registro PMASK está compuesto de dos registros: PMASKH y PMASKL. El registro PMASKH es usado para habilitar las interrupciones de los DMA's (DMA 2-10) y el registro PMAKSL habilita las interrupciones para DMA0, DMA1 y para los puertos externos.

Para mayor información acerca del control de interrupciones, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_Reference.*

- **Grupo de registros del puerto de enlace**

Los registros LRCTLx configuran las características de recepción para los puertos de enlace.

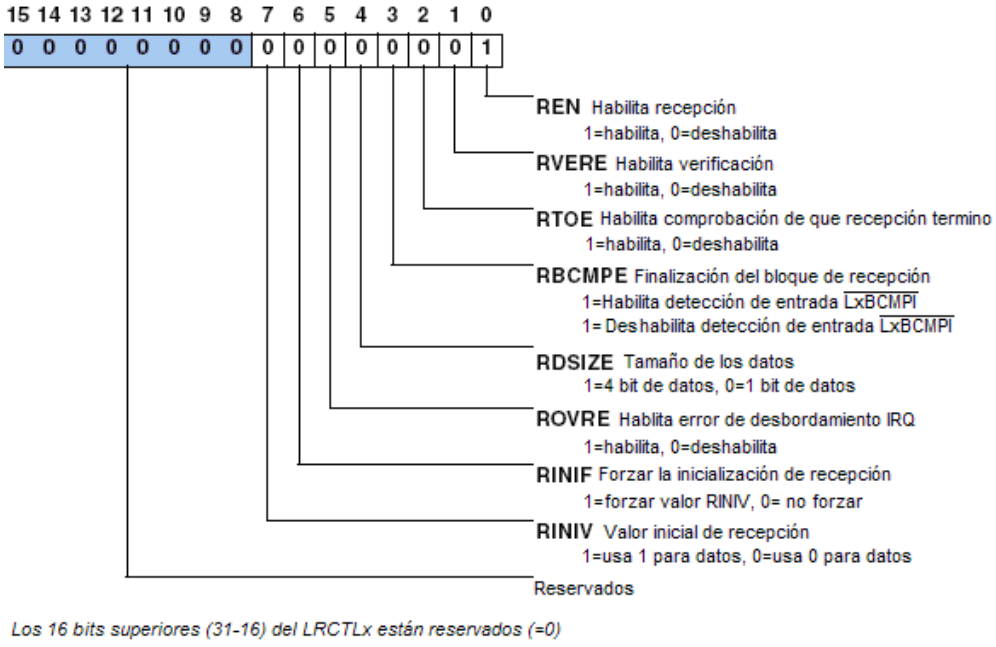

Figura. 2. 6. Descripción de bits (inferiores) del registro LRCTLx

- **Registros de control de transmisión de enlace (LTCTLx)**

Los registros LTCTLx configuran las características de transmisión para los puertos de enlace.

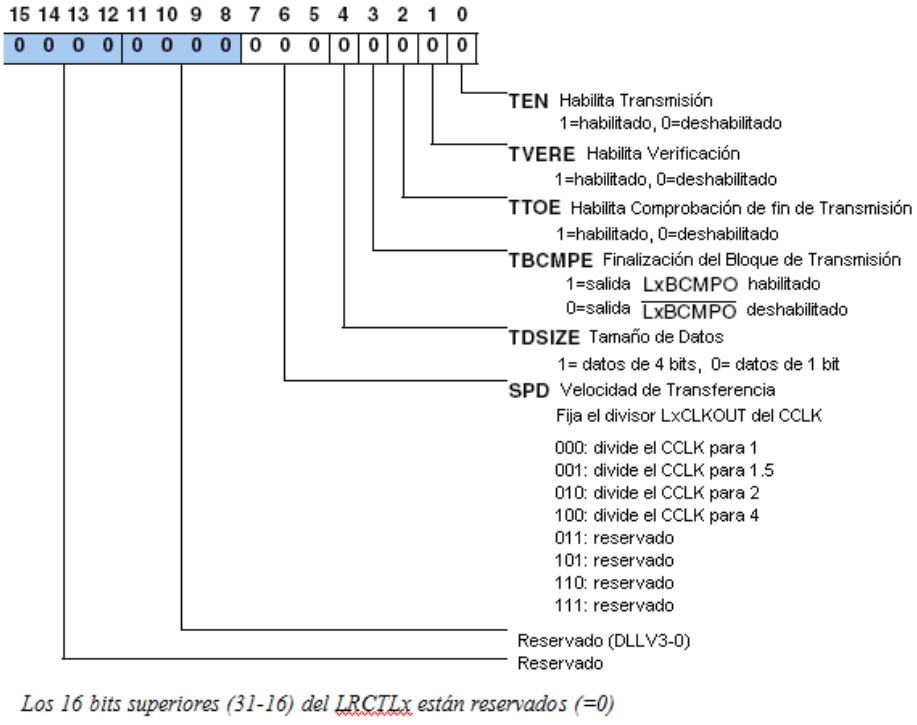

Figura. 2. 7. Descripción de bits (inferiores) del registro LTCTLx

- **Registros de estado de recepción de enlace (LRSTATx)**

Los registros LRSTATx indican el estado de recepción para los puertos de enlace.

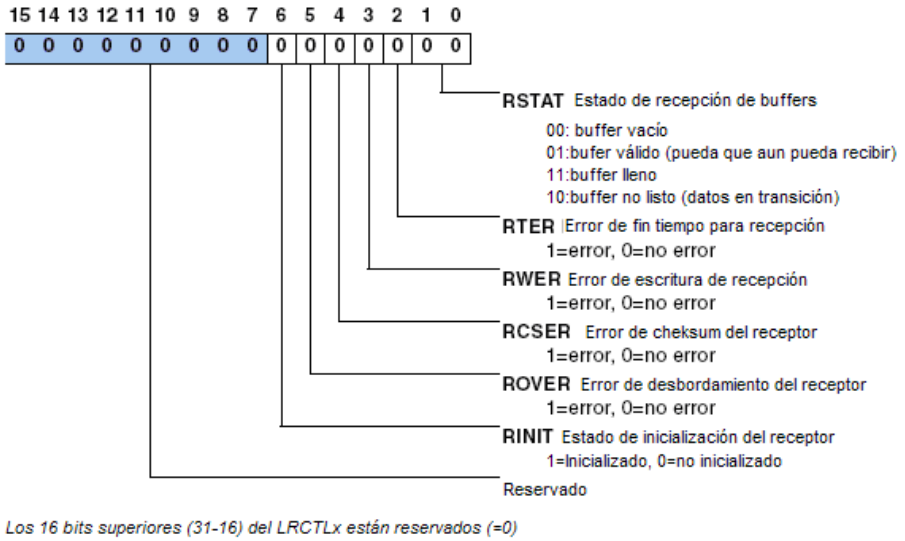

Figura. 2. 8. Descripción de bits (inferiores) del registro LRSTATx

- **Registros de estado de transmisión de enlace (LTSTATx)**

Los registros LTSTATx indican el estado de transmisión para los puertos de enlace.

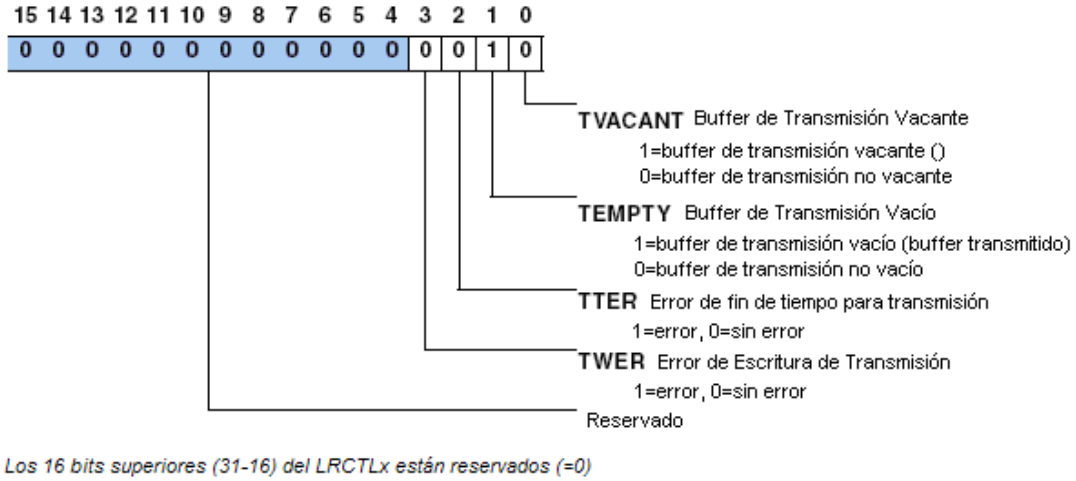

Figura. 2. 9. Descripción de bits (inferiores) del registro LTSTATx

- **Registro de configuración de sistema (SYSCON)**

El registro SYSCON define la configuración del bus de control. La descripción de los bits para este registro se muestra en las Figura 2.10 y Figura 2.11.

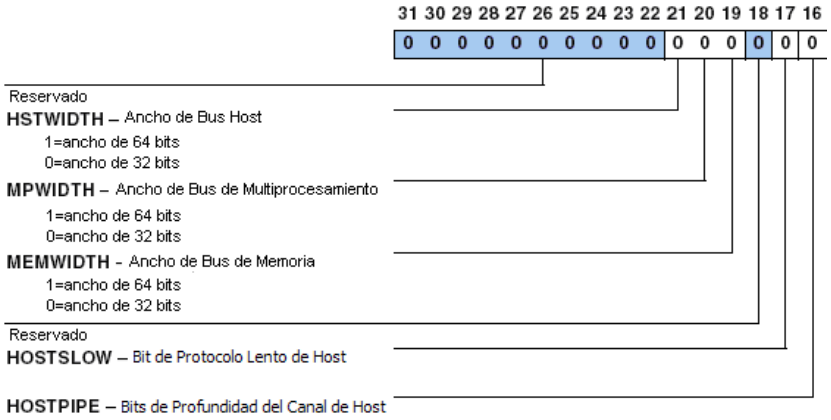

Figura. 2. 10. Descripción de los bits (superiores) del registro SYSCON

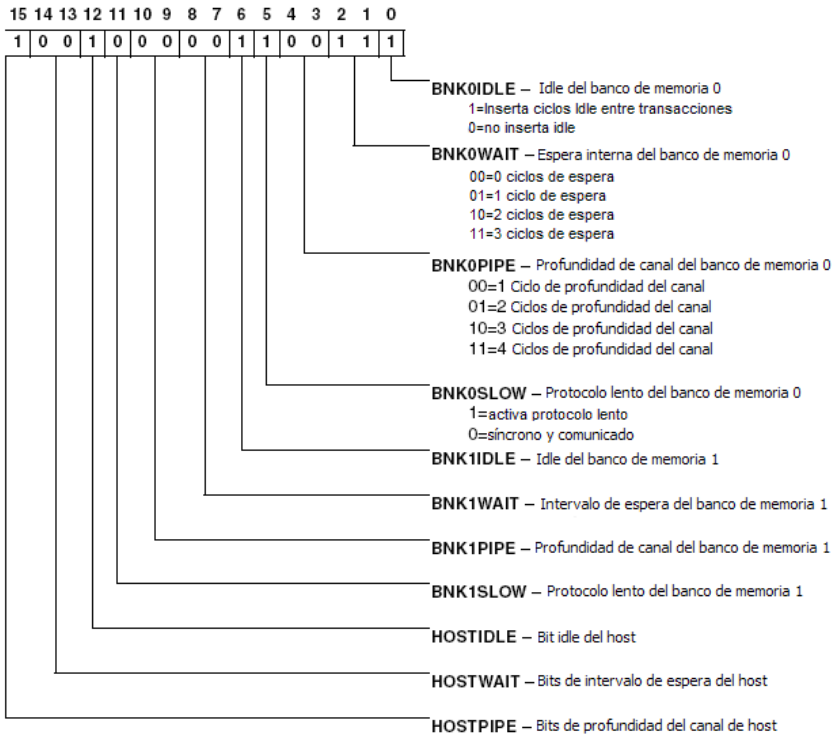

Figura. 2. 11. Descripción de los bits (inferiores) del registro SYSCON

- **Registro de configuración SDRAM (SDRCON)**

El registro SDRCON define la configuración de la SDRAM. La descripción de bits para este registro se muestra en la Figura 2.12.

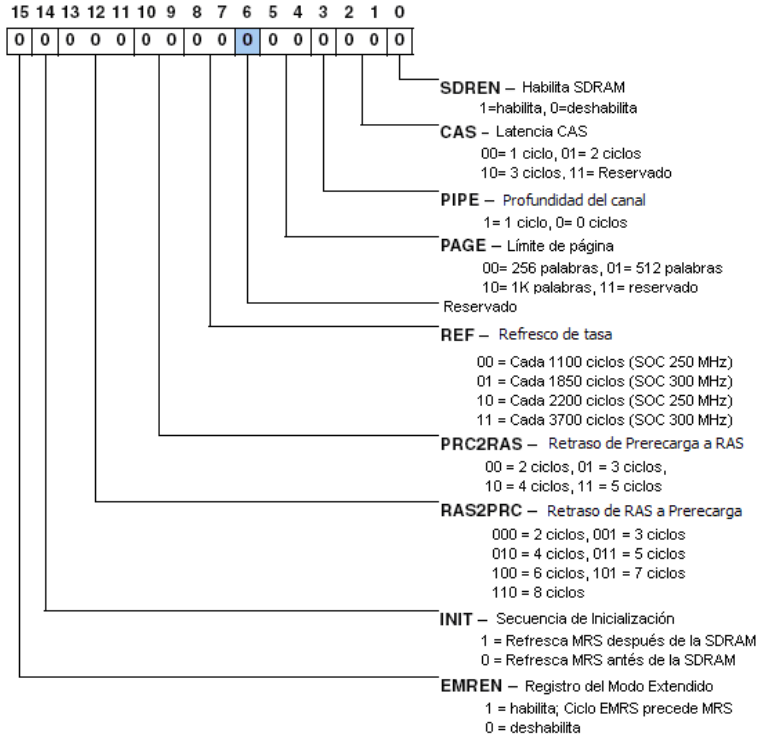

Figura. 2. 12. Descripción de los bits (inferiores) del registro SDRCON

2.1.6 Interfaz SOC (System on Chip)

La interfaz system on-chip (SOC) conecta los buses del núcleo del procesador (J, K, I y S) al bus SOC de los periféricos de I/O. El bus SOC tiene un ancho de 128 bits, y su reloj (SOCCLK) opera a la mitad de la tasa del reloj del núcleo del procesador (CCLK). Los periféricos que pueden ser maestros o esclavos en el bus SOC son:

- Interfaz SOC
- Puerto externo
- Unidad DMA

Los periféricos que pueden ser solo esclavos en el bus SOC son:

- Puertos de enlace
- Controlador de interrupciones
- Temporizadores
- Banderas
- Puerto JTAG

La Figura 2.13 muestra un diagrama de bloques para estos buses.

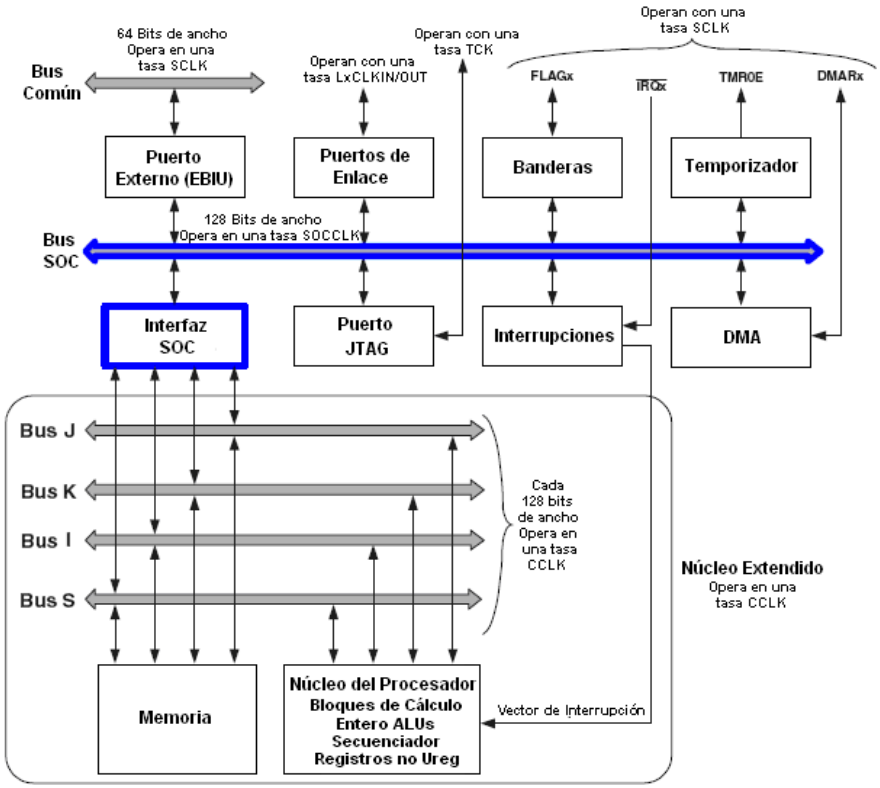

Figura. 2. 13. Conexiones del bus y la interfaz SOC

El bus SOC tiene un esquema de prioridad fijo para arbitrar entre los maestros que requieran realizar una operación sobre el bus SOC. De la más alta a la más baja la prioridad de requerimiento del bus SOC es:

1. Puerto externo (alta prioridad)
2. DMA (alta prioridad)

3. Interfaz SOC
4. Puerto externo (baja prioridad)
5. DMA (baja prioridad)

Cuando la operación del bus SOC ha arbitrado por el bus maestro y alcanza la interfaz SOC, la operación debe competir para acceder al bloque de memoria. El procesador usa un orden de prioridad cuando se arbitra entre dos o mas buses internos realizando accesos simultáneos al mismo bloque de memoria interna.

2.1.7 Temporizadores

El procesador TigerSHARC tiene dos temporizadores de 64 bits de propósito general: *timer0 y timer1*. Los temporizadores son contadores de corrida libre que son cargados con un valor inicial, decrementan su valor hasta cero (expirando), dan una indicación cuando han expirado, automáticamente recargan el valor y continúan corriendo. El indicador de expiración es normalmente una interrupción, pero puede ser también un pin de salida externa del *timer0*.

Cada temporizador contiene dos registros de 64 bits. Un registro es el valor inicial (TMRINxH/L) y el otro es el valor de corrida (TIMERxH/L).

Para mayor información, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_Reference* contenido en el CD de EZ-KITLite.

2.1.8 Banderas

El procesador TigerSHARC tiene cuatro pines de bandera de entrada o salida de propósito general (FLAG3-0). La configuración de estos pines es realizada a través del registro *FLAGREG* que fue revisado previamente.

Cada uno de estos pines puede ser configurado como una entrada o una salida. Como salidas, los programas pueden usar los pines de bandera para señalar eventos o condiciones para cualquier dispositivo externo tal como un procesador host. Como entradas, los programas pueden leer el estado de la entrada del pin de la bandera en el registro FLAGREG o usar el estado de la entrada del pin como una condición en instrucciones condicionales.

2.1.9 Interrupciones

El procesador TigerSHARC soporta una variedad de interrupciones (indicadores de eventos internos o externos). Las interrupciones pueden servir para cualquiera de los siguientes propósitos en un sistema:

- Sincronización entre el núcleo y operaciones fuera del núcleo
- Detección de error
- Operaciones de características de depuración
- Controles introducidos por una aplicación

Las interrupciones en el procesador TigerSHARC están clasificadas como interrupciones de software o interrupciones de hardware. Para cada interrupción existe un registro de vector en la tabla de vector de interrupciones (IVT) (en grupos de registros 0x38 y 0x39) y un número de bit en los registros de seguro de interrupción, máscara, y prioridad de máscara (PMASK).

Las interrupciones son de nivel sensitivo o borde sensitivo (las interrupciones de hardware pueden ser configuradas ya sea como nivel o borde sensitivo de acuerdo a la programación en el registro INTCLT).

2.1.10 Acceso Directo a Memoria

El acceso directo a memoria (DMA) es un mecanismo para transferencia de datos sin ejecutar instrucciones en el núcleo del procesador. El controlador DMA permite al

núcleo del procesador TigerSHARC, o a un dispositivo externo, especificar operaciones de transferencia de datos y retornar a un procesamiento normal mientras el controlador DMA realiza la transferencia de datos en un segundo plano.

El procesador TigerSHARC incluye 14 canales DMA de los cuales cuatro están dedicados para transferencia de datos hacia y desde dispositivos de memoria externa (incluyendo otros procesadores TigerSHARC), ocho para los puertos de enlace y dos para los registros AutoDMA. Estos canales DMA permiten los siguientes tipos de transferencia:

- Memoria interna hacia y desde memoria externa o periféricos de mapa de memoria
- Memoria interna hacia y desde memoria interna de otro procesador TigerSHARC a través el bus común.
- Memoria interna hacia y desde el procesador host
- Memoria interna hacia y desde puertos de enlace de I/O
- Memoria externa o periféricos del mapa de memoria hacia y desde puertos de enlace de entrada y salida
- Memoria externa hacia y desde periféricos externos
- Puertos de enlace de I/O hacia y desde puerto de enlace de I/O
- Bus común maestro hacia y desde memoria interna esclava a través AutoDMA

2.1.11 Puerto externo e interfaz SDRAM

El procesador TigerSHARC puede ser usado como un procesador simple o como un elemento en un sistema multiprocesador. El procesador puede actuar como un sistema o procesador TigerSHARC autónomo, o puede ser controlado por una computadora *host*. La arquitectura del sistema es flexible y puede ser implementado de acuerdo a los requerimientos de la aplicación.

El protocolo mas rápido es el protocolo *pipelined*. El procesador TigerSHARC usa este protocolo para interactuar con otros procesadores TigerSHARC dentro de un grupo.

El procesador puede además usar este protocolo para interactuar con un *host* y sistemas de memoria inteligente.

La unidad de interfaz del bus externo puede ser configurada de diferentes formas escribiendo en los diferentes registros de control de puerto externo (SYSCON y SDRCON).

Estos dos registros pueden ser inicializados en cualquier instante durante la operación del sistema o pueden ser restringidos para ser escritos solo una vez.

- **Características del bus externo**

El bus externo incluye:

- Ancho de bus: 64 o 32 bits, configurados por memoria, multiprocesamiento o interfaz host en forma separada.
- Estados IDLE programables
- Protocolo que soporta ciclos de espera ingresados por el esclavo previsto, usando el pin ACK
- Interfaz EPROM y FLASH: bus de datos de 8 bits con un número fijo de ciclos de espera, lectura o escritura.
- Interfaz *Host*
- Interfaz SDRAM: no requiere ciclos de espera.
- Respaldo para dispositivos lentos de I/O.
- Multiprocesamiento independiente con otro procesador TigerSHARC, basado en una arbitración de bus distribuido.
- Soporte de operaciones DMA para dispositivos de I/O externos a través del modo *handshake*.
- Respaldo para arbitración del bus común (solo respuesta) mientras el procesador esta en el estado de reset de software.

2.1.12 Puertos de enlace

El procesador TigerSHARC ADSP-TS201 tiene cuatro puertos de enlace *full-duplex*. Los puertos de enlace proveen un canal de comunicación opcional. Este canal tiene el propósito de ser usado para una comunicación punto a punto entre procesadores TigerSHARC en un sistema, pero puede ser usado además para interactuar con cualquier otro dispositivo que este diseñado para trabajar en el mismo protocolo.

Los datos son lanzados y enviados tanto en flanco alto y bajo del reloj de enlace. El control de transferencia en el núcleo ocurre a través de interrupciones. El control de transferencia en el DMA ocurre a través de los canales DMA dedicados para transmitir y recibir. El encadenamiento DMA es soportado, y todos los puertos de enlace pueden ser usados para arrancar.

Este puerto de enlace tiene dos partes: transmisor y receptor. El canal de transmisión tiene un doble *buffer* y el canal de recepción tiene un triple *buffer* (Figura 2.14). Los registros LBUFTXx y LBUFRXx son registros *Ureg* del mapa de memoria de 28 bits. Los registros de desplazamiento no son accesibles por software.

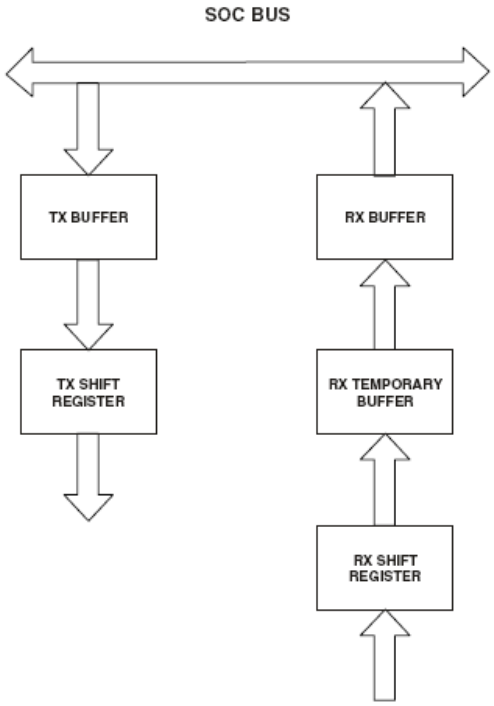

Figura. 2.14. Arquitectura del puerto de enlace

Cada puerto de enlace tiene dos canales independientes que pueden operar simultáneamente. El canal de transmisión transfiere datos a un segundo dispositivo, y el canal receptor obtiene los datos de un segundo dispositivo. Cada canal se comunica usando hasta 4 bits de datos, usando las señales LxCLKOUTP/N, LxACKI, LxCLKINP/N y LxACKO para controlar la transferencia de datos.

Los puertos de enlace del procesador ADSP-TS201 llegan a ser puertos de datos de alta velocidad LVDS. LVDS es un estándar para señalización diferencial entre elementos distantes con más altas velocidades que técnicas de finalización simple. Esta técnica permite la transmisión de datos entre dos procesadores TigerSHARC ADSP-TS201. LVDS provee más alta frecuencia, mayor inmunidad al ruido, menor consumo de potencia y menor interferencia electromagnética.

- **Datos de transmisión y recepción de enlace**

La transferencia de datos se la realiza escribiendo en el *buffer* de transmisión y leyendo del *buffer* de recepción. Todos los datos escritos en el *buffer* de transmisión son copiados al registro de desplazamiento una vez que esté esta vacío, y después son transmitidos. El receptor permite que los datos ingresen, solo cuando el registro de desplazamiento de recepción esta vacío, o cuando existe espacio en los *buffers* de recepción para aceptar los datos desde el registro de desplazamiento cuando la palabra cuádruple recibida esta completa. Después de que la palabra cuádruple entera ha sido recibida, el receptor mueve los datos desde el registro de desplazamiento al *buffer* de recepción cuando este esta libre. El control de enlace mueve los datos al *buffer* del receptor como su primera prioridad. Si el *buffer* de recepción esta lleno, este copia los datos dentro del *buffer* de recepción temporal y los mantiene ahí hasta que el *buffer* de recepción este libre. El receptor controla el flujo de datos no validando la señal LxACKO. Hasta dos palabras cuádruples pueden ser desplazadas después de la no validación de la señal LxACKO.

- **Enlace DMA**

Cada puerto de enlace esta asociado con dos canales DMA. Un canal es usado para transmitir datos mientras el otro es usado para recibir. Los dos canales DMA pueden interactuar con la memoria interna, memoria externa u otros *buffers* de puerto de enlace.

El puerto de enlace realiza un requerimiento de servicio al canal DMA de transmisión cuando del registro LBUFTXx está vació y el canal DMA esta habilitado. El puerto de enlace realiza un requerimiento DMA al canal de enlace DMA de recepción cuando este recibe una palabra cuádruple lo que significa que el registro LBUFRXx está lleno y el canal DMA está habilitado.

Para mayor información, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_Reference* contenido en el CD de EZ-KITLite.

2.2 SOFTWARE ADSP- TS201

El procesador TigerSHARC es un procesador DSP de tipo estático súper escalar que ejecuta desde una hasta cuatro *slots* de instrucciones de 32 bits en una línea de instrucción. Con pocas excepciones, las instrucciones se ejecutan con una tasa de transferencia de una línea de instrucción (4 *slots* de instrucciones) por ciclo de reloj del núcleo del procesador (CCLK).

La Figura 2.15 muestra el *slot* de instrucción y la estructura de la línea.

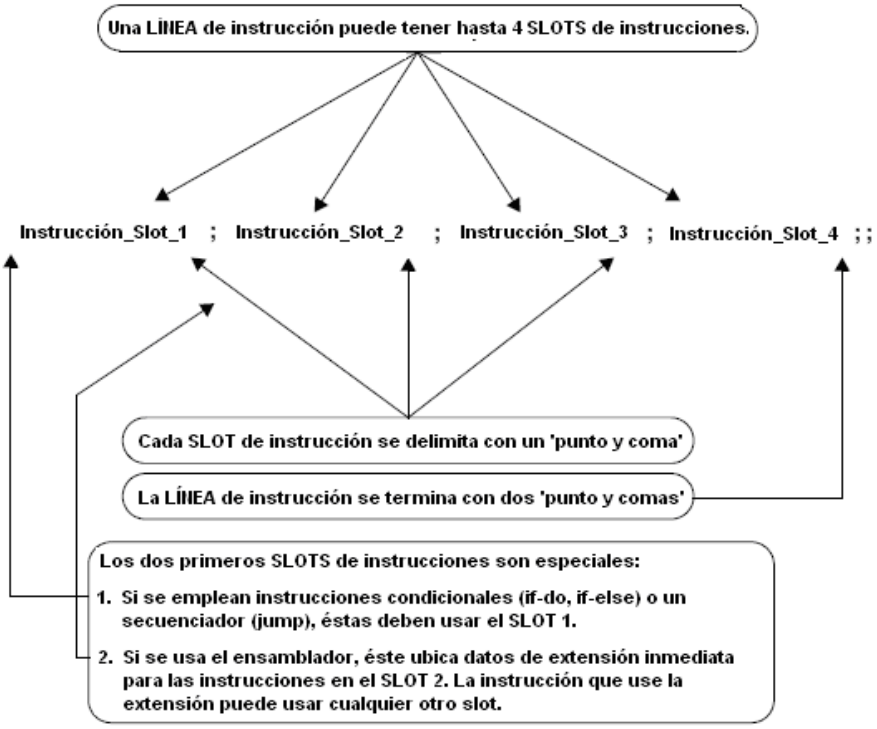

Figura. 2. 15. Estructura del *slot* y línea de instrucción

Es importante observar ciertas características del *slot* instrucción y la estructura de la línea de instrucción, y cómo esta estructura se relaciona con la ejecución de instrucción.

- Cada línea de instrucción consiste de hasta cuatro *slots* de instrucciones de 32 bits.
- Los *slots* de instrucciones se delimitan con un punto y coma ";"
- Las líneas de instrucción son finalizadas con dos punto y coma ";;"
- Cuatro instrucciones en una línea de instrucción pueden ser ejecutadas en paralelo.

- Instrucciones que usan datos de 32 bits (por ejemplo, direcciones directas de 32 bits o datos inmediatos de 32 bits) requieren de dos *slots* de instrucciones para ser ejecutadas; un *slot* para la instrucción y un *slot* para las extensiones inmediatas.
- El secuenciador del programa y las instrucciones de extensión inmediata requieren *slots* de instrucciones específicas. El *slot* de extensión inmediato es asignado por el ensamblador; una extensión inmediata no es una instrucción escrita específicamente por el programador.

Una instrucción es una palabra de 32 bits que activa uno o más unidades de ejecución del procesador TigerSHARC para llevar acabo una operación. El procesador ejecuta o detiene las instrucciones juntas en la misma línea de instrucción.

Sin considerar el tamaño (de una a cuatro instrucciones), las líneas de instrucción siguen una a continuación de la otra en memoria con una nueva línea de instrucción empezando una palabra desde donde la línea de instrucción previa finalizó. El fin de una línea de instrucción esta identificado por el MSB en la palabra de instrucción.

La sintaxis para las instrucciones del procesador TigerSHARC selecciona la operación que el procesador ejecuta y la opción en la cual el procesador ejecuta la operación. Las operaciones incluyen cálculos, movimiento de datos y controles de flujo de programas. Todos los controles sobre la ejecución de instrucción están incluidos en la sintaxis de la instrucción del procesador.

A continuación se presentan instrucciones en formato resumen. Este formato muestra todos los puntos seleccionables y opcionales disponibles para una instrucción. Las notaciones para estos puntos son:

Tabla. 2. 2. Notaciones para una instrucción

this | that | other	Listas de elementos delimitados con una barra vertical "|" indica que esa sintaxis permite selección de uno de los elementos. Uno de los elementos de la lista debe ser seleccionado. La barra vertical no es parte de la sintaxis de la instrucción.
{option}	Un grupo de elementos encerrados dentro de llaves "{ }" indica un elemento opcional. El elemento podría ser incluido u omitido. Las llaves no son parte de la sintaxis de la instrucción.
( ) [ ] , ; ;;	Paréntesis, corchetes, coma, punto y coma, doble punto y coma; y otros símbolos son elementos requeridos en la sintaxis de la instrucción y deben aparecer donde se muestra en resumen la sintaxis con una excepción. Paréntesis vacíos (ninguna opción seleccionada) podrían no aparecer en una instrucción.
Rm Rmd Rmq	Los nombres de los registros son elementos reemplazables en la sintaxis y aparecen en letra cursiva. Los nombres de los registros indican que la sintaxis requiere un registro simple (Rm), doble (Rmd) o cuádruple (Rmq).
< *imm* # >	Datos inmediatos (valores literales) en la sintaxis aparecen como < *imm* # > con # indicando el ancho de bit del valor.

Para mayor información, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el CD.

2.2.1 Modos de operación

El procesador TigerSHARC opera en uno de los tres modos:

- Modo usuario
- Modo supervisor
- Modo emulador

En modo usuario y supervisor, todas las instrucciones son ejecutadas normalmente. En modo usuario, sin embargo, el acceso al registro es limitado. Sin considerar el modo de operación (usuario, supervisor y emulador), todas las instrucciones son ejecutadas normalmente.

El modo de operación en que trabaje el procesador TigerSHARC determina cuales de los componentes del procesador están activos y pueden ser accedidos. El modo además, determina cuales excepciones son consideradas y como pueden ser manejadas. Las prioridades de los diferentes modos desde la más baja a la más alta son: modo usuario, modo supervisor y modo emulador.

- **Modo Usuario**

La operación del Modo Usuario es usada para algoritmos y códigos de control que no requieren la manipulación de los recursos del sistema. Muchos recursos del sistema no son accesibles en modo usuario. Si el procesador TigerSHARC intenta acceder a estos recursos, una excepción ocurre.

El modo usuario es con frecuencia usado cuando se corre un código fuera de un sistema de operación.

Los registros que pueden ser accedidos por el programa en modo usuario son:

- Registros del bloque de cálculo.
- Registros IALU.

- Registros del secuenciador: CJMP, registros de contador de lazo LC0 y LC1 y el registro de bandera estática (SFREG).

Todo el resto de registros no pueden ser escritos por el programa núcleo en modo usuario. Un intento para escribir en un registro protegido causa una excepción.

- **Modo Supervisor**

La mayoría de códigos que no están supuestos a ejecutarse bajo un sistema de operación deben estar diseñados para correr en modo supervisor. El modo supervisor permite al programa acceder a todos los recursos del procesador. El procesador TigerSHARC esta en modo supervisor cuando una de estas dos condiciones se cumple:

- El bit NMOD en el registro SQCTL es inicializado.
- Una rutina de interrupciones es ejecutada.

Normalmente cuando el procesador TigerSHARC es reseteado entra en un estado IDLE. El procesador sale del estado IDLE a un estado de ejecución cuando una interrupción es realizada. La interrupción pone al procesador TigerSHARC en modo supervisor. Si el bit NMOD es encerado, el procesador entra en modo usuario después de dejar una rutina de interrupción.

- **Modo Emulador**

El Modo Emulador es usado cuando se controla el procesador con una herramienta emuladora a través del puerto JTAG. El procesador TigerSHARC entra en modo de emulación cuando una excepción de emulación es generada.

Mientras el procesador TigerSHARC esta en modo emulador, la única fuente de instrucciones es el registro EMUIR y la única forma de salir de este modo es ejecutando un retorno de interrupción (RTI).

2.2.2 Registros de bloques de cálculo

Los archivos de registro de los bloques de cálculo X e Y contienen 32 registros de 32 bits los cuales sirven como una interfaz del bloque de cálculo entre bus interno del procesador y las unidades de cálculo. Los registros de archivo de registro (XR31-0 y YR31-0) son registros universales (*Ureg*) y registros de datos (*Dreg*).

Las entradas para cálculos usualmente vienen del archivo de registro. Sin embargo, en algunos casos, las entradas pueden venir de los registros internos de los bloques de cálculo. Los resultados van al archivo de registro, excepto cuando van a los registros internos del bloque de cálculo.

Hay muchas formas para nombrar registros en la sintaxis de ensamblaje del procesador TigerSHARC. La sintaxis del nombre del registro provee la selección de muchas características de instrucciones de cálculo. Al usar la sintaxis de nombre de registro en una instrucción, se puede especificar:

- Selección del bloque de cálculo
- Selección del ancho del registro
- Selección del tamaño del operando
- Selección del formato de datos

En la Figura 2.16 se muestra las partes de la sintaxis del nombre del registro y las características que la sintaxis selecciona.

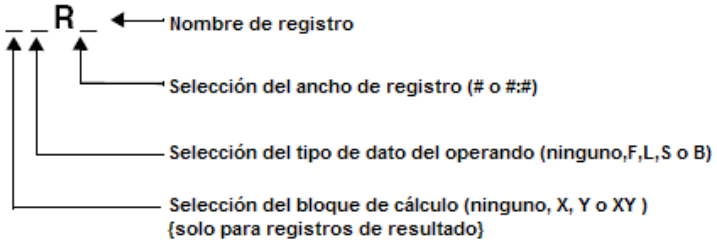

Figura. 2. 16. Sintaxis del nombre de registro del archivo de registro

- **Selección del bloque de cálculo**

Los prefijos del nombre del registro X e Y muestran en cual bloque de cálculo el registro reside: X = solo bloque de cálculo X, Y = solo bloque de cálculo Y y XY (o sin prefijo) = ambos bloques.

- **Selección del ancho del registro**

Cada registro de archivo de registro individual (XR31-0 y YR31-0) tiene un ancho de 32 bits. Para soportar datos de un tamaño mayor a una palabra de 32 bits, la sintaxis del ensamblaje del procesador permite combinar registros para soportar palabras más grandes. La sintaxis del nombre del registro para determinar el ancho del registro funciona:

- *Rs, Rm* o *Rn* indica un registro simple que contiene una palabra de 32 bits o menor.
- *Rsd, Rmd o Rnd* indica un registro doble que contiene una palabra de 64 bits o menor.
- *Rsq, Rmq o Rnq* indica un registro cuádruple que contiene una palabra de 128 bits o menor.

- **Selección del tipo de operando**

Para seleccionar el tamaño del operando dentro de un registro de archivo de registro, un prefijo de nombre de registro selecciona un tamaño que es igual o menor que el tamaño del registro. Estos prefijos para datos de punto-fijo funcionan:

- B - Indica una palabra de 1 byte. Los datos en un registro simple de 32 bits son tratados como cuatro palabras de 8 bits.
- S – Indica datos de palabra corta (16 bits). Los datos en un registro simple de 32 bits son tratados como dos palabras de 16 bits.
- Ninguno – Indica datos de palabra normal (32 bits).
- L – Indica datos de palabra larga (64 bits).

Las opciones B, S y L aplican para operaciones ALU y de desplazamiento. La selección del tamaño de los operandos difiere ligeramente para el multiplicador.

Para distinguir entre datos de punto-fijo y flotante, el prefijo F del nombre de registro indica que el registro contiene datos punto-flotante. El procesador soporta los siguientes formatos de punto flotante.

- Ninguno – Indica datos de punto-fijo
- *FRs, FRm o FRn* (datos de punto-flotante en un registro simple): indica datos de palabra normal (32 bits).
- *FRsd, FRmd o FRnd* (datos de punto-flotante en un registro doble): indica datos de palabra extendida (40 bits).

Es importante observar que el tamaño del operando influye en la ejecución de la instrucción. Por ejemplo, *SRsd = Rmd + Rnd* es una suma de cuatro operandos de datos cortos, almacenados en dos pares de registros.

2.2.3 Unidades de cálculo

- **ALU**

En relación a las operaciones con la ALU y los tipos de datos soportados se muestra que la unidad ALU de 64 bits dentro de cada bloque de cálculo soporta:

- Operaciones aritméticas con punto-fijo y flotante:
  - Suma (+)
  - Resta (-)
  - Mínimo (MIN)
  - Máximo (MAX)
  - Comparación (COMP)
  - Extracción (CLIP)
  - Valor absoluto (ABS)

- ❖ Operaciones aritméticas solo con punto-fijo.
  - Incremento (INC)
  - Decremento (DEC)
  - Suma lateral (SUM)
  - Complemento a 1 (ONES)

- ❖ Operaciones de conversión de datos (ascenso-descenso) solo con punto-fijo.
  - Expansión (EXPAND)
  - Reducción (COMPACT)
  - Combinación (MERGE)

- ❖ Operaciones lógicas.
  - AND
  - AND NOT
  - OR
  - XOR
  - PASS

- ❖ Operaciones aritméticas solo con punto-flotante.
  - Conversión a punto-flotante (FLOAT).
  - Conversión a punto-fijo (FIX).
  - Copia de signo (COPYSIGN)
  - Escalamiento (SCALB)
  - Raiz cuadrada (RSQRTS)
  - Sustracción de mantissa (MANT)
  - Extracción del exponente (LOGB)
  - Extensión de operando (EXTD)
  - Cambio a extendido de un operando simple (SNGL)

Examinando los operandos soportados por cada operación se muestra que las operaciones ALU soportan los siguientes tipos de datos.

- En operaciones aritméticas con punto-fijo.
  - Operandos de entrada de 8 bits (byte)
  - Operandos de entrada de 16 bits (corto)
  - Operandos de entrada de 32 bits (normal)
  - Operandos de entrada de 64 bits (largo)
  - Resultados de salida 8, 16, 32 o 64 bits.

- En operaciones de conversión de datos con punto-fijo.
  - De 8 bits a 16 bits
  - De 16 bits a 32 bits De 32 bits a 16 bits
  - De 64 bits a 32 bits
  - De 128 bits a 64 bits

- En operaciones lógicas.
  - Operandos de entrada de 32 bits (normal)
  - Operandos de entrada de 64 bits (largo)
  - Resultados de salida 32 o 64 bits.

- En operaciones aritméticas con punto-flotante.
  - Operandos de entrada de 32 bits (normal)
  - Operandos de entrada de 40 bits (extendido)
  - Resultados de salida 32 o 40 bits.

- **Operaciones ALU**

El procesador usa registros de bloque de cálculo para los operandos de entrada y resultados de salida de las operaciones ALU. Los registros de archivo de registro del bloque de cálculo son del XR31 al XR0 y del YR31 al YR0. La ALU tiene un registro doble de propósito especial, el registro PR, para resultados paralelos.

Todas las instrucciones ALU generan banderas de estado para indicar el estado del resultado si la opción NF (*no flag update*) no es usada.

- **Opciones de instrucción ALU**

La mayoría de las instrucciones ALU tienen opciones asociadas con ellas que permitan flexibilidad en como las instrucciones se ejecutan. Estas opciones modifican las ejecuciones detalladas de instrucciones y opciones que son particulares para un grupo de instrucciones (no todas las opciones son aplicadas para todas las instrucciones). Las opciones de las instrucciones aparecen en paréntesis en el final del *slot* de instrucción.

Las opciones de instrucción ALU incluyen:

(.)	Operación con signo, sin saturación, redondeo al par más cercano, operación fraccionaria.
(S)	Operación con signo, con saturación
(U)	Operación sin signo, sin saturación, redondeo al par más cercano.
(SU)	Operación sin signo, con saturación.
(X)	Extender operación para ABS.
(T)	Operación con signo, truncar.
(TU)	Operación sin signo, truncar.
(Z)	Resultado con signo retorna a cero de la operación para MIN/MAX.
(UZ)	Resultado sin signo retorna a cero de la operación para MIN/MAX.
(I)	Operación con signo, operación entera
(IU)	Operación sin signo, operación entera
(IS)	Operación con signo, con saturación, operación entera
(ISU)	Operación sin signo, con saturación, operación entera
(NF)	Sin actualización de bandera.

Para mayor información de las opciones de instrucción ALU, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el *CD EZ-Kit Lite*.

- **Estado de la ejecución ALU**

Las operaciones ALU actualizan las banderas de estado en el registro de estado aritmético del bloque de cálculo (XSTAT e YSTAT). Los programas pueden usar las banderas de estado para controlar la ejecución de instrucciones condicionales e iniciar interrupciones de excepción de software.

Para mayor información acerca de las banderas de estado ALU, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el *CD EZ-Kit Lite*.

- **Multiplicador**

El multiplicador toma sus entradas del archivo de registro y retorna sus salidas al archivo de registro o registros MR (acumulador).

Las operaciones del multiplicador soportan los siguientes tipos de datos:

- Operaciones de multiplicación y multiplicación-acumulación con fraccionarios y enteros punto-fijo soportan:
  - 8 operandos de entrada de 16 bits con 4 resultados de 16, 20, 32 o 40 bits.
  - 2 operandos de entrada de 32 bits con un resultado de 32, 40, 64 u 80 bits.

- Operaciones de multiplicación-acumulación complejas con fraccionario y entero punto-fijo soportan:
  - 2 operandos de entrada de 32 bits (16 bits reales y 16 bits imaginarios) con un resultado de 40 bits (20 bits reales y 20 bits imaginarios) o un resultado de 80 bits (40 bits reales y 40 bits imaginarios).

- Operaciones de multiplicación con fraccionarios punto-flotante soportan:

Dos operandos de entrada de 32 bits (precisión simple) con un resultado de 32 bits.

Dos operandos de entrada de 40 bits (precisión extendida) con un resultado de 40 bits.

- Operaciones de compactación de datos con punto-fijo soportan:
  - Operandos de entrada de 20 bits (cortos).
  - Operandos de entrada de 40 bits (normal).
  - Operandos de entrada de 80 bits (largos).
  - Resultados de 16 o 32 bits de salida.

- **Operaciones del multiplicador**

El multiplicador tiene un registro de 5 palabras de propósito especial (registro MR) para resultados acumulados. El procesador ADSP-TS201 utiliza el registro MR para almacenar los resultados de operaciones de multiplicación-acumulación de punto-fijo. Además, el multiplicador puede transferir los contenidos del registro MR al archivo de registro antes de una operación de acumulación. Los 32 bits superiores del registro (MR4) almacenan el desbordamiento para operaciones de multiplicación-acumulación.

- **Opciones de instrucción de multiplicación**

Las opciones de instrucción de multiplicador incluyen:

( ) Operación con signo, sin saturación, redondeo al par mas cercano, operación fraccionaria.

(U) Operación sin signo, sin saturación, redondeo al par más cercano, operación fraccionaria.

(nU) El operando Rm tiene entrada con signo, el operando Rn tiene entrada sin signo, sin saturación, redondeo al par mas cercano, operación fraccionaria.

(I) Operación con signo, operación entera.

(S) Operación con signo, con saturación.

(T) Operación con signo, truncamiento.

(C) Encera la operación.

(CR) Operación de redondeo y encerado.

(J) Operación de conjugado complejo.

(NF) Sin actualización de bandera.

Para mayor información de las instrucciones del multiplicador, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el *CD EZ-Kit Lite*.

- **Desplazador**

El desplazador tiene tres entradas y una salida de 64 bits. Los recorridos de entrada y salida del desplazador dentro del bloque de cálculo tienen varias implicaciones para el paralelismo de instrucciones.

- Las instrucciones del desplazador que usan tres entradas no pueden ser ejecutadas en paralelo con ninguna otra operación del bloque de cálculo.
- Para instrucciones FDEP, MASK, GETBITS y PUTBITS, existen tres registros que son ubicadas dentro del desplazador. Esta operación usa tres puertos de bloque de cálculo. La salida es ubicada en el mismo puerto.

Dentro de las instrucciones, la sintaxis de nombre de registro identifica el tipo y tamaño de dato del operando de entrada y de salida.

- **Operaciones del desplazador**

El desplazador opera sobre los archivos de registro del bloque de cálculo y opera sobre el registro desplazador BFOTMP (un registro desplazador interno el cual es usado para la instrucción PUTBITS). Las operaciones de desplazamiento pueden tomar su entrada Rm del archivo de registro y tomar su entrada Rn del archivo de registro o de datos inmediatos provistos en la instrucción.

- **Opciones de instrucción de desplazamiento**

Las instrucciones del desplazador incluyen:

( ) Llenado de ceros, justificado a la derecha.
(SE) Signo extendido, aplica a instrucciones FEXT, FDEP y GETBITS.
(ZE) Llenado de ceros; aplica a instrucción FDEP.
(NF) Sin actualización de bandera

Para mayor información de las instrucciones del desplazador, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el *CD EZ-Kit Lite*.

- **IALU**

El núcleo del procesador contiene dos Unidades Aritméticas Lógicas Enteras conocidas como IALUs. Las operaciones enteras de la IALU incluyen:

- Suma y resta.
- Desplazamientos aritméticos y lógicos a la derecha.
- Operaciones Lógicas.
- Funciones: Valor absoluto, mínimo, máximo y comparación.

Las IAULs proveen direcciones de memoria cuando se transfiere datos entre memoria y registros. IALUs dobles permiten direcciones simultáneas para dos accesos de memoria (lectura o escritura). Las operaciones de carga, almacenamiento y transferencia de datos de la IALU incluyen:

- Direccionamiento de memoria directa e indirecta.
- Direccionamiento de *buffer* circular.
- Movimiento y carga de registros Universales.
- Generación del puntero de memoria.

Cada instrucción especifica si una palabra normal, larga o cuádruple es accedida por cada bloque de memoria. Debido a que el procesador tiene dos IALUs, dos bloques de memoria pueden ser accedidos por cada ciclo.

Cada IALU contiene un archivo de registro de datos de 32 registros y ocho registros dedicados para direccionamiento de *buffer* circular. Todos los registros en la IALU son registros universales y tienen un ancho de 32 bits. Algunos puntos importantes son relacionados a los archivos de registro IALU son:

- Los registros del archivo de registro J-IALU son J30-J0 y los de K-IALU son K30- K0. A excepción de J31 y K31, estos registros son de propósito general y contienen solo datos enteros.
- Los registros J31 y K31 son registros de estado de 32 bits y pueden además estar referidos como JSTAT (J31) y KSTAT (K31).

Los registros dedicados para direccionamiento de *buffer* circular en cada IALU seleccionan la dirección base y la longitud del *buffer*. Estos registros dedicados trabajan con los cuatro primeros registros de propósito general de cada IALU para manejar hasta ocho *buffers* circulares. Algunos puntos importantes relacionados a los registros dedicados IALU son:

- El *índice* del *buffer* circular (dirección actual) es establecido por un registro de propósito general. Estos son J3-J0 en la J-IALU y K3-K0 en la K-IALU.
- La *base* del *buffer* circular (dirección inicial) es establecida por un registro dedicado. Estos son JB3-JB0 en la J-IALU y KB3-KB0 en la K-IALU.
- La *longitud* del *buffer* circular (número de ubicaciones de memoria) es establecido por un registro dedicado. Estos son JL3-JL0 en la J-IALU y KL3-KL0 en la K-IALU.
- El *modificador* del *buffer* circular (el tamaño entre las ubicaciones de memoria) es establecido por cualquier registro IALU de propósito general o un valor inmediato. El modificador no puede ser más grande que la longitud del *buffer* circular.

- Los registros de índice, base y longitud para controlar los *buffers* circulares trabajan como una unidad (J0 con JB0 y JL0).

Los IALUs pueden usar instrucciones de suma y resta para generar punteros de memoria con o sin direccionamiento de *buffer* circular.

2.2.4 Temporizadores

Los temporizadores son inicializados en estado IDLE después de resetear (los bits TMRxRN son encerados en el registro INTCTL). Para iniciar y ejecutar un temporizador, la aplicación inicializa el bit TMRxRN correspondiente. Cuando este bit es inicializado, el valor en TMRINxH/L es copiado al registro TIMERxH/L, y el temporizador empieza a contar regresivamente, una cuenta cada ciclo de reloj de interfaz SOC interna (SOCCLK = CCLK/2).

Los programas pueden detener el temporizador encerando el bit TMRxRN y reanudar la cuenta del temporizador desde ese punto inicializando el bit TMRxRN. Esto significa que la carga del registro TIMERxH/L no es afectada por la operación de los bits TMRxRN. Solo una escritura en el registro TMRINxH/L carga al correspondiente registro TIMERxH/L con un nuevo valor inicial.

Cuando la cuenta del temporizador alcance a cero, el temporizador establece las interrupciones de expiración de los dos temporizadores (alta y baja prioridad), reinicializa el contador a su valor inicial, y empieza a ejecutar nuevamente. Si el temporizador esta activo (el bit TMRxRN es inicializado), el escribir un valor en el registro TMRINxH/L no tiene efecto. Escribir un diferente valor en el registro TMRINxH/L mientras el temporizador está operando cambia el valor inicial después de la siguiente expiración del temporizador.

Cuando un bit de interrupción para un temporizador (INT_TIMER0LP, INT_TIMER1LP, INT_TIMER0HP o INT_TIMER1HP) en el registro ILATx es inicializado o encerado escribiendo en los registros ILATSTx o ILATCLx, ambas interrupciones, la de baja y alta prioridad, son inicializadas o enceradas.

Se usa el siguiente procedimiento para controlar los temporizadores:

1. (Si el temporizador esta actualmente ejecutándose) deshabilite el temporizador encerando su bit TMRxRN en el registro INTCTL.
2. (Si una interrupción esta como salida) configure la prioridad de la interrupción del temporizador baja y/o alta.
    a) Asigne las direcciones del vector interrupción para las rutinas cargando los registros del vector interrupción (IVTIMERxH/LP).
    b) Habilite las interrupciones del temporizador inicializando los bits INT_TIMERxH/LP en el registro IMASK.
    c) Habilite las interrupciones globales inicializando el bit SQCTL_GIE en el registro SQCTL.

Si la única salida es el pin TMR0E, los programas no necesitan configurar la interrupción del temporizador.

3. Cargue el valor inicial en los registros TMRINxH/L.
4. Ejecute el temporizador inicializando su bit de corrida TMRxRN en el registro INTCTL.

- **Pin de expiración de temporizador (TMR0E) y señal de tiempo.**

El pin de expiración del *timer0* (TMR0E) indica que el *timer0* ha expirado. Cuando el *timer* expira, el procesador emite un pulso en alto sobre el pin TMR0E para cuatro ciclos SCLK. Debido a que los accesos del núcleo del procesador para controlar el temporizador deben cruzar los dominios de reloj, la respuesta del temporizador puede verse afectada por la carga de la interfaz SOC ya que el temporizador es un bus esclavo sobre el bus SOC.

2.2.5 Banderas

Para configurar y usar los pines de bandera de entrada y salida, se usa el siguiente procedimiento:

1. Use los bits FLAGx_EN en el registro FLAGREG para configurar cada uno de los pines FLAG3-0 como una entrada (FLAGx_EN=0) o salida (FLAGx_EN=1).
2. (Para pines de bandera de salida) use los bits FLAGx_OUT en el registro FLAGREG para seleccionar los valores de salida para cada uno de los pines FLAG3-0.
3. (Para pines de bandera de entrada) lea los bits FLGx en el registro SQSTAT para ver los valores de entrada para cada uno de los pines FLAG3-0 o use las condiciones de bandera de entrada (FLAGx_IN) en instrucciones condicionales.

- **Pin de bandera (FLAG3-0) y señal de tiempo**

Los pines de bandera de entrada y/o salida permiten al procesador enviar señales a dispositivos externos o recibir entradas de ellos.

Para mayor información, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Hardware_Reference* contenido en el CD de EZ-KITLite.

2.2.6 Interrupciones

Las fases de las interrupciones en el procesador incluyen configuración de interrupción, manejo de interrupción y retorno desde la interrupción. El controlador de la interrupción incluye los siguientes registros de control y configuración de interrupción:

- Registro de control de interrupción (INTCTL).
- Registros de máscara de interrupción (IMASK).
- Registros de máscara de prioridad (PMASK).
- Registros de *latch* de interrupción (ILAT).

- **Rutinas de servicio de interrupción**

El siguiente algoritmo describe la rutina de servicio de interrupción (ISR):

1. El vector de interrupción en la tabla de vector de interrupciones (IVT) es usado como la siguiente dirección de búsqueda y la primera instrucción de ISR es empujado dentro del *pipeline*.

Mientras una interrupción esta en el *pipeline,* todas las interrupciones de *hardware* son deshabilitadas aunque esto no se refleje en los registros mascara.

En un ISR, la primera instrucción no debe ser RDS o RTI.

2. Las instrucciones en el *pipeline* se ejecutan.
3. Antes que la primera instrucción en la rutina de interrupción alcance la etapa EX2 en el *pipeline*, el habilitador de la interrupción global es revisado nuevamente. Si el bit fue encerado debido a que la interrupción fue generada anteriormente por una instrucción en el *pipeline*, entonces todas las instrucciones en el *pipeline* son abortadas y el procesador TigerSHARC empieza la búsqueda desde el flujo normal.
4. El contador de programa (PC) de retorno (el cual apunta a la instrucción que habría sido ejecutada) es guardado en el registro RETI para una interrupción de *hardware*, RETS para una interrupción de *software*.
5. Cuando la primera instrucción en la rutina de interrupción alcanza la etapa EX2 en el *pipeline* y la interrupción es disparada por borde, el correspondiente bit de interrupción en el registro ILAT es reseteado, y el mismo bit en el registro PMASK es inicializado.

Los pasos de ISR de *hardware* son:

1. El procesador determina el estado después de la interrupción de *hardware*.
2. El procesador determina si las máscaras están en set.
3. El procesador verifica si el GIE esta en set cuando la primera instrucción en la ISR alcanza la etapa EX2 en el *pipeline.*
4. Si existe una interrupción de *hardware* anidada, la ISR debería empezar guardando el contexto del procesador y los contenidos de RETIB.
5. La ISR se ejecuta, dando revisión a la interrupción.
6. Si la interrupción anidada fue deshabilitada, la ISR debería finalizar restableciendo los contenidos de RETIB, el contexto del procesador y ejecutando una instrucción RTI.
7. Después de que la ISR se complete, el procesador encera el PMASKN y salta a la dirección RETI.

El retorno de interrupción se realiza usando la instrucción RTI. La dirección de retorno debería ser puesta en el registro RETIB (alrededor de 8 ciclos antes de habilitar el uso del BTB). En rutinas de interrupción de *hardware* no anidadas, la dirección de retorno esta ya en el registro RETI.

2.2.7 Acceso Directo a Memoria

Las operaciones DMA pueden ser programadas por el núcleo del procesador TigerSHARC, por un procesador de un *host* externo o por el bus maestro del procesador TigerSHARC externo. La operación es programada escribiendo en los registros DMA TCB del mapa de memoria. El registro TCB es un registro de palabra cuádruple que indica la transferencia del bloque DMA. Un canal DMA es configurado escribiendo una palabra cuádruple en cada uno de los registros DMA TCB. Cada registro debe ser cargado con una dirección de inicio para el bloque, un modificador de dirección y un contador de palabra. Similar a los canales de enlace DMA, un canal de registro AutoDMA tiene solo un registro TCB.

En el caso de puertos de enlace, una vez que un bloque DMA es configurado y habilitado, las palabras de datos recibidas son automáticamente transferidas a la memoria interna, a la memoria externa u otro *buffer* de transmisión del puerto de enlace. De igual manera, cuando el enlace esta listo para transmitir datos, una palabra cuádruple es automáticamente transferida desde la memoria interna o externa hacia el *buffer* del enlace transmisor. Estas transferencias continúan hasta que todo el *buffer* de datos es recibido o transmitido.

Si el TCB es programado para realizar una interrupción, entonces las interrupciones DMA son generadas cuando todo un bloque de datos ha sido transferido. Esto ocurre cuando el registro contador del canal DMA se ha decrementado hasta cero y ultimo elemento de los datos ha sido transferido.

2.2.8 Registros de transferencia y control del bloque DMA (TCB DMA)

Cada uno de los cuatro canales de memoria externa DMA es controlado por un registro par TCB. Además, cada uno de los cuatro puertos de enlace hacia o desde los canales de memoria DMA son controlados por un registro TCB; al igual que, cada uno de los registros AutoDMA hacia los canales de memoria DMA.

- **Registros de transferencia y control del bloque (TCB)**

Cada registro TCB tiene una longitud de 128 bits y esta dividido en cuatro registros de 32 bits. Estos registros se muestran en la Figura 2.17:

- Registro DMA índice (DI).
- Registro DMA dimensión X de incremento y conteo (DX).
- Registro DMA dimensión Y de incremento y conteo (DY).
- Registro DMA de puntero de control y encadenamiento (DP).

Figura. 2.17. Registro TCB

a) Registro DIx

Este es un registro índice de 32 bits para el DMA. Este puede apuntar a la dirección de memoria interna, externa o puertos de enlace.

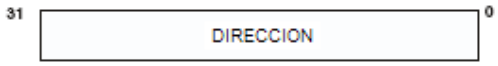

Figura. 2. 18. Registro DIx

b) Registro DXx

Si un DMA de dos dimensiones es deshabilitado, este registro contiene los valores de 16 bits modificados (LSBs) y 16 bits de conteo para el DMA. Si un DMA de dos dimensiones es habilitado, este contiene los valores de dimensión X de los 16 bits modificados y los 16 bits de conteo, donde X de conteo es el número de palabras normales a ser transferidas, y X modificado es el número de palabras normales que modificará el puntero de dirección.

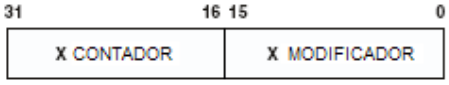

Figura. 2. 19. Registro DXx

c) Registro DYx

Este registro es usado junto con el registro DX cuando un DMA de dos dimensiones es habilitado. Este contiene los valores de dimensión Y de 16 bits

modificados y 16 bits de conteo, donde la Y modificada es la diferencia entre la dirección del ultimo elemento de datos en una fila y la dirección del primer elemento en la siguiente fila.

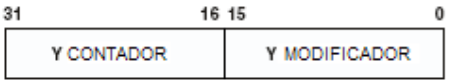

Figura. 2.20. Registro DYx

d) Registro DPx

Este registro es dividido en dos campos, donde el primero es dedicado al control DMA y el segundo es dedicado al encadenamiento.

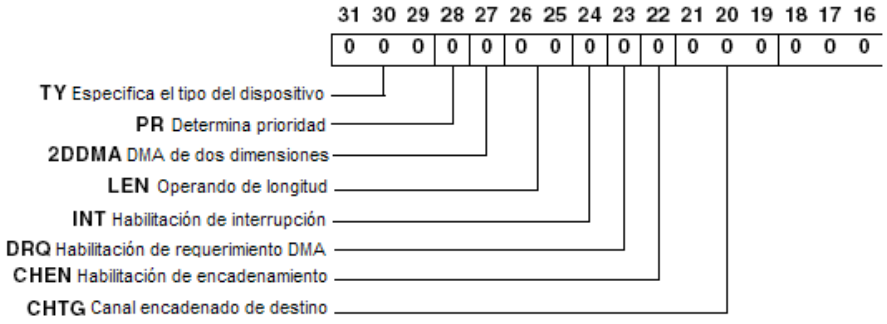

Figura. 2. 21. Descripción de bits (superiores) del registro DPx.

- **Operaciones del controlador DMA**

El controlador DMA soporta:

- Control DMA del puerto de enlace.
- Control DMA del puerto externo.
- Control del registro AutoDMA.

Para mayor información acerca del controlador DMA, hacer referencia al documento *ADSP_TS201_TigerSHARC_Processor_Programming_Reference* contenido en el CD de EZ-KITLite.

.

CAPITULO III

CARACTERÍSTICAS DE LA TARJETA ADSP-TS201 EZ-KIT LITE

3.1 ARQUITECTURA DEL SISTEMA

La Figura 3.1 describe la arquitectura del sistema en la tarjeta EZ-KIT Lite.

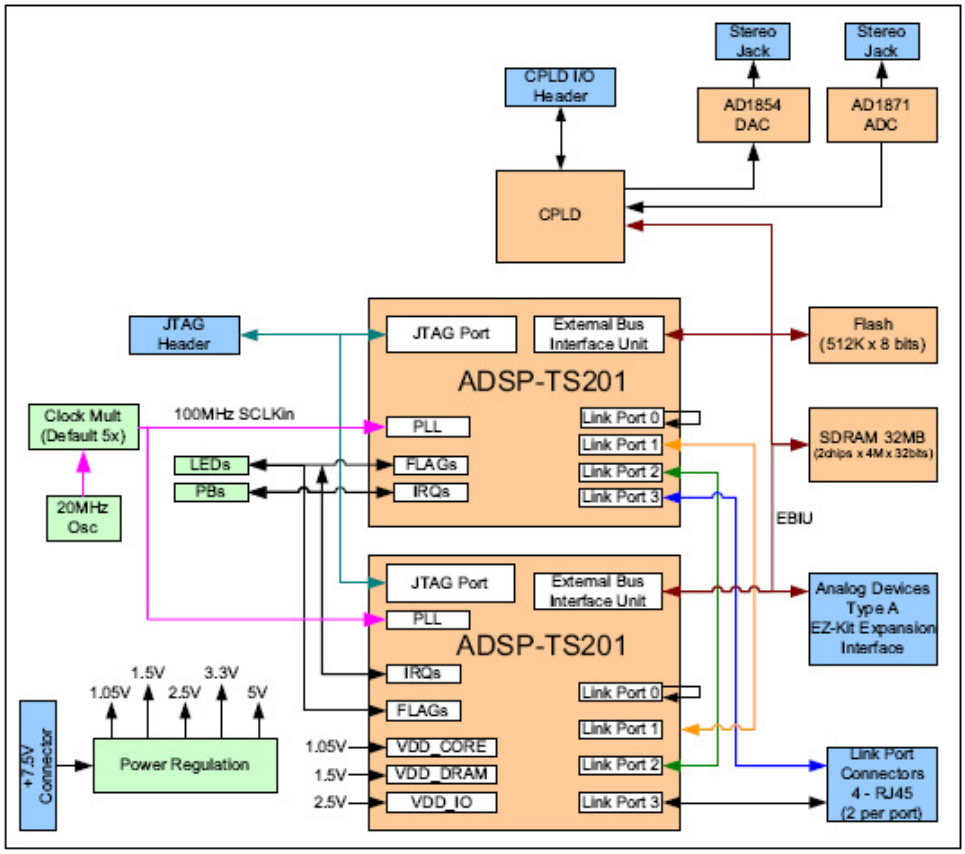

Figura. 3. 1. Arquitectura del Sistema

El EZ-KIT Lite ha sido diseñado para demostrar las capacidades del procesador TigerSHARC ADSP-TS201. El procesador es impulsado por tres reguladores separados: el del núcleo, de la DRAM interna y la interfaz externa (IO).

El voltaje del núcleo del procesador esta fijado en 1.05 V. La DRAM interna es impulsada por un regulador externo de 1.5 V. Finalmente, la IO opera a 2.5 V pero puede aceptar hasta niveles de 3.3 V.

Por defecto el núcleo del procesador trabaja a 500 MHz.

3.1.1 Interfaz de expansión

La interfaz de expansión consiste de tres conectores. La siguiente tabla muestra las interfaces que cada conector provee.

Tabla. 3. 1. Conectores de interfaz de expansión

Connector	Interfaces
J1	5V, GND, dirección, dato
J2	2.5V, GND, SDRAM señales de control, banderas, IRQs, timers, datos
J3	GND, reset, DMA, control de memoria, CLKOUT, Señales puerto de enlace

3.1.2 Interfaz de audio

La interfaz de audio de la tarjeta EZ-KIT Lite permite interactuar con el convertidor análogo-digital (ADC) y el convertidor digital-análogo (DAC) de la tarjeta. La interfaz de audio consiste de dos principales ICs: AD1871 y AD1854.

El AD1871 es un ADC de audio estéreo elaborado para aplicaciones de audio digitales que requieren una conversión análogo-digital de alto rendimiento. El AD1871 está configurado para trabajar a una frecuencia de muestreo fija de 48 KHz.

El AD1854 es un DAC de audio, *single chip* estéreo de alto rendimiento que entrega 113 dB de rango dinámico y 112 dB de SNR a una tasa de muestreo de 48 KHz.

Debido a que el ADSP no tiene ningún puerto serial (SPORTs), un dispositivo lógico programable complejo (CPLD) Xilinx genera las señales de control de la interfaz de audio entre el procesador y el circuito de audio. Fijando en alto la bandera FLAG3 del Procesador A (DSPA) habilita la interfaz de audio dentro del CPLD. Una vez que la interfaz de audio ha sido habilitada, los datos de audio pueden ser transferidos hacia y desde el procesador generando un ciclo DMAR0. Los datos de audio interactúan con el procesador a través de los 24 bits más bajos del bus de datos (D23-0).

Un conector CPLD IO (P6) ha sido agregado para permitir a un usuario conectar el CPLD con el puerto externo del procesador ADSP-TS201S (DSP A).

3.2 DIP SWITCHES

La figura 3-2 muestra la ubicación y las posiciones por defecto de los DIP Switch SW1, SW2 y SW10.

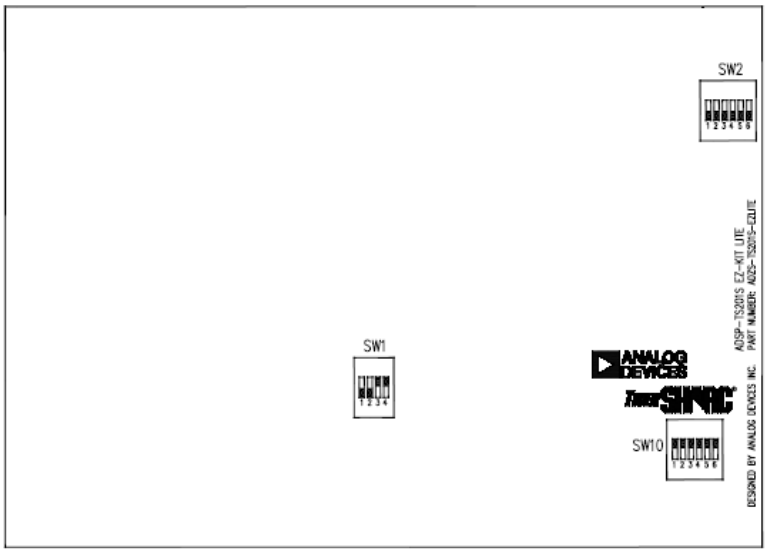

Figura. 3. 2. Ubicación de los *Switches* en la tarjeta ADSP

3.2.1 Selección de amplificación de audio (SW1)

El switch SW1 determina la amplificación de señales de derecha e izquierda conectadas a la LINE IN en el conector J9. La utilización de un micrófono puede ser habilitada simplemente variando la posición del *SW1* a los valores mostrados en la Tabla. 3.2.

Tabla. 3. 2. Selección de amplificación de audio (SW1)

Posición 1	Posición 2	Posición 3	Posición 4	Modo amplicación de audio
OFF	OFF	ON	ON	**No amplificación (por defecto)**
ON	ON	OFF	OFF	Para uso microfono electret

3.2.2 Selección del modo de arranque del procesador (SW2 Posición 1)

El *DIP switch* SW2 configura varios pines del procesador, el cual en sucesión, fija los modos de arranque del procesador después de encender o resetear.

No se debe cambiar las posiciones del SW1 mientras la tarjeta está encendida. Muchas de las posiciones de los pines pueden ser reconfiguradas vía *software* después de que el procesador es encendido.

La posición 1 del *switch* SW2 determina como el procesador arranca. La Tabla 3.3 muestra las configuraciones de los modos de arranque.

Tabla. 3. 3. Selección del modo de arranque del procesador (SW2 posición 1)

SW2 Posición 1	Modo de arranque
OFF	Arranque EPROM (por defecto)
ON	Arranque externo o arranque del puerto de enlace

3.2.3 Configuraciones del modo SYSCON/SDRCON (SW2 Posición 2)

La posición 2 del SW2 determina como el procesador controla la escritura en los registros SYSCON y SDRCON. La Tabla 3.4 muestra las configuraciones disponibles para cada tipo de escritura.

Tabla. 3. 4. Configuraciones del modo SYSCON/SDRCON (SW2 posición 2)

Posición 2	SYSCON/SDRCON Modo
OFF	SYSCON/SDRCON escribibles una vez (por defecto)
ON	SYSCON/SDRCON siempre escribibles

3.2.4 Configuración de habilitación de interrupciones (SW2 Posiciones 3 y 5)

Las posiciones 3 y 5 del *DIP switch* SW2 determinan como cada uno de los procesadores van a manejar las interrupciones. La Tabla 3.5 y Tabla 3.6 muestran las configuraciones de interrupción disponibles.

Tabla. 3. 5. Configuración de habilitación de bit de interrupción (SW2 posición 3)

SW2 Posición 3	Habilita el modo de interrupción para el procesador A (U11)
OFF	Deshabilita interrupción, modo de nivel-sensitivo (por defecto)
ON	Habilita interrupciones, modo de flanco-sensitivo

Tabla. 3. 6. Configuración de habilitación de bit de interrupción (SW2 posición 5)

SW2 Posición 5	Habilita el modo de interrupción para el procesador B (U12)
OFF	Deshabilita interrupción, modo de nivel-sensitivo (por defecto)
ON	Habilita interrupciones, modo de flanco-sensitivo

3.2.5 Configuraciones del ancho del puerto de enlace (SW2 Posiciones 4 y 6)

Las posiciones 4 y 6 del *switch* SW2 determinan el ancho de datos del puerto de enlace. La Tabla 3.7 y la Tabla 3.8 muestran las configuraciones disponibles para los dos tipos de anchos de puertos de enlace.

Tabla. 3. 7. Configuraciones del ancho del puerto de enlace (posición 4 de SW2)

SW2 Posición 4	Ancho de datos del puerto de enlace para el procesador A (U11)
OFF	Ancho de dato de 1 bit del puerto de enlace (por defecto)
ON	Ancho de datos de 4 bits para el puerto de enlace

Tabla. 3. 8. Configuraciones del ancho del puerto de enlace (posición 6 de SW2)

SW2 Posición 6	Ancho de datos del puerto de enlace para el procesador B (U12)
OFF	Ancho de dato de 1 bit del puerto de enlace (por defecto)
ON	Ancho de datos de 4 bits para el puerto de enlace

3.2.6 Configuraciones del *switch* FLAGs e IRQs

El *DIP switch* SW10 determina el origen de la bandera y las señales IRQ conectadas a cada una de los posibles procesadores. El origen puede ser modificado para manejar las redes a través de un pulsador o un origen externo a través de la cabecera de expansión. La tabla 3.9 muestra las configuraciones de origen de interrupción y bandera disponibles.

Tabla. 3. 9. Configuraciones del switch de FLAGs y IRQs (SW10)

DSP A		DSP B		DSP A	DSP B	Uso con
Posición 1 (FLAG0)	Posición 2 (FLAG1)	Posición 3 (FLAG0)	Posición 4 (FLAG1)	Posición 5 (IRQ0)	Posición 6 (IRQ0)	
OFF	OFF	OFF	OFF	OFF	OFF	Origen externo
ON[1]	ON	ON	ON	ON	ON	Switch pulsador de la tarjeta

1 Configuración por default

3.3 LEDS Y PULSADORES

La Figura 3.3 muestra las ubicaciones de los LEDs y los pulsadores.

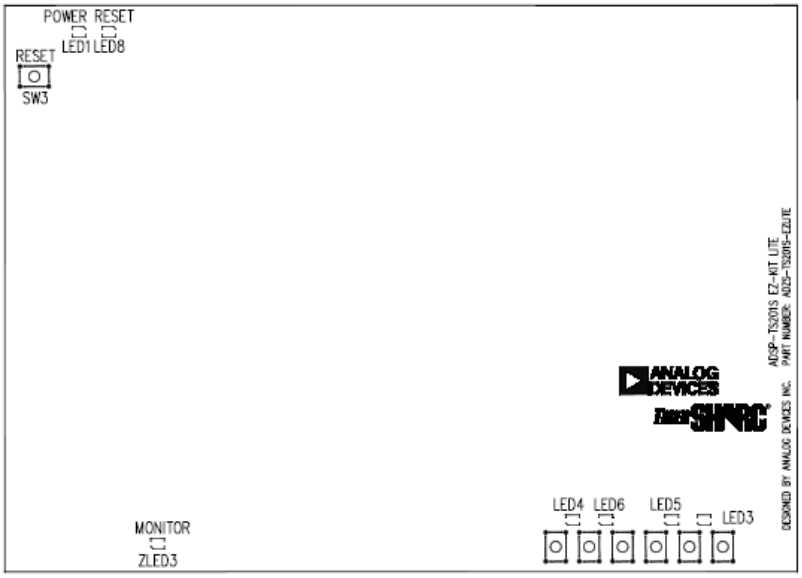

Figura. 3. 3. Ubicaciones de LEDs y pulsadores

3.3.1 LED de encendido (LED1)

El LED verde, LED1, indica que la energía esta siendo suministrada adecuadamente a la tarjeta.

3.3.2 LED de reset (LED8)

Cuando el LED8 esta encendido, indica que el reset maestro de todos los principales ICs esta activa.

3.3.3 LEDs de Bandera (LED3-6)

Los LEDs de bandera se conectan a los pines FLAG programables del procesador (FLAG2 y FLAG3). Los LEDs se activan en alto y se encienden con una salida de “1”

desde el procesador. La Tabla 3.10 muestra las señales FLAG y sus correspondientes LEDs.

Tabla. 3. 10. LEDs de banderas

Pin de Bandera	LED de referencia	Pin de Bandera	LED de referencia
FLAG2_A	LED4	FLAG2_B	LED5
FLAG3_A	LED6	FLAG3_B	LED3

3.3.4 LED monitor USB (ZLED3)

El LED monitor USB indica que la comunicación USB ha sido inicializada exitosamente, permitiendo conectarse al procesador usando Visual DSP++ 5.0. Si ZLED3 no se enciende, se debe tratar de resetear la tarjeta o reinstalar el controlador del USB.

3.3.5 Pulsadores de bandera programables SW6-9

Cuatro pulsadores son provistos para entradas de usuario de propósito general. Los pulsadores SW6-9 se conectan a los pines FLAG programables del procesador. Los pulsadores se activan en alto y cuando son presionados, envían un "1" al procesador. La Tabla 3.11 muestra las señales FLAG y sus correspondientes pulsadores.

Tabla. 3. 11. Pulsadores de Bandera

Pin de Bandera	Pulsador de referencia
FLAG0_A	SW9
FLAG1_A	SW8
FLAG0_B	SW6
FLAG1_B	SW7

3.3.6 Pulsadores de interrupción (SW4 y SW5)

Dos pulsadores SW4 y SW5, son provistos para interrupciones de usuario. Los pulsadores se conectan a los pines de interrupción del procesador. Los pulsadores se activan en bajo; y, cuando se presionan, envían un "0" al procesador. La Tabla 3.12 muestra las señales de interrupción y sus correspondientes pulsadores.

Tabla. 3. 12. Pulsadores de interrupción

Pin de interrupción	Pulsador de referencia
IRQ0_A	SW4
IRQ0_B	SW5

3.3.7 Pulsadores de Reset (SW3)

El pulsador RESET, SW3, resetea todos los ICs en la tarjeta, excepto la interfaz USB después de que esta ha sido configurada.

3.4 CONECTORES

La ubicación de los conectores se muestra en la Figura 3.4.

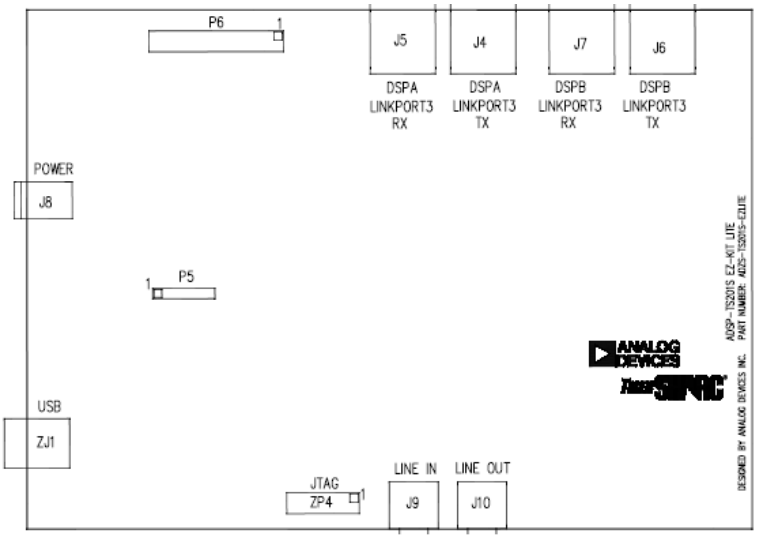

Figura. 3. 4. Ubicación de los conectores

Los conectores que se pueden encontrar dentro de la tarjeta son:

- Dos *jacks* de audio estéreo (J9 y J10).
- Energía (J8)
- JTAG (ZP4)
- USB (ZJ1)
- Tres interfaces de expansión (J1-3)
- Cuatro puertos de enlace (J4-7): Existen cuatro conectores RJ-45. Dos conectores son usados para el puerto de enlace 3 del procesador A, y dos son usados para el puerto de enlace 3 del procesador B.

3.5 INSTALACIÓN DE HARDWARE

3.5.1 Instalación de los controladores del cable USB

1. Energizar la tarjeta.
2. Conectar el puerto USB (ZJ1) de la tarjeta al puerto USB de la PC utilizando el cable proporcionado en el EZ-KIT Lite.
3. Aparecerá el siguiente cuadro en pantalla una vez conectado el cable.

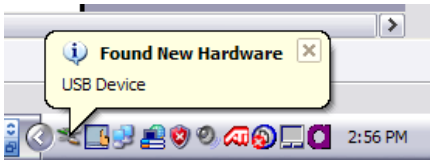

Figura. 3. 5. Reconocimiento del Cable USB del EZ-KIT Lite

4. Aparecerá además el cuadro de dialogo de asistente de instalación de *hardware*. Se debe seleccionar la opción "*No, not this time*" para poder instalar los controladores desde el CD de instalación. Presionar "*Next*"

Figura. 3. 6. Página de inicio del Asistente de instalación del controlador del cable USB

5. Seleccionar la opción "*Install the software automatically*". Presionar "*Next*".

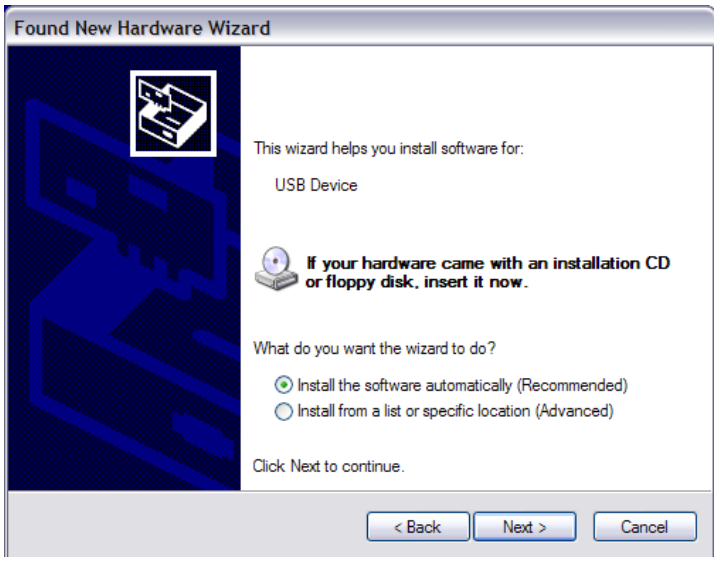

Figura. 3. 7. Ventana para instalación automática del controlador del cable USB

6. En el siguiente cuadro de dialogo procederá a buscar los controladores de instalación del dispositivo USB dentro del CD.

Figura. 3. 8. Proceso de búsqueda del controlador del cable USB

7. Una vez encontrado el controlador se espera el proceso de instalación.

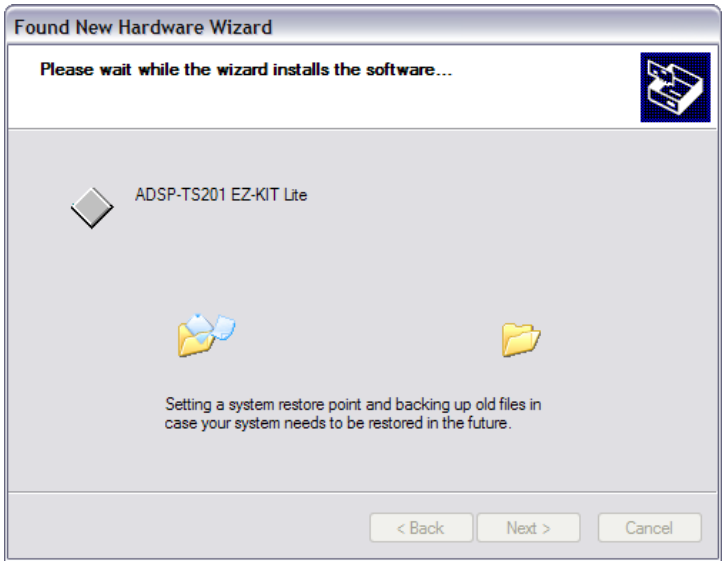

Figura. 3. 9. Proceso de instalación del controlador del cable USB

8. Se ha finalizado la instalación, presionar "*Finish*"

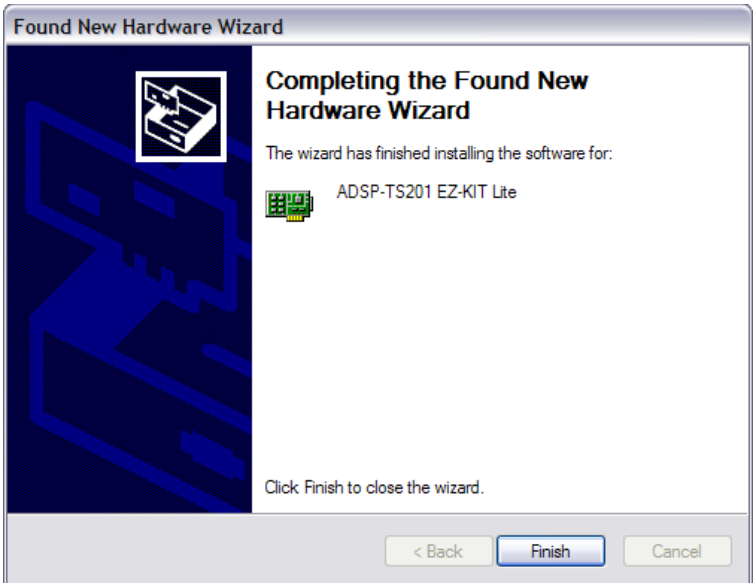

Figura. 3. 10. Ventana de instalación del controlador del cable USB completado

3.5.2 Instalación de los controladores de la tarjeta ADSP-TS201

1. Una vez terminada la instalación del dispositivo USB, aparecerá el siguiente cuadro automáticamente.

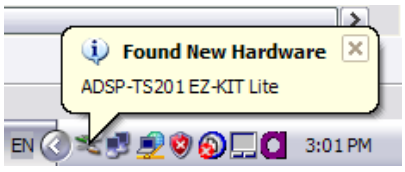

Figura. 3. 11. Reconocimiento de la tarjeta EZ-KIT Lite ADSP-TS201

2. Aparecerá además el cuadro de diálogo de asistente de instalación de *hardware*. Se debe seleccionar la opción "*No, not this time*" para poder instalar los controladores desde el CD de instalación. Presionar "*Next*"

Figura. 3. 12. Página de inicio del Asistente de instalación del controlador de la tarjeta

3. Seleccionar la opción "*Install the software automatically*". Presionar "*Next*".

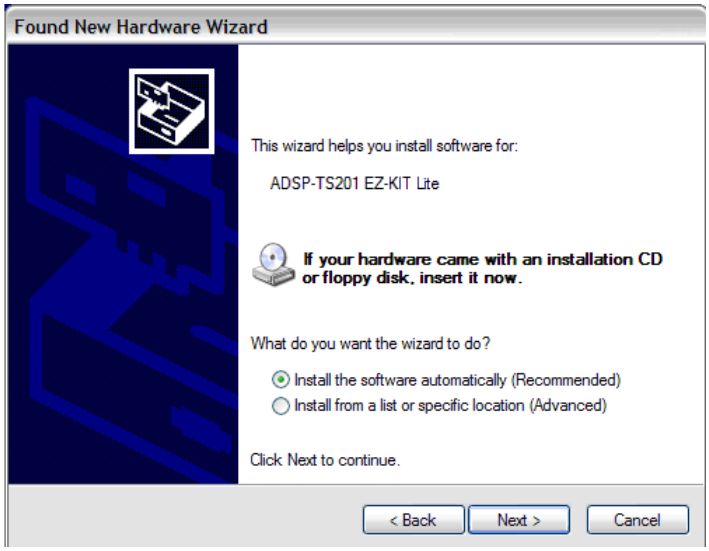

Figura. 3. 13. Ventana para instalación automática del controlador de la tarjeta

4. En el siguiente cuadro de diálogo procederá a buscar los controladores de instalación de la tarjeta ADSP-TS201 EZ-KIT Lite dentro del CD.

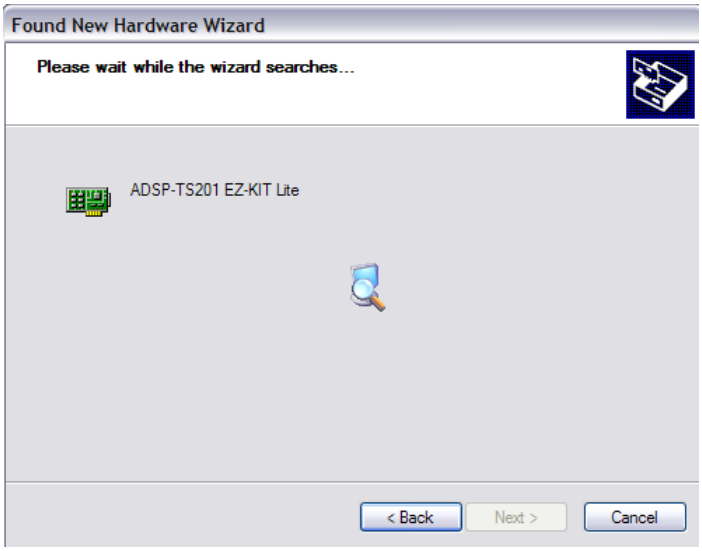

Figura. 3. 14. Proceso de búsqueda del controlador de la tarjeta.

5. Una vez finalizada la instalación de los controladores de la tarjeta presione "*Finish*".

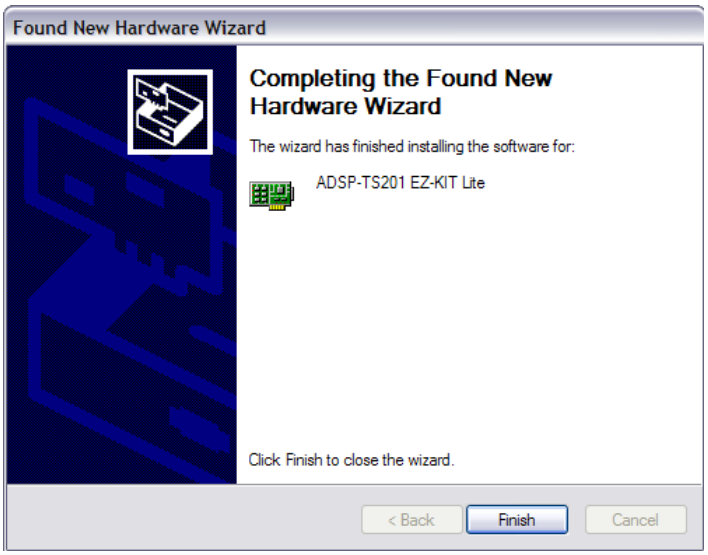

Figura. 3. 15. Ventana de instalación del controlador de la tarjeta completado

Una vez que sea a completado satisfactoriamente la instalación de los controladores, aparecerá un cuadro de diálogo como el siguiente en pantalla.

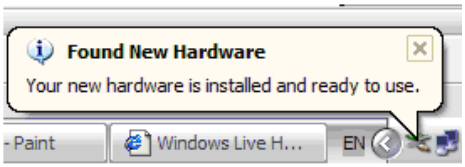

Figura. 3. 16. Cuadro de diálogo de controladores instalados correctamente

3.6 INSTALACIÓN DEL SOFTWARE

1. Ingresar el CD de instalación. El programa de instalación se ejecutará automáticamente.

2. Aparecerá el siguiente cuadro en pantalla. Presione "*Next*" para continuar con la instalación.

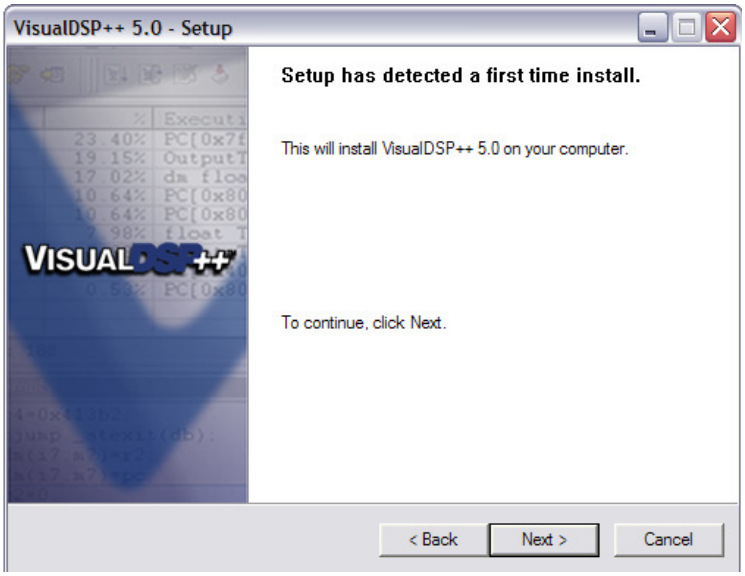

Figura. 3. 17. Ventana de inicio para instalación de software VisualDSP++ 5.0

3. Aceptar los términos en el acuerdo de licencia seleccionando la opción indicada en el siguiente gráfico. Presione "*Next*".

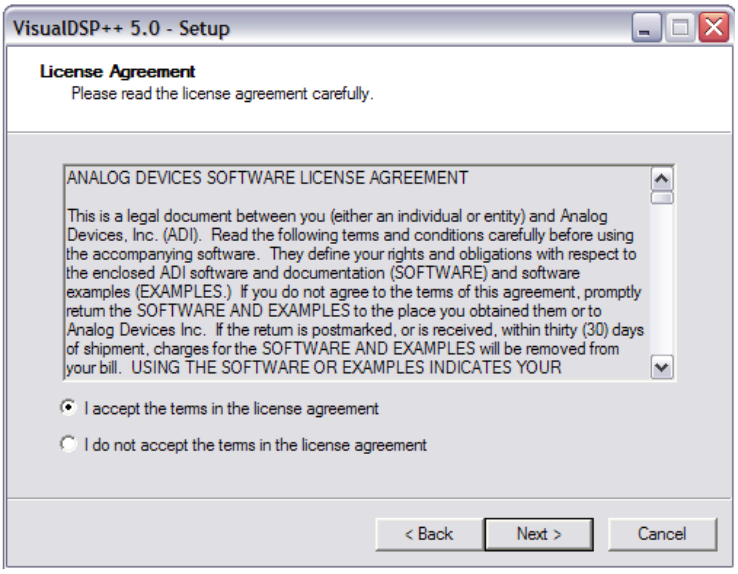

Figura. 3. 18. Acuerdo de licencia del Software

4. Ingresar el nombre del Usuario y el Nombre de la Compañía para la cual se destina el uso de la tarjeta. Presione "*Next*".

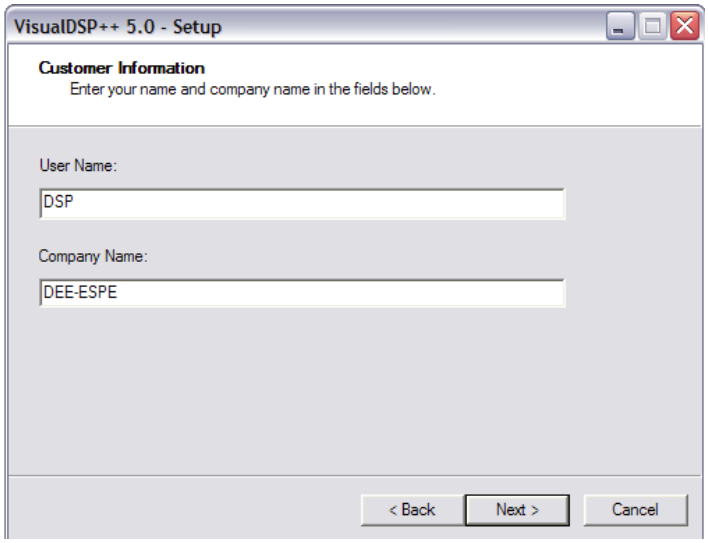

Figura. 3. 19. Ventana de información del consumidor

5. Seleccionar la ubicación en la que se guardará el *software* de la tarjeta. Presione "*Next*".

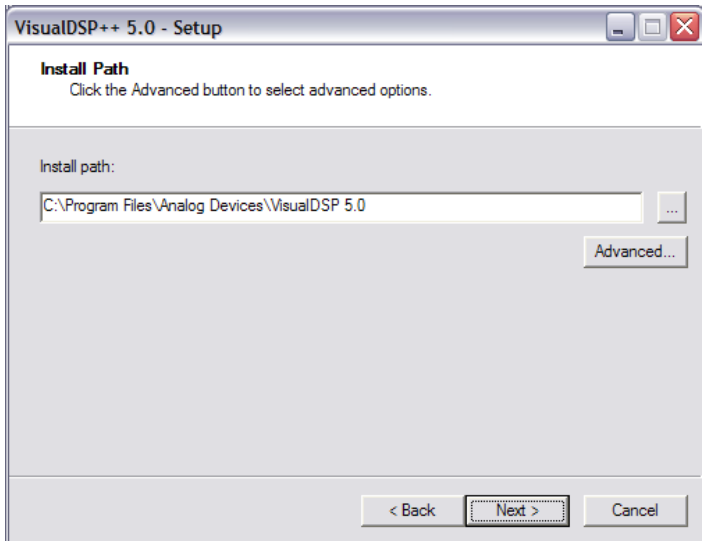

Figura. 3. 20. Ruta para instalación del *software*

6. Verificar si la información ingresada es correcta y presione "*Install*". Caso contrario presione "*Back*".

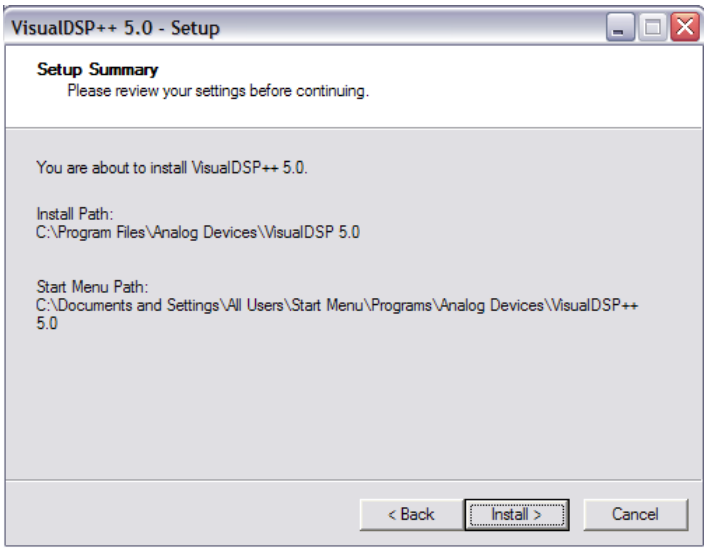

Figura. 3. 21. Ventana de resumen de la configuración del *software*

7. Esperar a que se culmine el proceso de instalación.

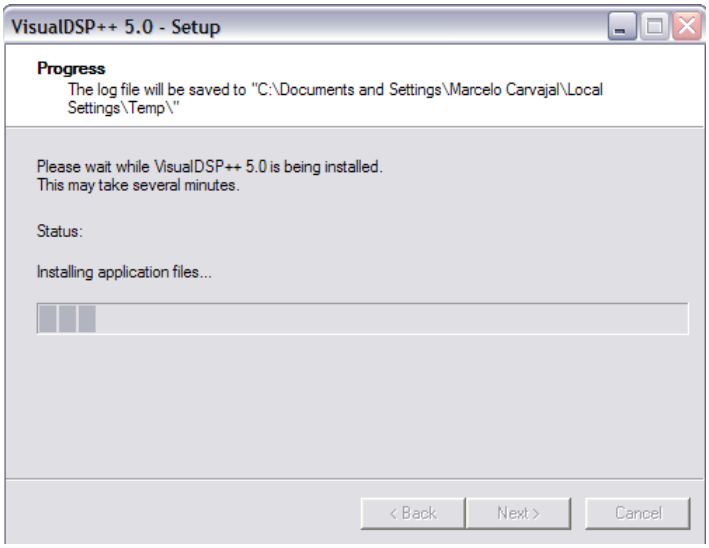

Figura. 3. 22. Proceso de instalación del *software*.

8. Se observa en el cuadro de diálogo el mensaje de que la instalación ha sido completada exitosamente. Presionar "*Next*".

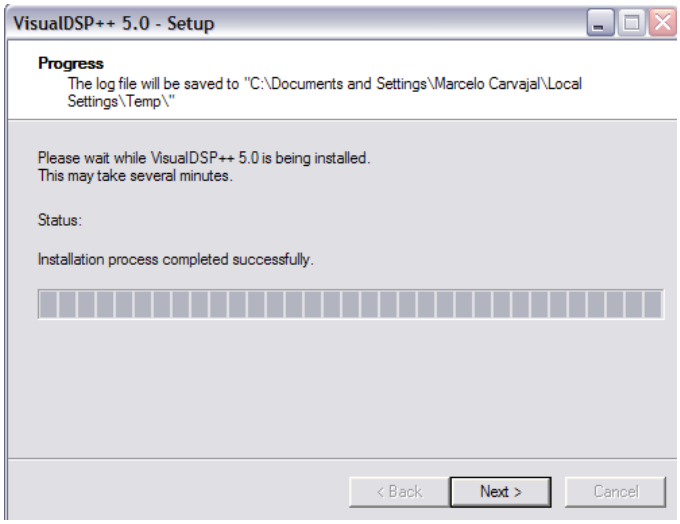

Figura. 3. 23. Instalación de *software* completada exitosamente

9. Finalmente aparecerá un cuadro de dialogo en el cual se puede realizar la actualización del software VisualDSP++ 5.0 y en el que además se indica que la instalación ha finalizado.

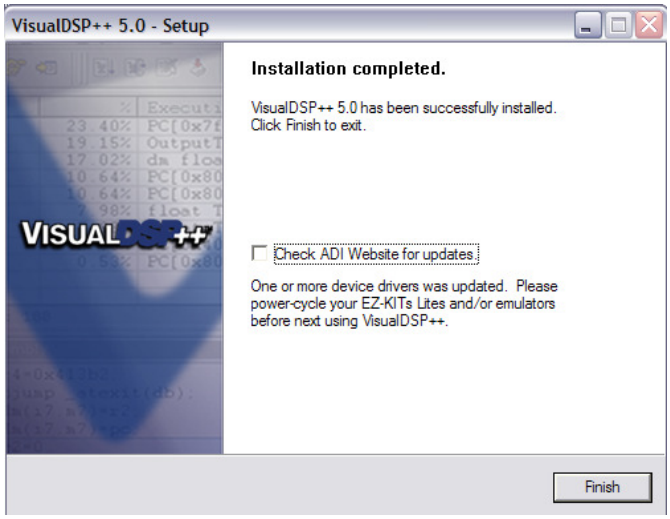

Figura. 3. 24. Ventana de instalación finalizada y para actualización de *software*

3.6.1 REGISTRO Y VALIDACIÓN DE LICENCIA DE LA TARJETA EZ-KIT LITE ADPS-TS201

1. Al intentar abrir el programa, un mensaje de información indica que ninguna licencia puede ser encontrada; se debe presionar "*Yes*" para poder instalar la licencia.

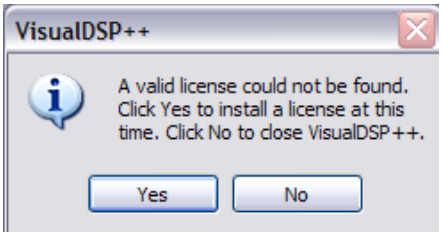

Figura. 3. 25. Mensaje para instalar licencia de software

2. En la pestaña de licencias del cuadro de diálogo *"About"*, presionar "*New*".

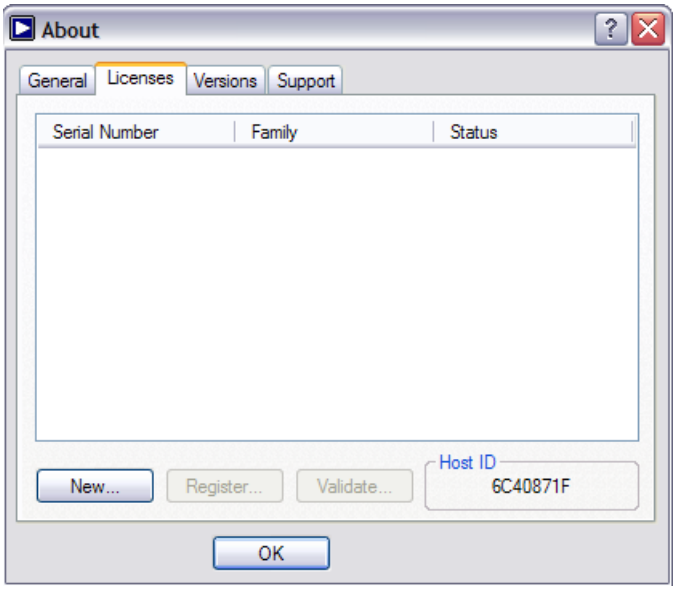

Figura. 3. 26. Ventana para agregar una nueva licencia

3. Seleccionar la opción "*Node-Locked license or Test Drive license*".

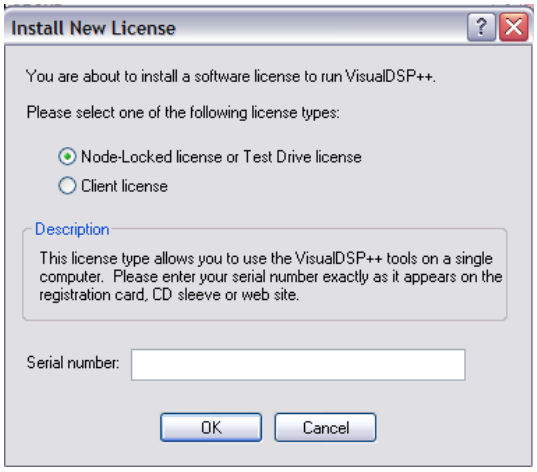

Figura. 3. 27. Ventana para seleccionar el tipo de licencia

4. Ingresar el número serial de la tarjeta exactamente como se muestra en la parte posterior del sobre que contiene el CD del *KIT*. Presione "*OK*".

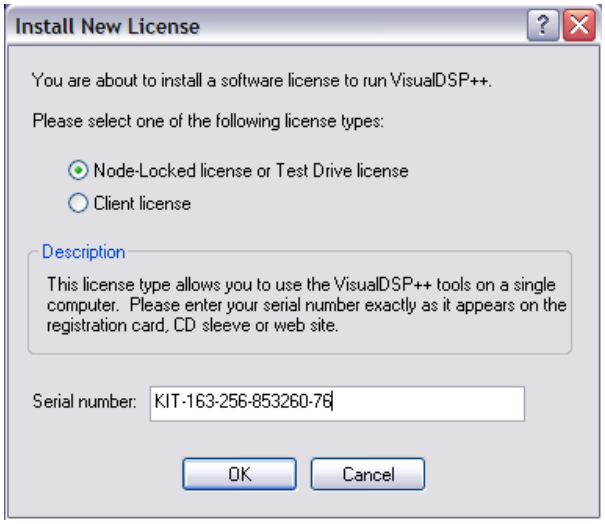

Figura. 3. 28. Ingreso del número serial de la tarjeta EZ-KIT Lite ADSP-TS201

5. Si el número serial fue ingresado correctamente aparecerá el siguiente mensaje.

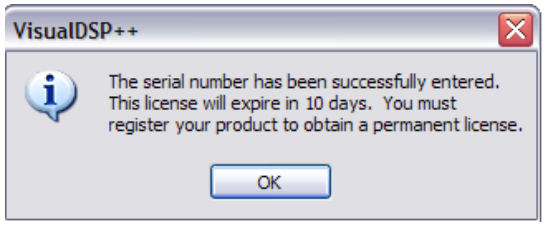

Figura. 3. 29. Mensaje de número serial ingresado correctamente

6. Para registrar la instalación, presione "*Register*" seleccionando previamente el número serial del *kit* correspondiente. Presione "*OK*".

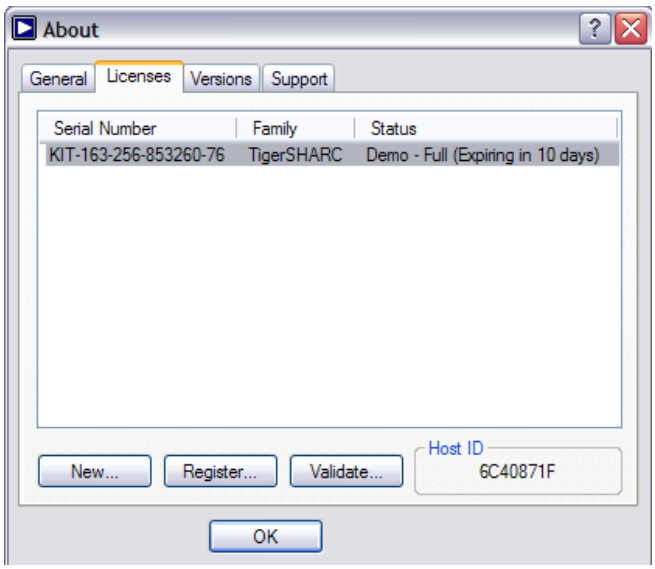

Figura. 3. 30. Ventana para acceder al registro de la licencia

7. Se abrirá la página Web de registro de licencia de *Analog Devices*. Se debe seguir las indicaciones mostradas en la página para completar el registro de licencia.

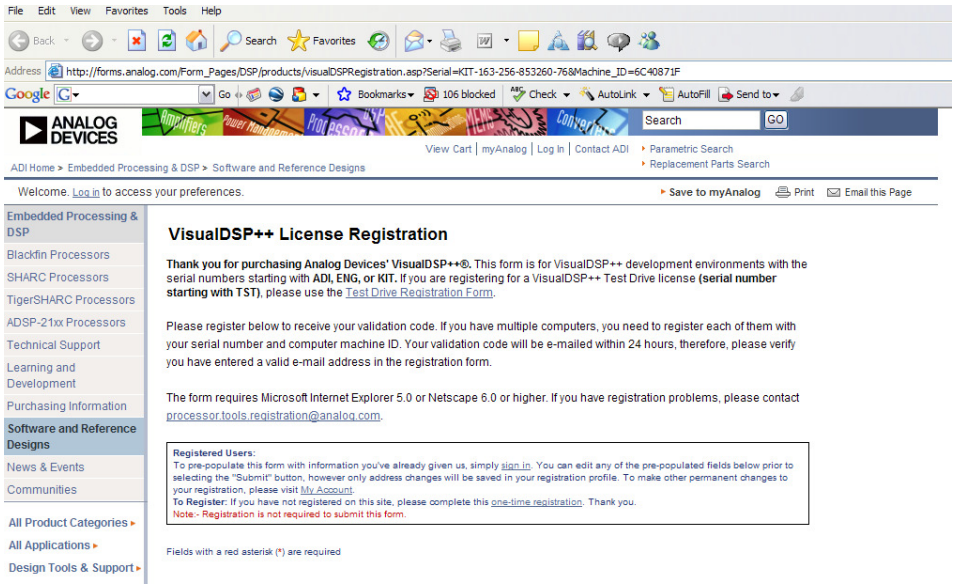

Figura. 3. 31. Página WEB para el registro de la licencia de la tarjeta

8. Ahora la instalación está registrada y validada para el período de prueba. Un código de validación será enviado al e-mail registrado en el punto anterior.

9. Para ingresar el código de validación del menú **Inicio**, seleccione **Programas → Analog Devices → Visual DSP++ 5.0→ Visual DSP++ Environment**. Una vez abierto el programa, seleccionar **Help → About Visual DSP++.**
10. Seleccionar la pestaña "*Licenses*" y escoger el número serial que se quiere validar. Presionar "*Validate*".

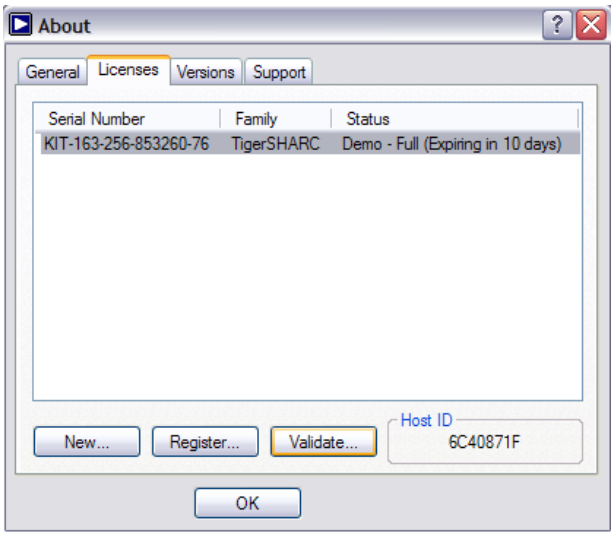

Figura. 3. 32. Ventana que indica el estado de la licencia (Demo-Full)

11. Ingresar el código de validación en el cuadro de diálogo. Presionar "*OK*" para completar la validación de la licencia.

Figura. 3. 33. Ventana para ingreso del código de validación

12. Finalmente se mostrará un mensaje de que el código de validación de la licencia es correcto.

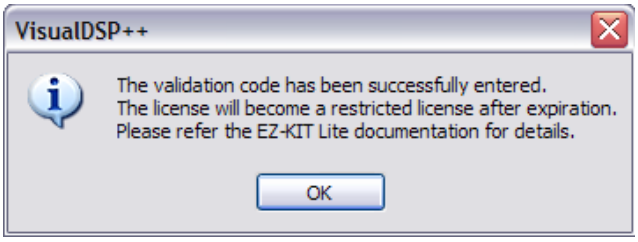

Figura. 3. 34. Mensaje de código de validación ingresado exitosamente

CAPITULO IV

DESCRIPCIÓN Y MANEJO DEL SOFTWARE DE DESARROLLO "VISUALDSP++ 5.0"

4.1 DESCRIPCIÓN DEL SOFTWARE VISUALDSP++ 5.0

4.1.1 Introducción a VisualDSP++ 5.0

- **Características de VisualDSP++ 5.0**

VisualDSP++ 5.0 provee las siguientes características:

- Capacidades de edición extensa.
- Manejo flexible de proyectos.
- Fácil acceso para herramientas de desarrollo de código.
- Opciones flexibles de construcción de proyectos.
- Soporte VisualDSP++ 5.0 Kernel (VDK)
- Manejo flexible del espacio de trabajo (*workspace*). Crea hasta 10 espacios de trabajo y rápidamente los intercambia entre ellos.
- Fácil movimiento entre actividades de construcción y depuración.

La Figura 4.1 muestra el Desarrollo Integrado y Ambiente de Depuración (IDDE).

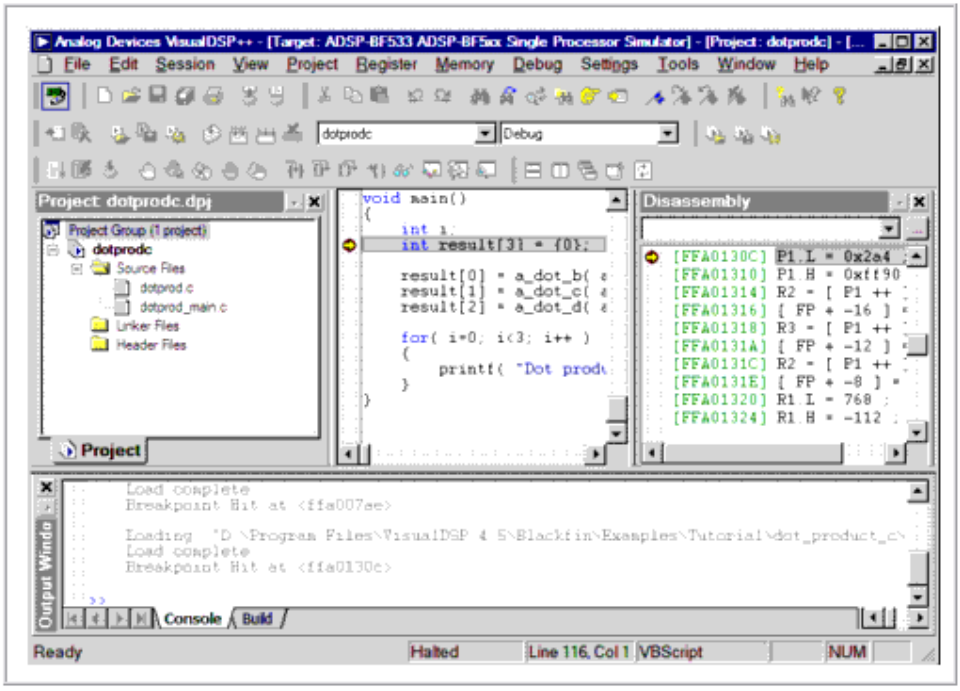

Figura. 4.1. IDDE de VisualDSP++ 5.0

VisualDSP++ 5.0 reduce el tiempo de depuración debido a que provee las siguientes características claves:

- Actividades de depuración fáciles de usar.
- Soporta múltiples lenguajes (C, C++ o Ensamblador).
- Control de depuración efectivo.
- Herramientas para mejorar rendimiento.

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

- **Herramientas de desarrollo de código**

Las herramientas de desarrollo de código incluyen:

- Compilador C/C++
- Librería en tiempo de corrida con alrededor de 100 rutinas matemáticas, DSP y librerías de tiempo de corridas C.
- Ensamblador
- Enlazador
- Cargador
- Simulador
- Emulador

Estas herramientas permiten desarrollar aplicaciones que aprovechan la arquitectura del procesador. El editor de enlace del VisualDSP++ 5.0 soporta multiprocesamiento, memoria compartida y sobrecarga de memoria.

- **Características de edición de archivo fuente**

VisualDSP++ 5.0 simplifica las tareas que envuelven a archivos de origen. Todas las actividades necesarias para crear, observar, imprimir, mover y localizar información que se enlistan a continuación son fáciles de realizar:

- Edición de archivos de texto
- Ventanas de edición
- Especifica color para la sintaxis
- Evaluación de expresión contexto-sensitivo
- Iconos de estado
- Muestra detalles de errores y código de infringimiento

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

- **Características de manejo del proyecto**

VisualDSP++ 5.0 provee un flexible manejo del proyecto para el desarrollo de aplicaciones, incluyendo el acceso a todas las actividades necesarias para crear, definir y construir proyectos. Entre dichas aplicaciones se encuentran:

- Definir y manejar proyectos.
- Acceso y manejo de las herramientas de desarrollo de código.
- Muestra y responde a resultados de la construcción del proyecto.
- Maneja archivos fuente.

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

4.1.2 Entorno VisualDSP++ 5.0

VisualDSP++ 5.0 es una interfaz de usuario intuitiva y fácil de usar para programación de procesadores TigerSHARC. En este capítulo se analiza el entorno de trabajo de VisualDSP++ 5.0, incluyendo la ventana principal y las ventanas de depuración.

De la ventana principal de aplicación, se puede abrir la ventana de proyecto, ventanas de edición, ventana de salidas y varias ventanas de depuración.

- **Ventana Project**

Para abrir una ventana *Project*, se escoge *View* y *Project Window*. La Figura 4.2 muestra una ventana de proyecto con VDK (VisualDSP++ 5.0 Kernel) habilitado.

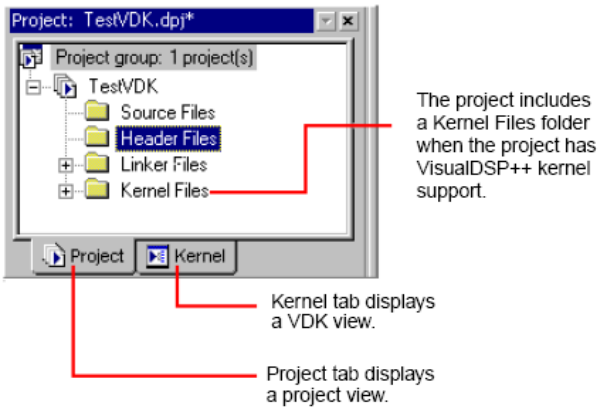

Figura. 4.2. Ejemplo de ventana de proyecto con etiqueta Kernel

La ventana del proyecto puede incluir dos sub pestañas:

- La pestaña Proyecto 

, la cual esta siempre disponible, despliega una representación jerárquica de una depuración de proyectos, carpetas, archivos y dependencias de sesión.
- La pestaña Kernel 

aparece cuando VDK esta habilitado para un proyecto. Esta despliega información relacionada a VDK.

▪ Pestaña de Proyecto

La pestaña de Proyecto despliega un grupo de proyecto, la cual puede contener cualquier número de proyectos. Solo un proyecto; sin embargo, esta activo a la vez. La figura 4.3 muestra un ejemplo de información que se despliega en Vista de Proyecto.

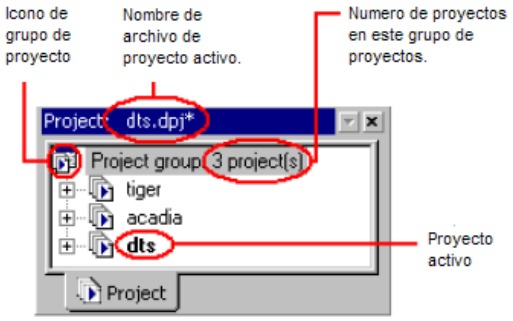

Figura. 4.3. Pestaña de proyecto

- **Nodos de proyecto**

La ventana de proyecto comprende los tipos de nodos descritos en la tabla 4.1.

Tabla. 4.1. Tipos de nodos en la ventana de proyectos

Nodo	Icono	Descripción
Grupo de proyecto		Solo un grupo de proyecto es permitido en una sesión de depuración.
Proyecto		Multiples proyectos permitidos, pero solo uno está activo.
Carpeta		Carpeta cerrada.
		Carpeta abierta revelando su contenido.
Archivo		Archivo que usa opciones de proyecto.
		Archivo cuyas opciones difieren de las opciones de proyecto.
		Archivo excluido de la configuración actual.
		Creador de archivo habilitado.
Dependencia de proyecto		Proyecto sobre el cuál otro proyecto depende.

- **Reglas de la ventana de proyecto**

Las siguientes reglas dictan como los archivos y subcarpetas se comportan en el árbol de archivos de ventana de proyecto.

- Se puede incluir cualquier archivo en un proyecto.
- Solo un archivo .ldf es permitido.
- No se puede añadir el mismo archivo dentro del mismo proyecto más de una vez.
- Solo un proyecto activo es permitido.

- Un archivo con una extensión no reconocida es ignorada al construir el proyecto.
- Cuando se añade un archivo a un proyecto, el archivo es ubicado en la primera carpeta configurada con la misma extensión. Si no hay carpetas presentes de la misma extensión, el archivo va a nivel del proyecto.

- **Asociaciones de archivo**

VisualDSP++ 5.0 asocia las extensiones de archivo como indica la Tabla 4.2.

Tabla. 4. 2. Extensiones de archivo

Herramientas	Extensión de archivos
Compilador	.c, .cpp, and .cxx
Ensamblador	.asm, .s, and .dsp
Enlazador	.ldf, .dlb, and .doj

- **Ventanas de edición**

Se usa las ventanas de edición para observar y editar archivos. Se abre tantas ventanas de edición como se quiera desde la ventana de proyectos haciendo doble clic sobre un archivo o escogiendo *Open File* del menú que se abre al presionar clic derecho. La Figura 4.4 muestra los ítems que pueden estar contenidos en ventanas de edición.

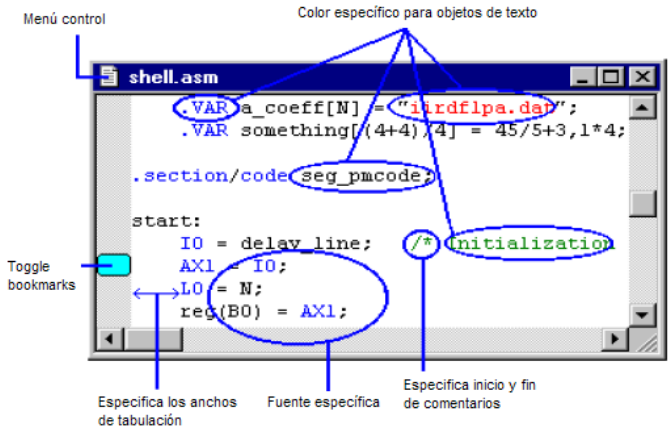

Figura. 4.4. Ítems en las ventanas de edición

Las ventanas de edición soportan:

- Comentarios, palabras claves y configuraciones de tabulación con un color codificado definido por el usuario.
- Dos modos de vista: modo origen y modo mixto.
- Buscar/reemplazar con búsqueda general y ajustando con expresiones generales.
- Saltar al siguiente o previo error de sintaxis.
- Copiar, cortar, pegar, deshacer y rehacer funciones.

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

- **Ventana de salida**

La ventana de salida despliega:

- Mensajes de texto de entrada y salida estándar.
- Información del estado de construcción para la construcción del proyecto actual.
- Mensajes de herramientas de desarrollo de código y provee acceso a errores en archivos de origen.

La ventana de salida sirve además como una interfaz de escritura. La Figura 4.5 muestra la información del estado de construcción.

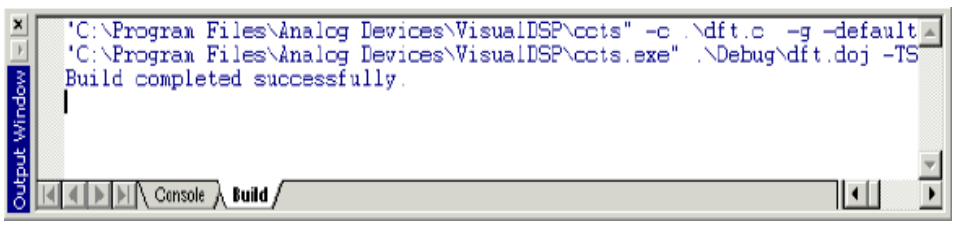

Figura. 4.5. Información del estado de construcción en la ventana de salida

- **Página de construcción y página de consola**

Las dos pestañas de la ventana de salida, *Console* y *Build* proveen diferente información y capacidades.

- **Página de construcción (Pestaña *Build*)**

La página de construcción despliega los mensajes de error generados durante una construcción. Al escoger *Next Error* o *Prev Error* del menú de *Edit* permite desplazarse entre los mensajes de errores.

- **Página de consola (Pestaña *Console*)**

La página de consola de la ventana de salida permite:

- Observar los mensajes de error de VisualDSP++ 5.0 o del estado de la sesión.
- Observar la salida STDIO de los programas C/C++.
- Observar los mensajes de I/O (cadenas).
- Realizar selección múltiple, copiar, pegar y borrar.
- Auto-completar comandos.
- Ejecutar un comando realizado previamente haciendo doble clic sobre el comando.
- Ingresar comandos multilínea añadiendo un carácter "*backslash* \" al final de la sentencia.

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

- **Ventanas de memoria**

Se abre las ventanas de memoria desde el menú *Memory*. Use las ventanas de memoria para:

- Ver y editar los contenidos de memoria.
- Desplegar la dirección de un valor.
- Bloquea el número de columnas desplegadas actualmente.
- Seguir la pista de una expresión.

- **Ventanas de Registro**

Se accede a las ventanas de varios registros a través del menú *Register*. La Figura 4.6 muestra un ejemplo de menú de registro para un procesador TigerSHARC.

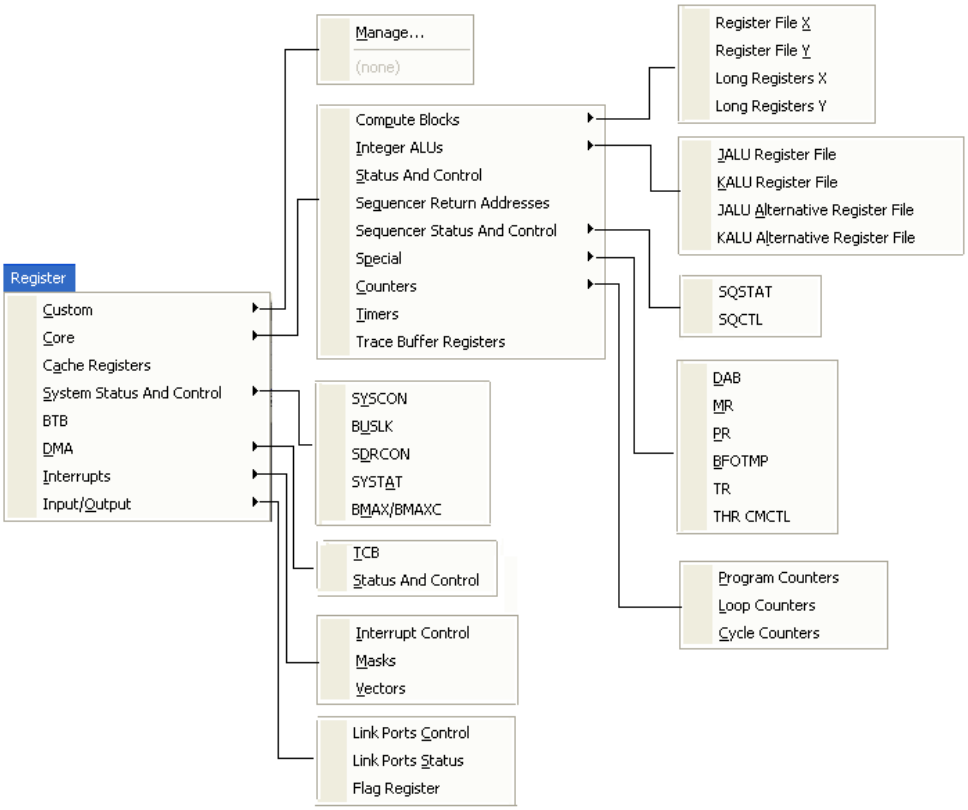

Figura. 4. 6. Menú de registro para un procesador TigerSHARC

Una ventana de registro permite:

- Observar y cambiar el contenido de registros.
- Cambiar la presentación de la ventana (formato del número)

- **Ventanas *Plot***

Una ventana *Plot* despliega un *plot* de memoria, el cual es una visualización de los valores obtenidos de la memoria de procesador. Se puede desplegar una o mas ventanas escogiendo *View*, *Debug Windows*, *Plot* y *New*.

En el cuadro de diálogo *Plot Configuration*, se especifica los contenidos de un *plot*. En el cuadro de diálogo *Plot Settings*, se especifica la presentación del *plot*. La Figura 4.7 muestra un ejemplo de una ventana *Plot*.

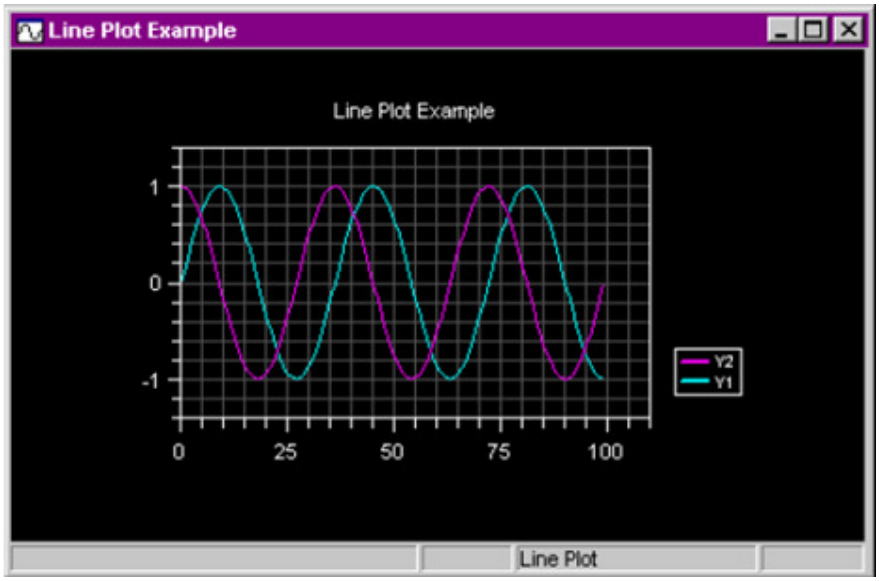

Figura. 4.7. Ventana *Plot*

4.1.3 Herramientas de depuración

- **Conectando a una sesión de depuración**

Cuando se abre el VisualDSP++ 5.0 por primera vez, este no se conecta a ninguna sesión. Para establecer una sesión en la ventana principal se escoge la pestaña *SESSION* y a continuación *NEW SESSION* (Figura 4.8).

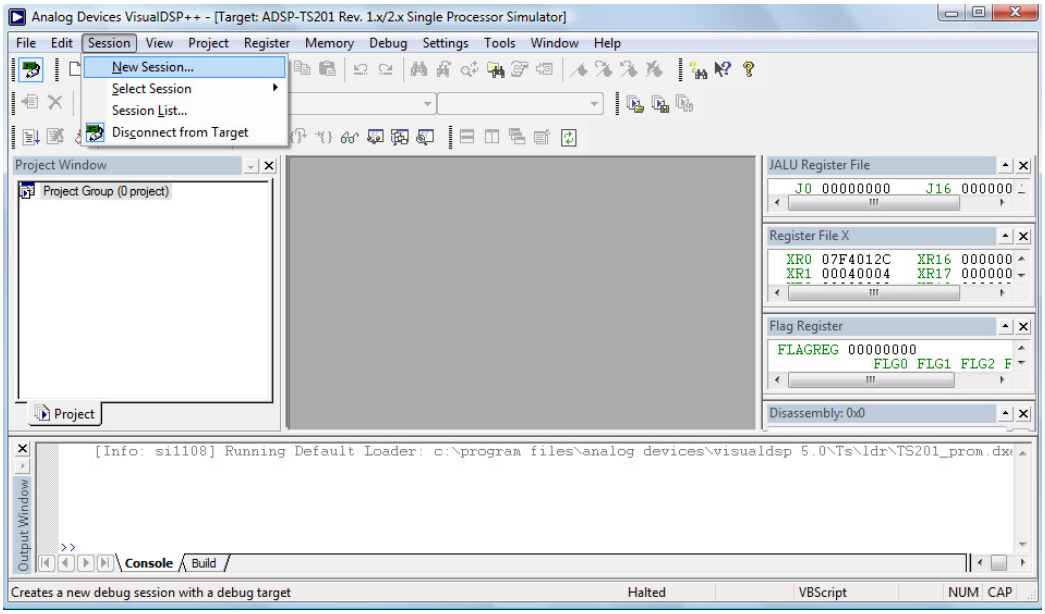

Figura. 4.8. Ventana principal de VisualDSP++ 5.0 y pestaña de Sesión

Además, VisualDSP++ 5.0 puede conectarse a un número de diferentes sesiones de depuración, que hayan sido previamente creadas, donde cada sesión tiene su propia aplicación y beneficios.

Los pasos para seleccionar el tipo de sesión son:

a) Selección del Procesador

Al hacer clic en *NEW SESSION* aparece la siguiente pantalla en la cual se debe escoger el tipo de procesador que se va a utilizar ver Figura 4.9, para nuestro caso el procesador TigerSHARC ADSP-TS201.

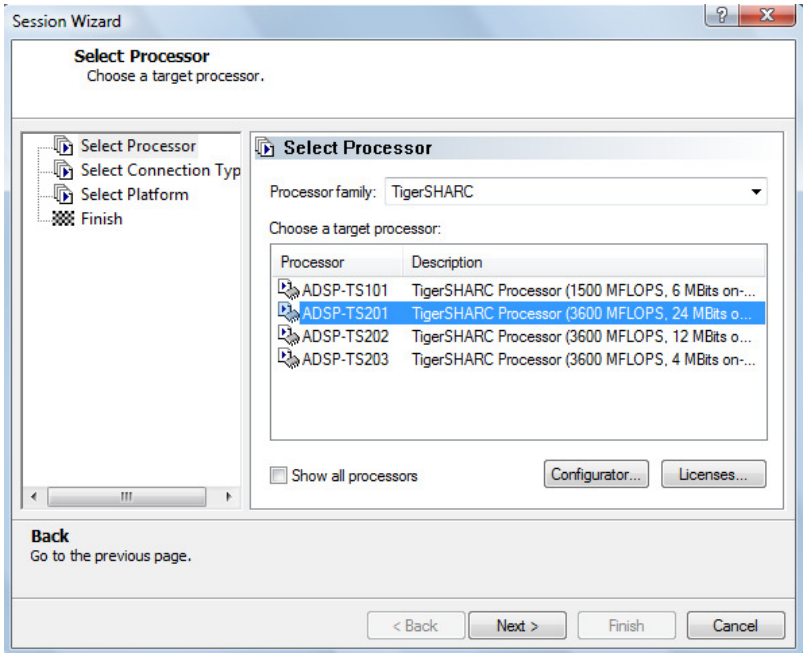

Figura. 4.9. Ventana de selección de procesador

b) Tipo de Conectividad

c)

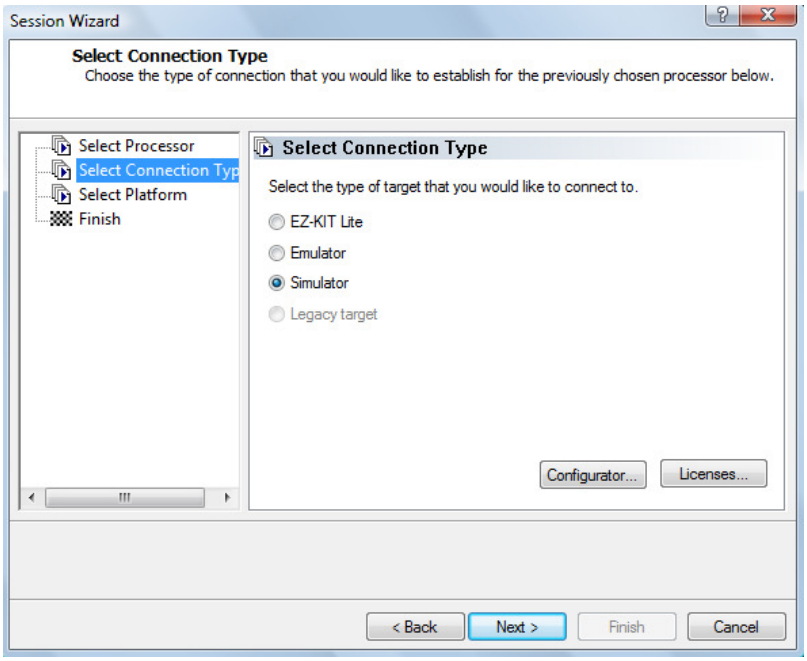

Figura. 4.10. Ventana de selección de tipo de conectividad

Los tipos de conectividad disponibles en VisualDSP++ 5.0 son:

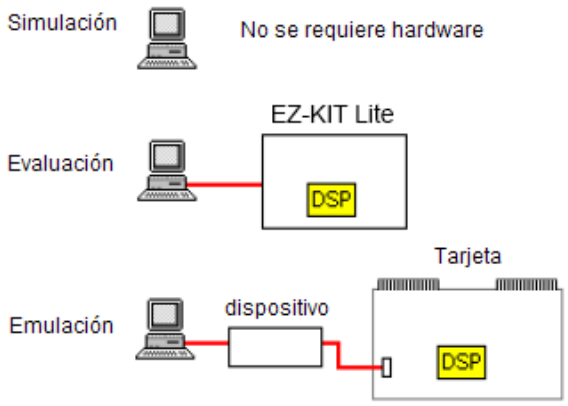

Figura. 4.11. Tipos de conectividad

- **Simulador**

Este es un modelo de *software* del procesador. Los simuladores ofrecen ventajas únicas; la primera es que ningún *hardware* externo es requerido. Además, los simuladores ofrecen una intuición única para los trabajos internos del procesador lo cual no es posible con sesiones basadas en *hardware*. El inconveniente es que un simulador es de varias órdenes de magnitud más lento que el mismo *hardware*. El modelo de *software* simula solo el procesador, haciendo difícil simular con precisión un sistema complejo que envuelve más de un procesador.

- **EZ-KIT Lite (Evaluador)**

Esta destinado para conexión USB entre la PC y la tarjeta *EZ-KIT Lite*. Una conexión *EZ-KIT* es fácil de manejar y es parte de la *EZ-KIT Lite*. Una vez que la tarjeta de *hardware* esta disponible para desarrollo, se usa una sesión emulador para conectarse al *hardware*.

- **Emulador**

Este es un emulador JTAG, el dispositivo ideal para conectarse a *hardware*, dando el mejor rendimiento y máxima flexibilidad. Es un módulo separado de la PC y el

EZ-KIT Lite que provee una conexión con gran ancho de banda entre la PC y un dispositivo siendo depurado. Un emulador requiere ser conectado a cualquier *hardware* diferente al *EZ-KIT Lite*.

Usar el configurador VisualDSP++ 5.0 (Figura 4.12) para alinear la sesión de *hardware* externo.

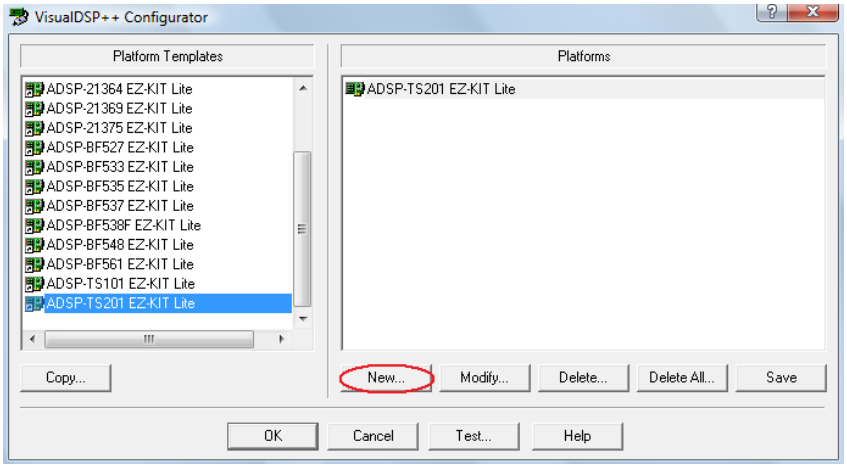

Figura. 4.12. Configurador de VisualDSP++ 5.0

d) Plataformas

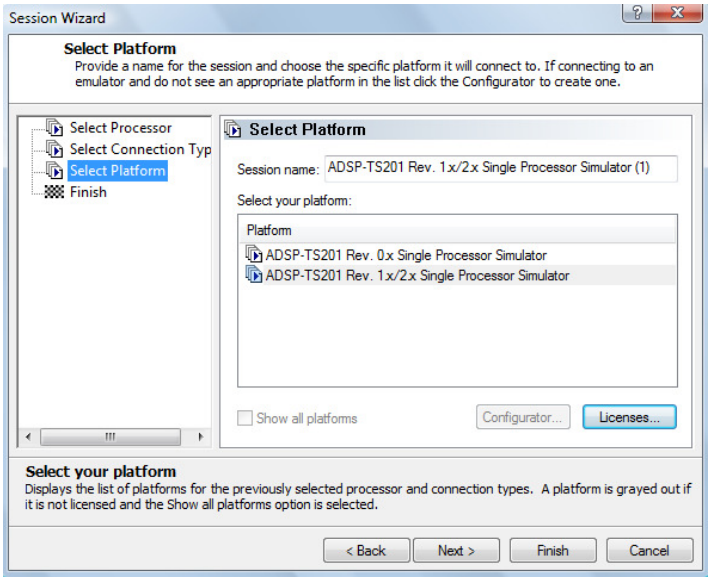

Figura. 4.13. Ventana de selección de plataforma

Una plataforma se refiere a la configuración del procesador con la cual una sesión se comunica. Varias plataformas pueden existir para una sesión de depuración dada. Por ejemplo, si tres emuladores son instalados en el sistema, se debe seleccionar *emulator 2* como la plataforma que se quiere usar. La plataforma que se use depende de la etapa de desarrollo del proyecto.

- **Seleccionar una nueva sesión de depuración**

Previamente VisualDSP++ 5.0 debe ser cerrado. A continuación, se usa el siguiente procedimiento para forzar una nueva sesión.

1. Mantener presionada la tecla CTRL del teclado. No soltarla hasta que la sesión *Wizard* aparezca.
2. Abrir VisualDSP++ 5.0 como normalmente se lo hace. La sesión *Wizard* aparecerá.
3. Especificar y activar una sesión de depuración.

- **Cargando el programa ejecutable**

Una vez que se ha especificado la sesión de depuración, se inicia la sesión cargando el programa ejecutable.

Después de una exitosa construcción del programa ejecutable, VisualDSP++ 5.0 (si esta configurado) carga el programa automáticamente a la sesión actual cuando el tipo de procesador de sesión corresponde al procesador del proyecto. Si la sesión actual no coincide con el tipo de procesador de proyecto, se está obligado a escoger otra sesión.

Si la carga automática no está configurada, VisualDSP++ 5.0 no intenta cargar el programa ejecutable automáticamente después de una exitosa construcción.

Esta opción de depuración ahorra tiempo debido a que no se tiene que cargar el programa ejecutable manualmente, y se puede iniciar al depurar inmediatamente después de construir el proyecto exitosamente.

- **Comandos de ejecución de programa**

Se puede ejecutar los comandos de ejecución del programa desde el menú de depuración o desde la barra de herramienta como se muestra en la Figura 4.14.

Los archivos ejecutables se ejecutan hasta que un evento tal como un punto de corte, punto de observación o un comando *Halt* por petición de usuario detenga la ejecución. Cuando la ejecución del programa se detiene, todas las ventanas se actualizan a direcciones y valores actuales.

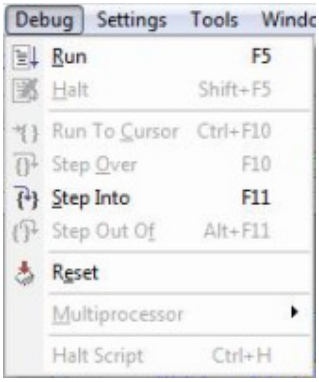

Figura. 4.14. Menú *Debug* (Depuración)

Usar los comandos mostrados en la Figura 4.14 para controlar la ejecución del programa, los comandos se describen en la Tabla 4.3.

Tabla. 4.3. Uso de comandos para la ejecución del programa

Comando	Descripción
Run	Corre un programa ejecutable. El programa corre hasta que un evento lo detenga, tal como un punto de corte o intervención del usuario. Cuando la ejecución del programa se detiene, todas las ventanas se actualizan para la actual dirección y valor.
Halt	Detiene la ejecución del programa. Todas las ventanas se actualizan después que el procesador se detiene. Los valores que cambiaron se destacan, y el estado de la barra despliega la dirección donde el programa se detiene.
Run to Cursor	Corre el programa a la línea donde está el cursor. Se puede colocar el cursor en la ventana de edición y ventana Disassembly.
Step Over	(Código C/C++ solo en la ventana de edición) Pasos simples hacia adelante a través de las instrucciones de programa. Si la línea de origen llama a una función, la función es ejecutada completamente, sin efectuar paso a paso las instrucciones de programa.
Step Into	(Ventana de edición o Ventana Diassembly) Pasos simples a través del programa una instrucción C/C++ o assembly a la vez. Las funciones encontradas son ingresadas.
Step Out Of	(Código C/C++ solo en la ventana de edición) Realiza pasos múltiples hasta que la función actual retorna a su llamado, y se detiene en la instrucción que sigue inmediatamente el llamado de la función.

Para mayor información, hacer referencia al documento *50_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

4.2 DESARROLLO DE PROYECTOS EN VISUALDSP++ 5.0

4.2.1 Etapas de desarrollo de un proyecto

- **Pasos para el desarrollo de un programa**

En el VisualDSP++ 5.0 *Environment*, el desarrollo de programas consiste de los siguientes pasos:

1. Crear un proyecto.
2. Configurar las opciones del proyecto.
3. Añadir y editar archivos de origen de proyecto.
4. Especificar las opciones de construcción del proyecto.

5. Construir una versión de depuración (Archivo ejecutable) del proyecto.
6. Crear una sesión de depuración y cargar el ejecutable.
7. Ejecutar y depurar el programa.
8. Construir una versión publica del proyecto.

Siguiendo estos pasos, se puede construir proyectos consistentes y exactos con mínima administración del proyecto. Este proceso reduce el tiempo de desarrollo para tener más concentración en desarrollo del código.

Paso I: Creación de un proyecto

Todo desarrollo en VisualDSP++ 5.0 ocurre dentro de un proyecto. El archivo de proyecto (.dpj) almacena información de la constitución del programa: lista de archivos de origen y propiedades de las opciones de herramientas de desarrollo.

VisualDSP++ 5.0 incluye la herramienta *Project Wizard* que simplifica la creación de un nuevo proyecto.

Paso II: Configurar opciones de Proyecto

Definir el procesador a ser utilizado y establecer las opciones de proyecto (o aceptar propiedades por defecto). El cuadro de diálogo *Project Options* provee acceso para las opciones del proyecto, lo cual permite la correspondiente construcción de las herramientas para procesar los archivos del proyecto.

Paso III: Añadir y editar archivos fuente en un proyecto

Un proyecto normalmente contiene uno o más archivos fuente C, C++, o ensamblador. Después de crear un proyecto y definir el procesador, añada los archivos nuevos o existentes al proyecto. Usar el editor de VisualDSP++ 5.0 para crear nuevos archivos o editarlos solo si existen archivos de texto.

a) Añadir Archivos al Proyecto

Se puede añadir cualquier tipo de archivo al proyecto. Las herramientas de desarrollo selectivamente procesan solo tipos de archivo reconocidos al construir el proyecto

b) Crear archivos para añadir al proyecto

Se puede crear nuevos archivos tipo texto. El editor puede leer o escribir archivos de texto con nombres arbitrarios. Al añadir archivos al proyecto el árbol de archivos del proyecto se actualiza en la ventana Project.

c) Editar archivos

Se puede editar los archivos que se incluyan dentro del proyecto. Para abrir un archivo para su edición, se hace doble clic sobre el icono del archivo en la ventana *Project*. El editor tiene una interfaz estándar estilo *Windows* y soporta múltiples ventanas abiertas. Además, el usuario puede configurar el color de la sintaxis y lenguaje específico del procesador.

Después de crear un proyecto, escoger el procesador y añadir o editar los archivos fuente del proyecto, se procede a configurar las opciones de construcción del proyecto. Se debe especificar las opciones o aceptar las opciones por defecto en VisualDSP++ 5.0 antes de usar las herramientas de desarrollo que crean el archivo ejecutable. El usuario puede especificar las opciones para todo un proyecto o para archivos individuales.

d) Configuración

La configuración de un proyecto controla su construcción. Por defecto, las opciones son *Debug* o *Release*.

Al seleccionar *Debug* y dejar el resto de opciones con sus configuraciones por defecto, construye un proyecto que puede ser depurado. El compilador genera información de depuración.

Al seleccionar *Release* y dejar el resto de opciones con sus configuraciones por defecto, construye un proyecto con limitadas o sin capacidades de depuración.

Paso IV: Construcción de una versión de depuración del proyecto

Después, construya una versión de depuración. Los mensajes de estado de cada herramienta de desarrollo de código aparecen en la ventana de salida como vaya progresando la construcción.

Paso V: Crear una sesión de depuración y cargar el ejecutable

Después de construir exitosamente un archivo ejecutable, establecer una sesión de depuración. Ejecutar proyectos que fueron desarrollados como sesiones de *hardware* o *software*. Después de especificar el procesador, tipo de conexión y la plataforma, cargar el archivo ejecutable del proyecto. Del icono *General* del cuadro de diálogo *Preferences*, se puede configurar a VisualDSP++ 5.0 para cargar el archivo automáticamente y avanzar a la función principal del código.

Paso VI: Ejecución y depuración el programa

Después de crear exitosamente una sesión de depuración y construir y cargar el programa ejecutable, se procede a ejecutar y depurar el programa. Si el proyecto no es actual (se ha desactualizado archivos fuente o información de dependencia), VisualDSP++ 5.0 muestra un mensaje para construir el proyecto antes de cargar y depurar el archivo ejecutable.

4.2.2 Creación, Depuración y Ejecución de un Programa en C

Para este punto, previamente se procede a crear un proyecto en lenguaje C con el nombre *tutorial_c.dpj* en el directorio *Program Files/Analog Devices/VisualDSP 5.0/TS/Examples*.

Los archivos fuente del proyecto fueron creados de tal manera que puedan cumplir con los pasos que se detallan en este punto del capítulo.

Paso I: Inicio de VisualDSP++ 5.0 y abrir un proyecto

Para iniciar VisualDSP++ 5.0 y para abrir un proyecto se procede de la siguiente forma:

1. Clic en el botón *Inicio* y seleccionar *Programas, Analog Devices, VisualDSP++ 5.0*, y *VisualDSP++ Environment.*

Cuando se necesita conectar a una sesión *Debug*, se da clic en el botón *Connect to target* ( ) de la barra de herramientas o se elige en el menú *Session* la opción *Select Session* Para crear una nueva sesión *Debug*, se selecciona *New* en el menú *Session* (Figura 4.15).

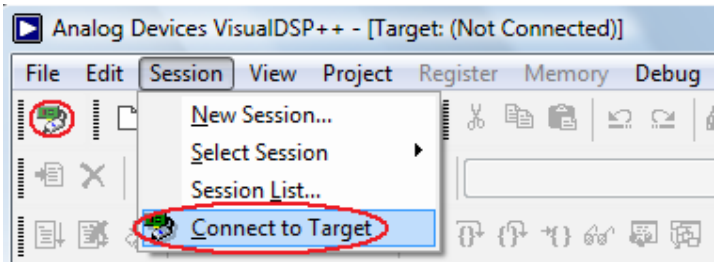

Figura. 4.15. Conexión a la tarjeta

2. En el menú *File* seleccionar *Open* y luego *Project*. VisualDSP++ 5.0 muestra la ventana de diálogo *Open* → *Project* como se muestra en la figura 4.16.

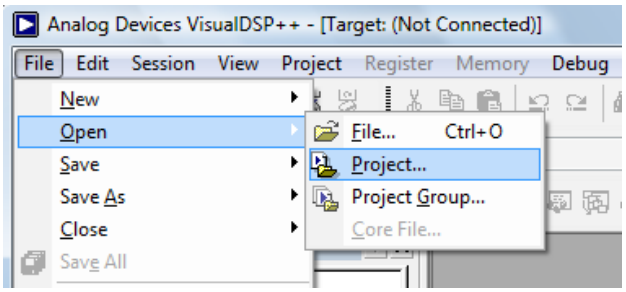

Figura. 4.16. Pasos para abrir un proyecto

3. En la ventana seguir la siguiente ruta: *Program Files/Analog Devices/VisualDSP 5.0/TS/Examples/No hardware required*.

4. Abrir el proyecto *tutorial_c.dpj*.

VisualDSP++ 5.0 carga el proyecto en la ventana *Project*, como se muestra en la Figura 4.17. En el ambiente muestra mensajes en la ventana *Output* de las características y archivos dependientes del proyecto.

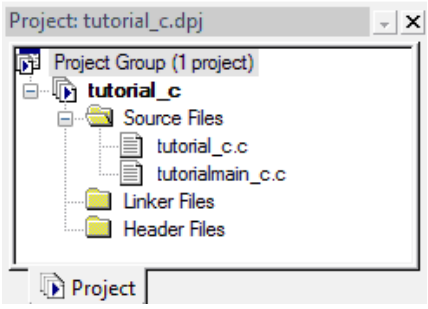

Figura. 4.17. Proyecto cargado en la ventana de Proyectos

El proyecto *tutorial_c.dpj* comprende dos archivos fuente de lenguaje C, *tutorial_c.c* y *tutorialmain_c.c*, los cuales definen los arreglos y cálculos del proyecto respectivamente.

5. En el menú *Settings*, escoger *Preferences* para abrir el cuadro de diálogo de *Preferences*, como se muestra en la Figura 4.18.

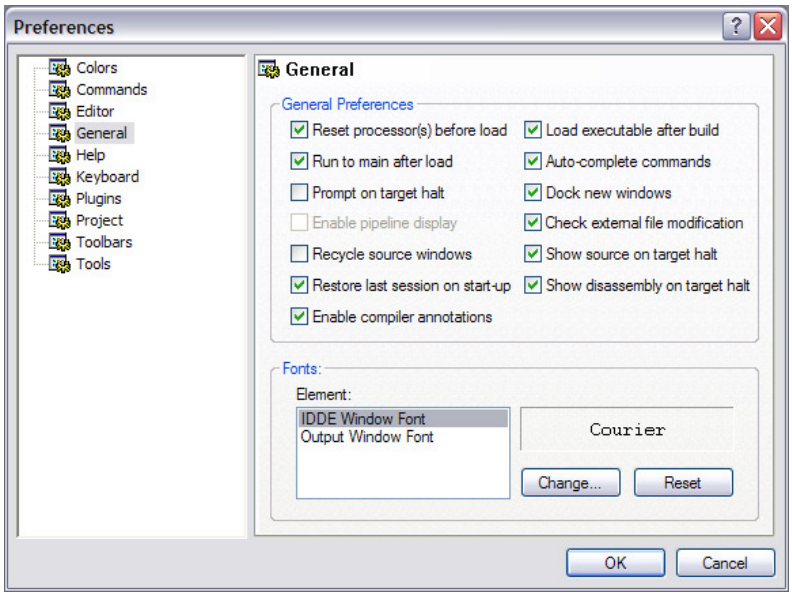

Figura. 4.18. Cuadro de diálogo de *Preferences*

6. En la ventana de *Preferences* (Figura 4.18), seleccionar el icono general, deben estar seleccionadas las siguientes opciones:
   - *Run to main after load*
   - *Load executable after build*

7. Dar clic en *OK* para cerrar el cuadro de diálogo de *Preferences*.

Aparecerá la ventana principal de VisualDSP++ 5.0, y el proyecto esta listo para su construcción.

Paso II: Construcción del proyecto *tutorial_c.dpj*.:

Para construir el proyecto *tutorial_c.dpj*:

1. En el menú *Project*, escoger *Build Project*.

VisualDSP++ 5.0 revisa y actualiza las dependencias del proyecto y entonces construye el proyecto utilizando los archivos fuente.

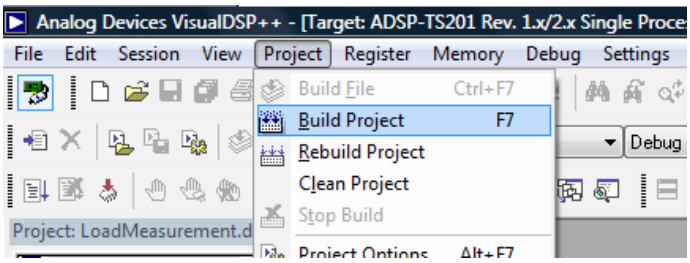

Figura. 4.19. Construcción de un proyecto

Según el progreso de la construcción, en la ventana *Output* muestra el estado de los mensajes (informacional y error) de las herramientas. Por ejemplo cuando una herramienta detecta una sintaxis inválida o una referencia extraviada, la herramienta reporta el error en la ventana *Output*.

Si se da doble clic en el nombre del archivo del mensaje de error, VisualDPS++ 5.0 abre el archivo fuente en el editor de ventana. Se puede editar la fuente para corregir el error, reconstruir y poner en marcha la sesión *Debug*. Si el proyecto construido es actualizado (los archivos, dependencias, y opciones no tienen que cambiar desde la última construcción), ninguna construcción es realizada a menos que se ejecute el comando *Rebuild All*, en lugar de eso se verá "*Project is up to date*.". Si al construir no se tiene errores, un mensaje mostrara lo siguiente *"Build completed successfully."*

En la Figura 4.20 observe que el compilador detecta un identificador indefinido y despliega el error en la pestaña *Build* de la ventana *Output*.

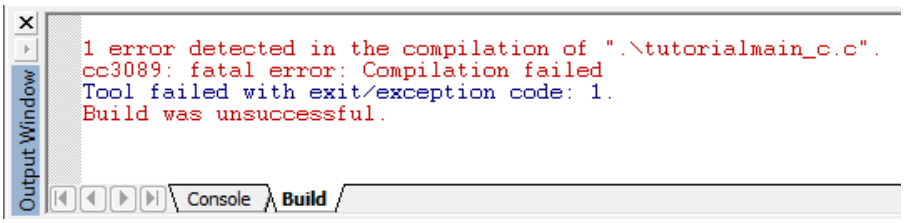

Figura. 4.20. Ejemplo de mensaje de error

2. Dar doble clic en el texto de mensaje de error en la ventana *Output*.

VisualDSP++ 5.0 abre el archivo de código fuente *tutorial_c.dpj* en el editor de ventana y el lugar en las líneas de cursor donde está el error, como indica la Figura 4.21.

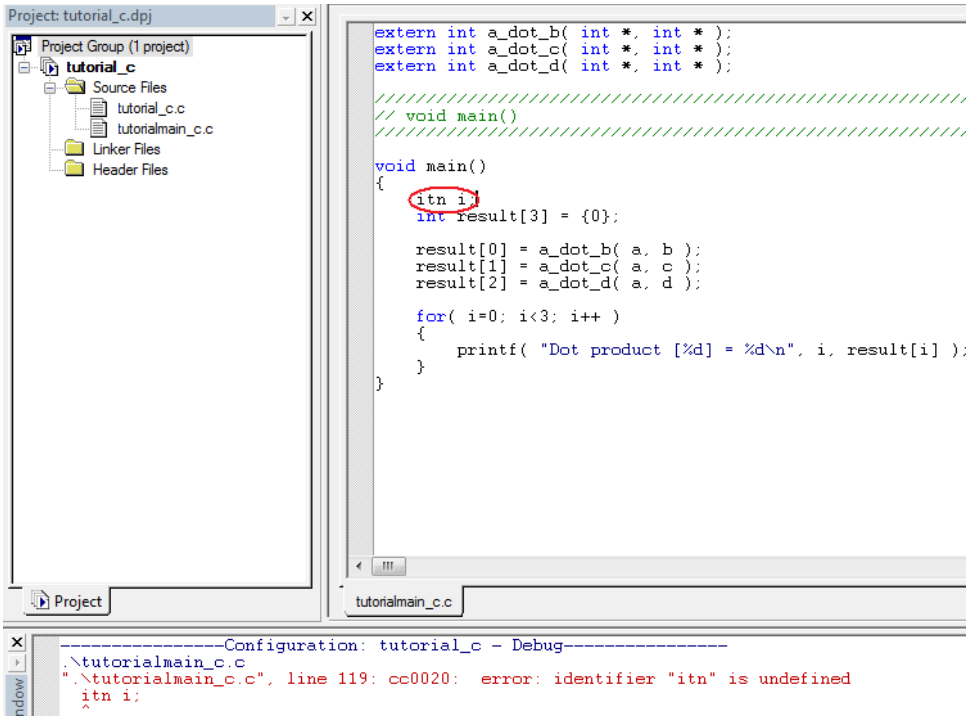

Figura. 4.21. Ventana de edición del archivo fuente que contiene el error

En el editor de ventana se muestra que el comando "*int*", usado para declaración de variables tipo entero, está mal escrito y en su lugar se encuentra erróneamente el comando *"itn"*.

3. En el editor de ventana, dar clic en *itn* y reemplazarlo por *int*. El color del comando *int* cambiará, lo cual significa que el comando ingresado es válido.

4. Guardar el archivo fuente escogiendo *tutorialmain_c.c* desde *File* → *Menu Save.*

5. Construir el proyecto escogiendo otra vez *Build Project* en el menú *Project*. El menú es construido sin ningún error, como se lo puede observar en *Build* de la ventana *Output* (Figura 4.22).

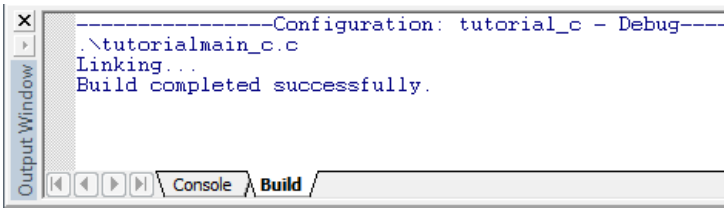

Figura. 4.22. Ventana *Output* que indica construcción exitosa

Paso III: Ejecución del programa

En este procedimiento se debe:

- Establecer la sesión *Debug* antes de correr el programa.
- Observar la ventana del depurador y el cuadro de diálogo.

Desde que se habilitó *Load executable after build* en el icono *General* del cuadro de dialogo de *Preferences*. El archivo ejecutable *dotprodc.dxe* hace la descarga del objetivo automáticamente. Si no se conecta al objetivo *debug*, VisualDSP++ 5.0 hará que se conecte a una, usando una sesión *Debug* existente (o crea una nueva *Debug*).

Escoja la opción *Select a session or create a new session* como (Figura 4.23).

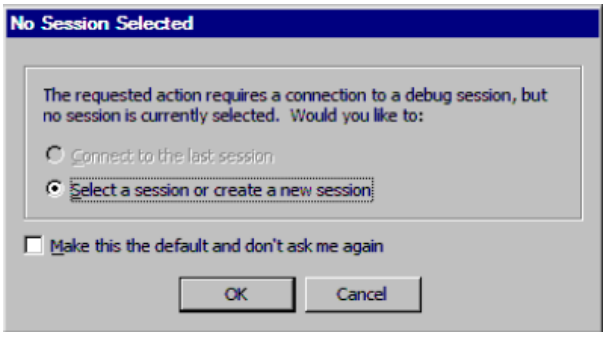

Figura. 4.23. Sesión no seleccionada

Clic en *OK* para crear una nueva sesión. Esto abrirá el asistente de *Sesión*. Una vez creada la sesión, se debe ejecutar el programa como se indica en la Figura 4.24.

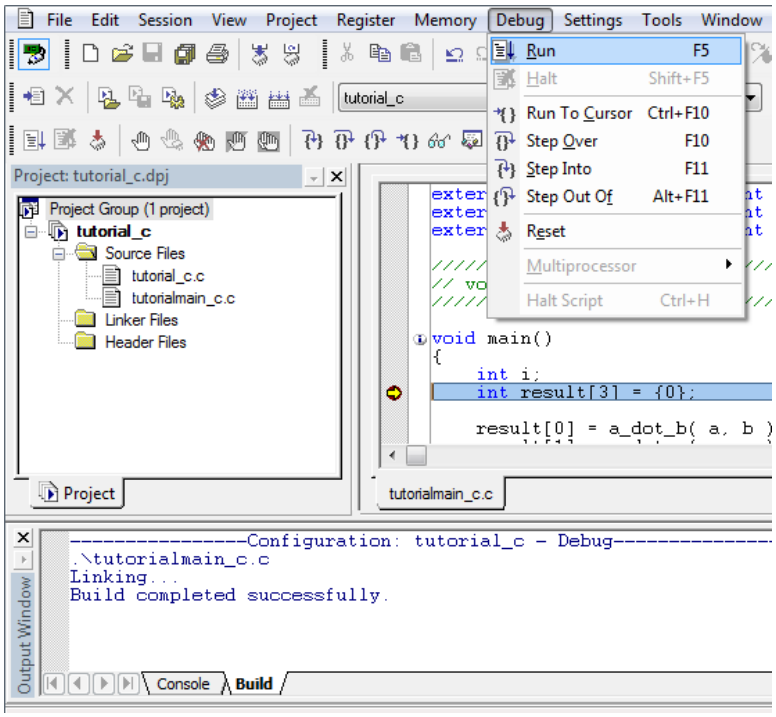

Figura. 4.24. Ejecución de un proyecto.

Los datos de salida que se mostrarán en pantalla son los siguientes:

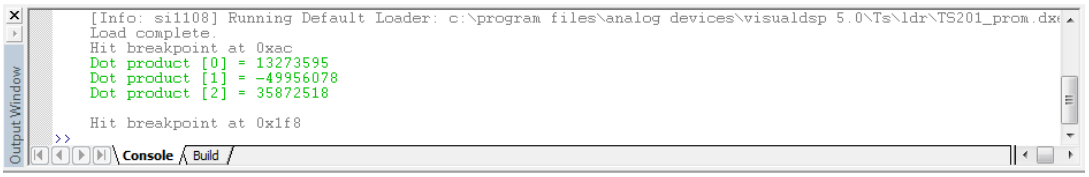

Figura. 4.25. Ventana de salida de datos del programa

4.2.3 Modificar un programa en C para llamar una rutina de lenguaje ensamblador

En este punto se va a realizar lo siguiente:

1. Crear un nuevo proyecto
2. Añadir archivos al nuevo proyecto
3. Crear un Archivo de Enlace para enlazar con la rutina de lenguaje Ensamblador.
4. Modificar el programa en C para llamar una rutina de lenguaje Ensamblador.
5. Reconstruir el proyecto.

Paso I: Crear un nuevo proyecto

Para crear un Nuevo proyecto:

1. Del menú *File*, escoger *Close* y después *Project tutorial_c.dpj* para cerrar el proyecto tutorial_c. Dar clic en *Yes* cuando se pregunte si se desea cerrar todas las ventanas de origen.
2. Del menú *File*, escoger *New* y después *Project* para abrir el *Project Wizard*, que se muestra en la Figura 4.26.

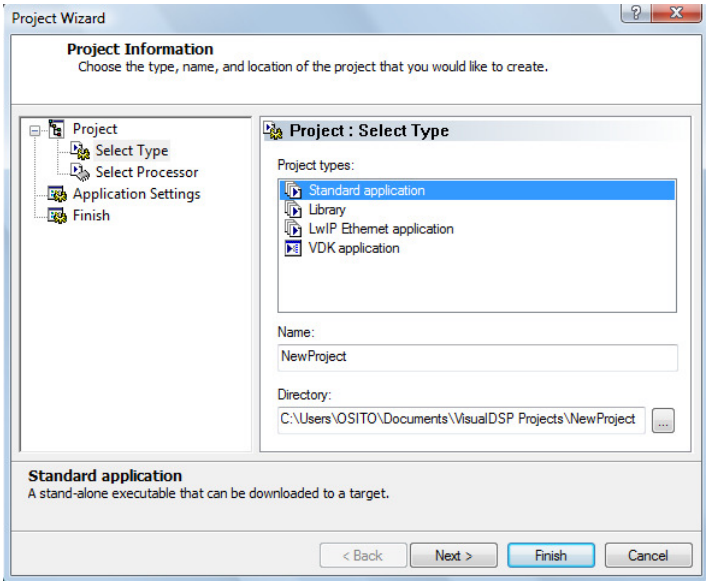

Figura. 4.26. Ventana del asistente para creación de proyectos

3. Seleccionar *Standard application* en el cuadro de *Project types.*
4. En el nombre del archivo escribir tutorial_asm_c.
5. Presione *Next* para mostrar la página de Selección del Procesador.
6. Verifique que el Tipo de Procesador sea ADSP-TS201 y la *Silicon Revision* esté en *Automatic*. Presione *Next* para mostrar la página *Application Settings*.
7. En *Select the project output type for your application*, verificar que *Executable* (.dxe) sea seleccionado.

Paso II: Añadir archivos al Nuevo proyecto tutorial_asm_c

Para añadir archivos al Nuevo proyecto:

1. Desde el menú *Project* seleccione la opción *Add to Project*, y después escoja *File(s).*

Aparecerá la página para añadir archivos al proyecto y se escogerá los archivos deseados. Para nuestro caso en primer lugar escogeremos los archivos *tutorial_c.c y tutorialmain_c.c* usados en el ejercicio anterior y presionamos *Add*. Además, se va a añadir al proyecto el archivo *tutorial_fun.asm*, creado previamente, con el cual vamos a realizar el enlace. A continuación se muestra la Figura 4.27 donde se añade los archivos:

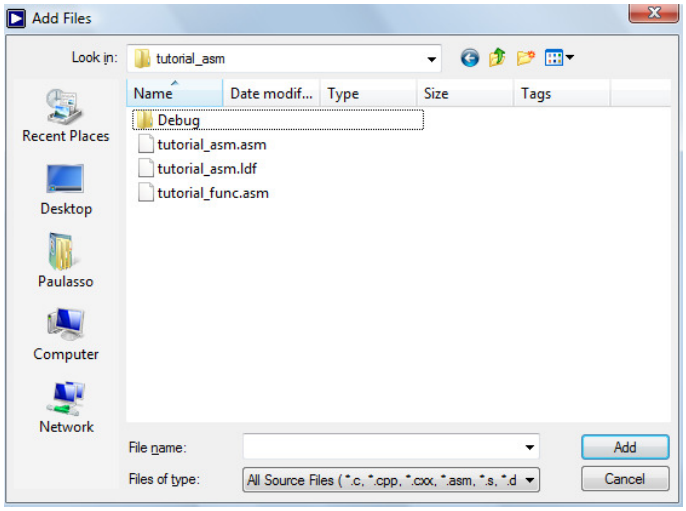

Figura. 4.27. Añadir archivos a un proyecto

El archivo *tutorial_fun.asm* se muestra a continuación:

```
// ASSEMBLY DOT PRODUCT FUNCTION
// File: tutorial_func.asm

.SECTION my_asm_section;

.global _a_dot_c_asm;

_a_dot_c_asm:


    #ifdef XR0_SET_TO_2
    xR0=0x2;;
    #else
    xr0=j5;;
    #endif

    xr1=j4;;
    J1 = xr0;;
    JL1 = 0;;
    J2 = xr1;;
    LC0 = 0x19;;

    loop:
    xr2=[j31+j1];;
    xr1=[j31+j2];;
    j2=j2+1;;
    j1=j1+1;;
    xr17:16=r2*r1(I);;
    xr19:18=r19:18+r17:16;;
    IF NLC0E, JUMP loop;;

    cjmp(np);;

_a_dot_c_asm.end:
```

tutorial_func.asm

Figura. 4.28. Archivo tutorial_fun.asm

2. Presione el botón *Rebuild All* ( ) para construir el proyecto. Los archivos en C se abren en la ventana de edición y la carga se completa.

La versión en C del proyecto está ahora completa. A continuación se procederá a modificar los archivos en C para llamar a la función en lenguaje ensamblador.

Paso III: Modificar los archivos en C del Proyecto

Seguir el siguiente procedimiento:

1. Modificar *tutorialmain_c.c* para llamar la función *a_dot_c_asm* en lugar de *a_dot_c*.

2. Declare y llame la función *a_dot_c_asm* como se muestra en la figura 4.29.

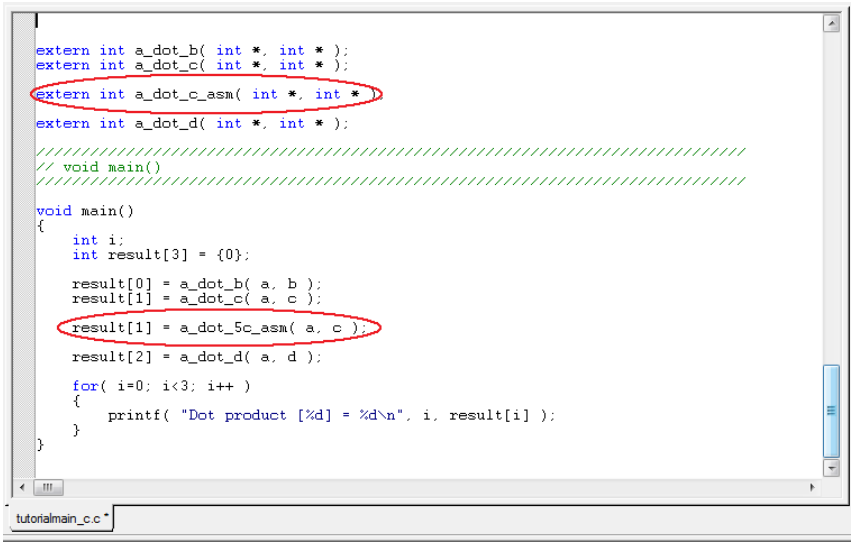

Figura. 4.29. Modificación de archivos en C

3. Comente la función *a_dot_c* utilizando el comando "//" al inicio de la línea de declaración de la función como de su llamado. La figura 4.30 muestra este procedimiento:

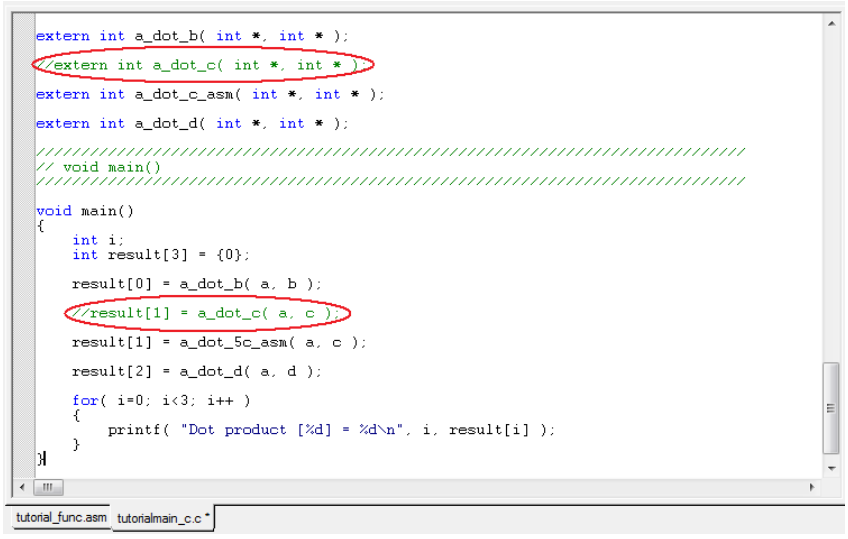

Figura. 4.30. Invalidar la función a_doc_c

4. Guarde el archivo con los cambios realizados.

Paso IV: Crear un Archivo de Enlace para la rutina de lenguaje ensamblador

1. En el menú principal seleccione *Tools* y después la opción *Expert Linker* y escoger la opción *Create LDF*. Se mostrará la siguiente pantalla:

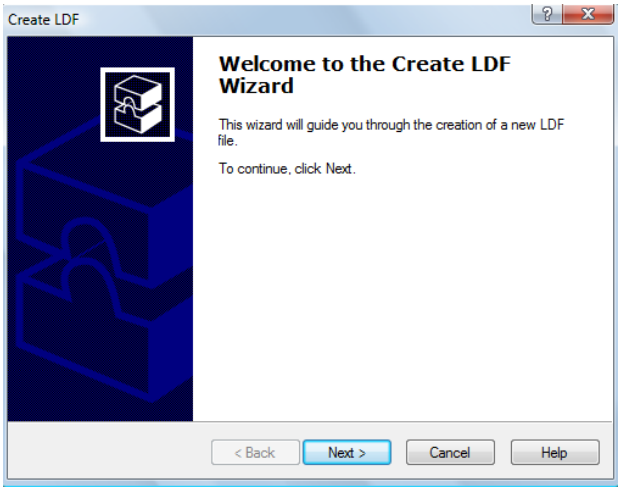

Figura. 4.31. Creación de un LDF

2. Presione *Next* para mostrar la página *Project Information*. En esta página se da el nombre al archivo LDF y se escoge el tipo de proyecto que para nuestro caso es *Tipo C* ya que el archivo principal del proyecto está en lenguaje C.

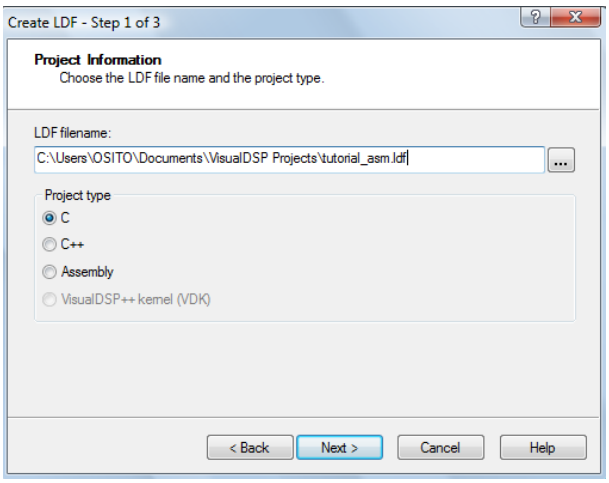

Figura. 4.32. Configuración de archivo de enlace LDF

Presione *Next* en los pasos siguientes y finalmente *Finish* para crear el nuevo archivo de enlace LDF que se mostrará en la ventana del Proyecto.

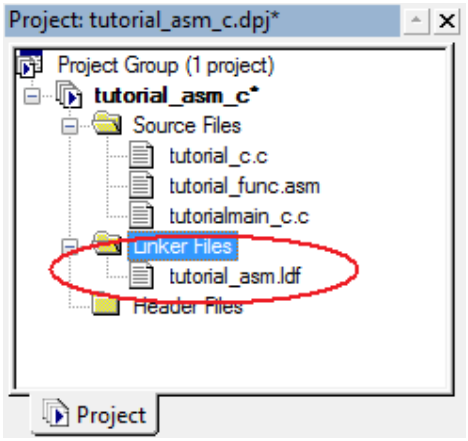

Figura. 4.33. Archivo de enlace LDF

Paso V: Usando *Expert Linker* para modificar el archivo de enlace *tutorial_asm.ldf*

En este procedimiento:

1. Construya el proyecto.
2. Observe el error que aparece en la ventana de salida (Figura 4.34).

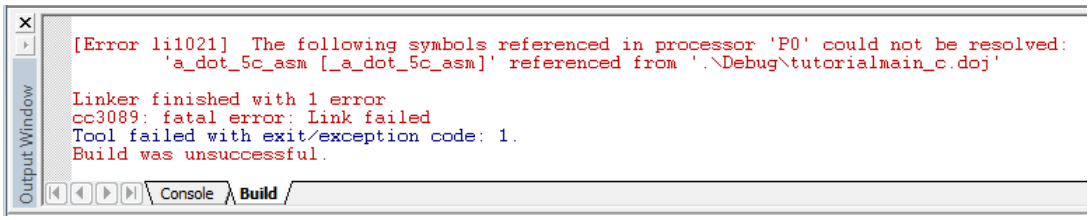

Figura. 4.34. Ventana de error de archivo de enlace LDF

3. En la ventana del proyecto, hacer doble clic sobre el archivo de enlace *tutorial_asm.ldf*. La ventana *Expert Linker* (Figura 4.35) se abre con una representación gráfica del archivo.

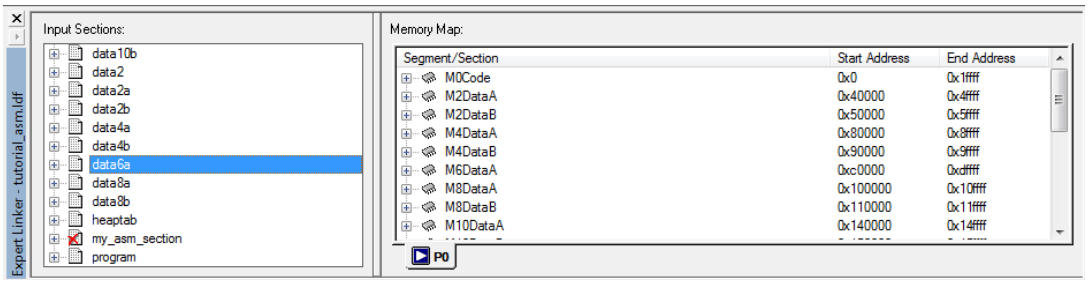

Figura. 4.35. Ventana de Expert Linker

4. Para desplegar la vista tipo árbol del mapa de memoria mostrada en la Figura 4.36, hacer clic derecho sobre dicha ventana, escoger *View Mode*, y después escoger *Memory Map Tree*.

La ventana izquierda (*Input Sections*) contiene una lista de las secciones de entrada que constan en el proyecto o están mapeados en el archivo *.ldf*. Una X roja está sobre el icono en frente de la sección llamada "*my_asm_section*" debido a que el *Expert Linker* ha determinado que la sección no está mapeada por el archivo .ldf.

La ventana derecha (*Memory Map*) contiene una representación de los segmentos de memoria que el *Expert Linker* definió cuando fue creado el archivo .ldf.

5. Para mapear *my_asm_section* dentro del segmento de memoria se realiza lo siguiente:

En la ventana *Input Sections*, se abre *my_asm_section* haciendo clic sobre la pestaña (+) en que está junto a ella. La sección de entrada se expande para mostrar que las macros del enlace $COMMAND_LINE_OBJECTS, $OBJECTS y archivo *tutorial_func.doj* tienen una sección que no ha sido mapeada. En la ventana *Memory Map*, expanda M0CODE y arrastre el icono $OBJECTS de la ventana *Input Sections* dentro de la sección de salida ***code*** bajo M0CODE. Al hacer esto, como se muestra en la Figura 4.36, la X roja desaparece debido a que la sección *my_asm_section* ha sido mapeada.

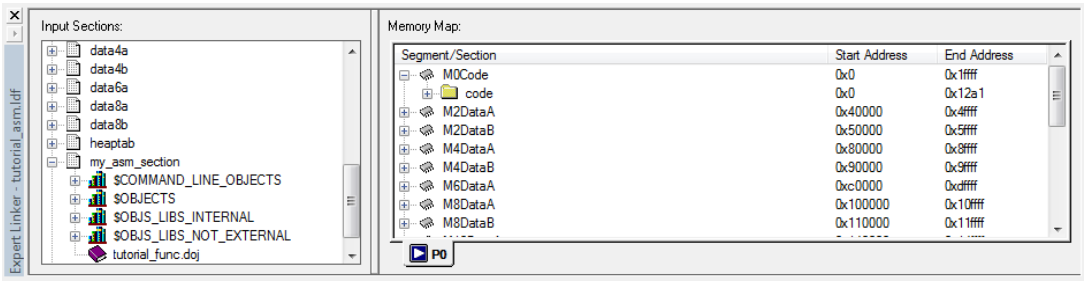

Figura. 4.36. Cambios en *Expert Linker*

6. Guardar los cambios realizados.

Paso VI: Reconstruir y ejecutar tutorial_asm_c

Para ejecutar *tutorial_asm_c*:

1. Construya el proyecto. Al final de la construcción, la ventana de Salida despliega lo siguiente:

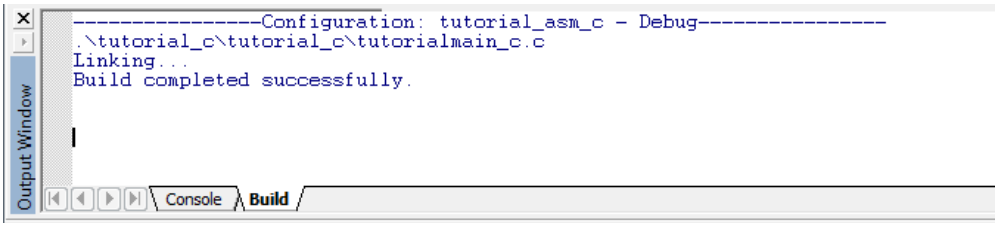

Figura. 4.37. Ventana de salida indicando que es satisfactoria la compilación

2. Presione el botón el botón *Run*  para ejecutar *tutorial_asm_c*.

El programa calcula los productos y muestra los resultados en la ventana *Console* de la ventana de salida. Cuando el programa deja de ejecutarse, el mensaje *"Halted"* aparece en la barra de estado en la parte inferior de la ventana principal de VisualDSP++ 5.0.

Los resultados obtenidos son idénticos a los resultados obtenidos en el ejercicio 1.

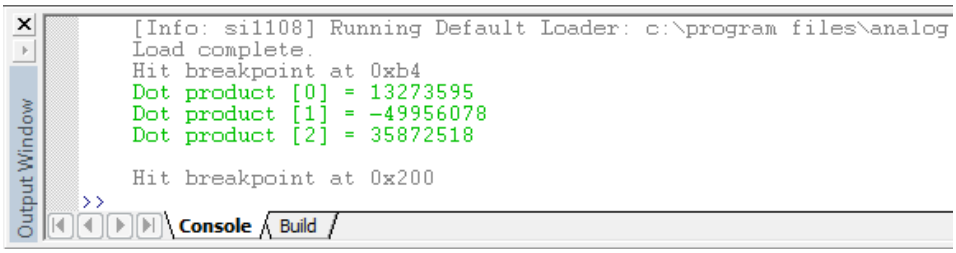

Figura. 4.38. Ventana de resultados después de la ejecución del programa

4.2.4 Representación gráfica de datos

En este punto se va a realizar el siguiente procedimiento:

PASO A: Cargar y depurar un programa preconstruido que aplica un filtro IIR simple a un *buffer* de datos de entrada. El programa se encuentra en la siguiente ruta *C:\Program Files\Analog Devices\VisualDSP 5.0\TS\Examples\No Hardware Required\ADSP-TSxxx\iir_flp32*.

PASO B: Usar la herramienta VisualDSP++ 5.0 *Plotting* para mostrar gráficamente los diferentes arreglos de datos, tanto antes como después de ejecutar el programa.

Paso A: Carga del programa IIR

Para cargar el programa:

1. Mantener la ventana *Disassembly* y la página *Console* (de la ventana *Output*) abiertas, pero cierre el resto de ventanas.

2. Del menú *File,* escoger *Load Program.* El cuadro *Open a Processor Program* aparece.

3. Seleccionar y cargar el programa IIR preconstruido. Para cargar el programa hacer doble clic sobre la subcarpeta *Debug* que se encuentra dentro de la carpeta principal del proyecto. Dar doble clic sobre el archivo .DXE para cargar el programa.

Si VisualDSP++ 5.0 no abre una ventana de edición (mostrada en la figura 4.39) hacer clic derecho en la ventana *Disassembly* y seleccione *View Source*.

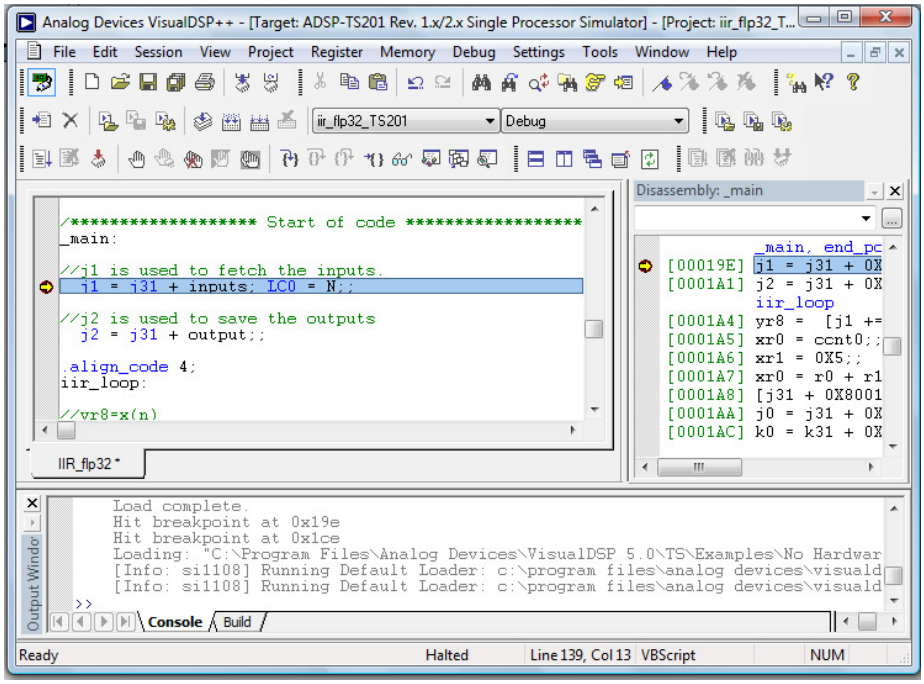

Figura. 4.39. Carga del programa IIR

4. El código fuente del programa contiene dos arreglos de datos globales, *inputs y output*, para los datos de entrada y salida respectivamente.

Paso II: Abrir una ventana *Plot*

Para abrir una ventana *Plot:*

1. Del menú *View,* escoger *Debug Windows* y *Plot.* Después se escoger *New* para abrir el cuadro de configuración del *Plot*, que se muestra en la Figura 4.40. Aquí se añade el conjunto de datos que se quiere mostrar en la ventana *Plot*.

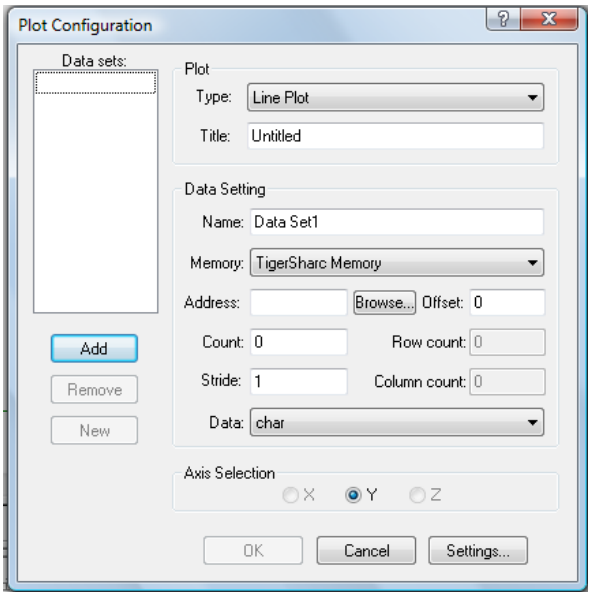

Figura. 4.40. Cuadro de configuración de Plot

En este cuadro se especifican los siguientes valores:

- En el recuadro *Type*, seleccionar *Line Plot*.
- En el recuadro *Title*, escribir el título de la gráfica IIR.

Ingresar los dos arreglos de datos a graficar usando los valores que se muestran en la Tabla 4.4.

Tabla. 4.4. Arreglos de datos: *inputs* y *output*

Recuadro	Datos de entrada	Datos de salida	Descripción
Name	Entrada	Salida	Arreglo de datos
Memory	TigerSharc Memory	TigerSharc Memory	Memoria de datos
Address	inputs	Output	La dirección de estos arreglos de datos es la de los arreglos indata o output. Presione la opción *Browse* para seleccionar el valor de la lista de símbolos cargados.
Count	64	64	El arreglo tiene 128 elementos, por lo tanto se va a graficar todos sus elementos.
Stride	1	1	Los datos están contiguos en memoria.
Data	Float	Float	Los arreglos de entrada y salida son arreglos de valores decimales.

Después de ingresar uno por uno cada arreglo de datos, presionar *Add* para añadir cada arreglo de datos a la lista de datos a la izquierda del cuadro de configuración del *Plot*. El cuadro de configuración del *Plot* debería estar ahora como el que se muestra en la figura 4.41.

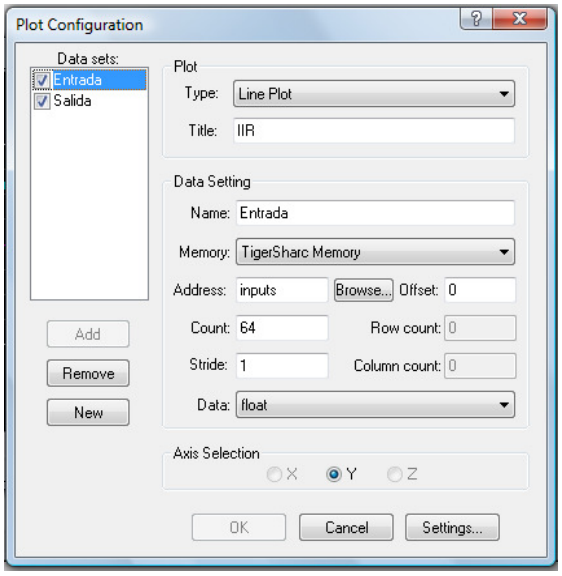

Figura. 4. 41. Cuadro de Configuración del *Plot* con los arreglos de datos de entrada y salida.

Presionar *OK* para aplicar los cambios y abrir una ventana *Plot* con este conjunto de datos. La ventana *Plot* muestra los dos arreglos de datos. Por defecto, el simulador inicializa la memoria en cero; por lo que, los datos de salida (*output*) aparecen como una línea horizontal, como se muestra en la Figura 4.42.

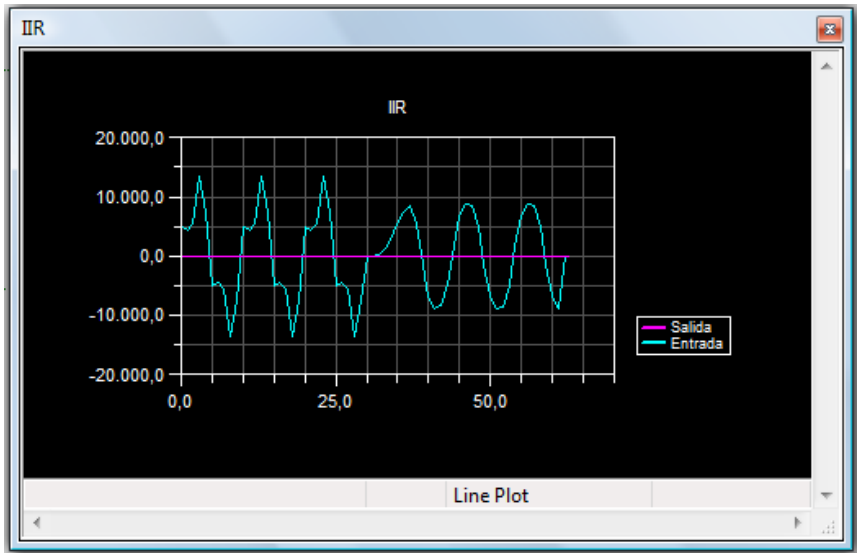

Figura. 4.42. Ventana *Plot*: Antes de ejecución del programa IIR

Dar clic derecho en la ventana *Plot* y escoger *Modify Settings*. En la página general del cuadro de configuración de *Plot* (*Plot Settings*), habilitar *Legend* y presionar *OK* para desplegar el cuadro de leyenda de los datos.

Paso III: Ejecutar el programa IIR y observar los datos.

Para ejecutar el programa IIR y observar los datos:

1. Presione F5 para ejecutar el programa.
   Cuando se detiene el programa, se observará los resultados del filtro IIR en el arreglo de salida *output*. Los dos arreglos de datos se encuentran ahora visibles en la ventana *Plot*, como se muestra en la figura 4.43.

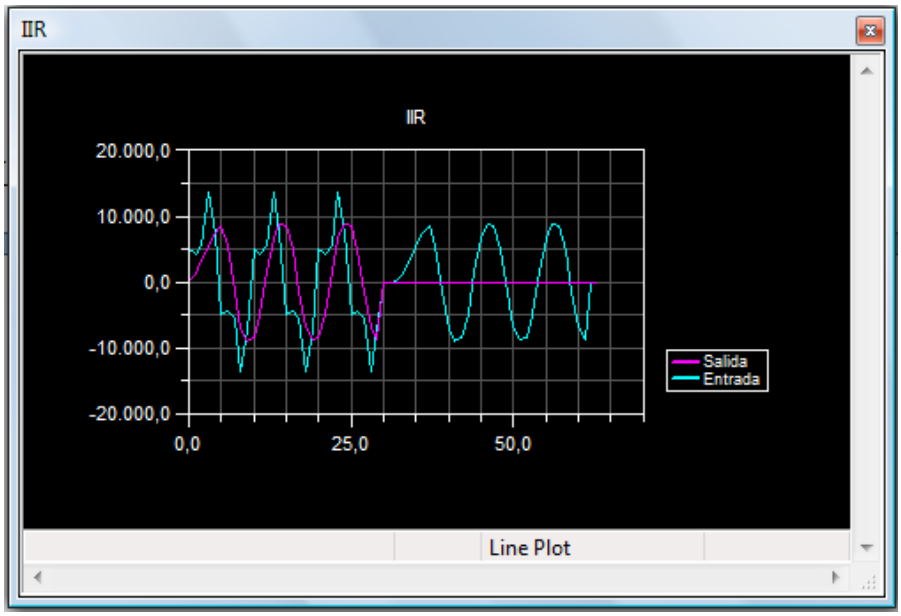

Figura. 4.43. Ventana *Plot*: después de la ejecución del programa IIR.

4.3 VISUAL DSP++ 5.0 KERNEL (VDK)

Las características que se presentan a continuación muestran varios de los beneficios de usar un *Kernel* sobre un procesador.

4.3.1 Desarrollo Rápido de Aplicaciones

La integración entre el ambiente VisualDSP++ 5.0 y el VDK permite un desarrollo rápido de aplicaciones. El uso de una generación de código automático y archivos *templates*, así como una interfaz de programación estándar para los dispositivos, permite concentrarse sobre los algoritmos y el control de flujo deseado que en lugar de los detalles de la implementación. El VDK soporte del uso de lenguaje C, C++ y ensamblador.

4.3.2 Reutilización de Código

Muchos programadores inician un nuevo proyecto escribiendo las partes que se encargan de la transferencia de datos. Esta necesaria lógica de control usualmente es creada para cada diseño y frecuentemente no es reutilizado para proyectos subsecuentes. El VDK provee esta funcionalidad en una librería. Además el *Kernel* esta diseñado para promover una organización ya que permite particionar largas aplicaciones en bloques comprensibles.

El *Kernel* esta diseñado exclusivamente para aprovechar las aplicaciones comunes del usuario y alentar a la reutilización de dichos códigos.

4.3.3 Particionamiento de una aplicación

Una cadena VDK es una encapsulación de un algoritmo y sus datos asociados. Al iniciar un nuevo proyecto, se usa este concepto de cadena para aprovechar la arquitectura *Kernel* y reducir la complejidad del sistema. Debido a que muchos algoritmos pueden estar compuestos de sub algoritmos, una aplicación puede ser particionada en unidades funcionales más pequeñas (*threads*) que pueden ser individualmente codificadas y probadas. Estos sub algoritmos después pueden llegar a ser componentes reutilizables en sistemas más robustos y escalables.

4.3.4 Creación de un proyecto VDK

Para crear un proyecto VDK se realiza los siguientes pasos:

1. Del menú *File*, escoger *New* y después *Project* (VDK no puede ser añadido a un proyecto existente). El *Project Wizard* aparece.
2. En *Project Type* seleccione *VDK aplication*, seleccionar el nombre del proyecto y directorio.
3. Seleccionar *Next*. La lista de procesadores soportados por VDK aparecerá.
4. Especificar el procesador a ser utilizado.
5. Seleccionar *Finish* para crear el proyecto VDK.

El IDDE automáticamente genera los archivos de origen VDK (VDK.cpp y VDK.h) y los añade al proyecto. Un tercer archivo, *nombre_proyecto.vdk,* es añadido al proyecto para habilitar el soporte VDK en el IDDE. No modificar, sobrescriba o remover VDK.h o VDK.cpp

Para mayor información, hacer referencia al documento *50_vdk_ug.pdf* contenido en el CD de *EZ-KIT Lite*.

CAPITULO V

PROGRAMACIÓN DEL PROCESADOR DIGITAL DE SEÑALES ADSP-TS201 Y DESARROLLO DE APLICACIONES

5.1 SET DE INSTRUCCIONES

Se tiene instrucciones muy específicas que difícilmente se encuentran en otro microprocesador, el conjunto de instrucciones están claramente optimizadas para aplicaciones DSP.

5.1.1 Instrucciones ALU

La ALU realiza todas las operaciones aritméticas (adición/sustracción) para el proceso de datos de formato punto-fijo y punto flotante, realiza operaciones lógicas en formato punto-fijo. Estas instrucciones se describen en la tabla 5.1.

Tabla. 5.1. Instrucciones ALU

Rs = rm +/- rn	**Adición/Sustracción** Suma/sustrae los operandos de los registros *Rm* y *Rn*. El resultado es colocado en el registro *Rs.* No existe ningún registro especial que actúe como acumulador. Para realizar las operaciones en punto flotante se incluye la "f". Ejs: r1 = r1+ r5;; (punto fijo) fr1 = r1 – r5;; (punto flotante)

Rs= rm+rn **Adición/Sustracción con acarreo/Pedir prestado**

rq= CI Esta instrucción suma y resta con acarreo los operandos *Rm* y *Rs*. El acarreo (CI) es indicado por la bandera AC en X/YSTAT. Ej:

```
r1 = r0 + r0;;
r2 = CI;;
```

Rs = (rm+rn)/2 **Adición/Sustracción con división para dos**

Esta instrucción suma o resta los operandos *Rm* y *Rn* y divide el resultado para dos en el registro *Rs*, con redondeo al par más cercano. Ejemplo:

```
r9 = (r3 + r5)/2;;
```

Rs = abs rm **Valor absoluto**

Esta instrucción toma el valor absoluto del operando *Rm*, y coloca el valor absoluto del resultado normalizado en el registro *Rs*.

Ejemplo:

```
r5 = abs r4;;
r3 = abs (r0 + r1);;
```

Rs = - rm **Complemento a dos**

Permite sacar el complemento a dos al operando *Rm*, el resultado es guardado en el registro *Rs*. Ejemplo:

```
r2 = - r1;;
```

Rs = MAX(rm,rn) **Máximo/mínimo**

rs = MIN(rm,rn) Estas instrucciones retornan el máximo o mínimo entre dos operandos en los registros *Rm* y *Rn*. El resultado es ubicado en el registro *Rs*. La comparación es realizada byte a byte o palabra a palabra dependiendo del tamaño del dato.

Ejemplo:

```
Sr9:8= MAX (r3:2, r1:0);;
```

rs = INC Rm **Incremento/Decremento**

rs = DEC Rm Estas instrucciones añaden uno o restan uno al operando en el registro *Rm*. El resultado es ubicado en el registro *Rs*.

Ejemplo:

```
r6 = INC r3;;
r6 = DEC r3;;
```

COMP (rm,rn) **Comparación**

Esta instrucción compara el operando en el registro *Rm* con el operando en el registro *Rn*. La instrucción pone en set la bandera AZ si los dos operandos son iguales, y la bandera AN si el operando en el registro *Rm* es mas pequeño que el operando en *Rn*. Ejemplo:

```
COMP (r3, r5) ;;
```

rs = ones rm **Conteo de unos**

Esta instrucción cuenta el numero de unos en el operando en el registro *Rm*. El resultado es ubicado en el registro *Rs*. Ejemplo:

```
r6 = ONES r3;;
```

AND, AND NOT, OR o XOR **Operaciones Lógicas**

Estas instrucciones realizan operaciones lógicas AND, AND NOT, OR o XOR de los operandos en los registros *Rm* y *Rn*. La instrucción NOT lógicamente complementa el operando en el registro *Rm*. El resultado es ubicado en el registro *Rs*. Ejemplo:

```
r5 = r4 AND NOT r8;;
r3 = r2 XOR r7;;
```

rsd=expand S*rm* **Expansión**

Estas instrucciones suman o restan los operandos en los registros *Rm* y *Rn*, después analizan los resultados y de acuerdo a estos expanden valores de 8 bits a 16 bits o valores de 16 a 32 bits. Ejs:

```
r3:2 = expand sr4;;    expande dos palabras cortas
                       a dos normales.
```

rs = fix frm by rn **Conversión de punto-flotante a punto fijo**

Permite realizar una conversión de punto flotante a punto fijo con un factor de escalamiento determinado por *Rn*. Ejemplo:

```
r0 = fix fr1;;
r0 = fix fr1 by r2;;
```

fr0=float rm by rn **Conversión de punto fijo a punto-flotante**

Realiza la conversión de punto fijo a punto flotante con un factor de escalamiento determinado por *Rn*.

```
fr0 = float r1 BY r2;;
```

If condición salto — Esta instrucción permite probar una bandera o una condición, si es verdadera o menor que cero salta. Ejemplo:

```
lc0=2;;
if NLC0E, jump nlms_alg;
```

El contador lc0 se decrementa cada que pasa por la condición y salta a la etiqueta nlms_alg si el valor de lc0 es cero.

5.1.2 Instrucciones IALU

La IALU realiza operaciones aritméticas y lógicas sobre datos de punto fijo (enteros), los registros IALU son J30 hasta J0 y del K30 hasta K0. Las instrucciones IALU se detallan en la Tabla. 5.2.

Tabla. 5.2. Instrucciones IALU

[js+1]=rm — Permite cargar el valor de un registro universal en la posición de memoria de datos determinada por el índice *Js*. Después del acceso se le suma o resta el valor del modificador "1" a *Js*.

Ejemplo:

```
CB[j0+=-1]=xr0;;
CB[j0+=1]=xr3;;
```

rs=CB[ks+=1] — Permite cargar el valor que se encuentra en la dirección determinada por el índice *Ks*. Después del acceso se le suma o resta el valor del modificador "1"a *Ks*. Ejemplo:

```
xr4=CB[k0+=1];;
```

rs=[J31+jm] — Usa un índice que es puesto en cero (registro *J31*), permite cargar el dato de la posición de memoria indicado por el índice *Jm* a un registro universal o IALU. Ejemplo:

```
rs=[J31+0x004553];;
j2=[j31+j0];;
```

[js+jm]=rn — Carga el valor del registro *Rn* en la dirección de la suma del índice *Js* y el modificador *Jm*. Ejemplo:

```
[j0+j3]=r1;;
```

CB[js+=1]=rm	**Buffer Circular** Para usar la opción CB, las instrucciones de suma y resta IALU requiere que los registros de base y longitud J-IALU o K-IALU relacionados sean previamente configurados. Las instrucciones de suma y resta IALU con la opción CB calculan la dirección modificada del índice sumado o restado el modificador. El IALU ubica el valor modificado en *JS* o *KS*.

5.1.3 Instrucciones del multiplicador

El multiplicador realiza operaciones de multiplicación para el procesador sobre datos de punto-fijo y punto-flotante y realiza operaciones de multiplicación/acumulación para el procesador sobre datos de punto-fijo. Esta unidad además realiza operaciones de multiplicación complejas para el procesador de datos de punto-fijo. El multiplicador ejecuta operaciones de compactación de datos sobre resultados acumulados cuando se mueve datos al archivo de registro en formatos punto-fijo. Las instrucciones del multiplicador se describen en la Tabla. 5.3.

Tabla. 5.3. Instrucciones del multiplicador

rs = rm*rn	**Multiplicación** Realiza el producto de dos registros en punto fijo o punto flotante. Ejemplos: r0 = r1 * r2;; fr0= r3 * r4;;
MRsd +=rm*rs	**Multiplicación/Acumulación** Realiza la multiplicación entre los operandos *Rm* y *Rn*, después el resultado de la multiplicación es sumado al valor del registro *MR*. Ejemplo: MR3:2 += r1 * r2;;

MRsd +=rm** rn **Multiplicación/Acumulación de complejos**

Esta es una operación de multiplicación/acumulación de complejos de 16 bits que multiplica el valor complejo en el registro *Rm* con el valor complejo en *Rn*, y suma o resta el resultado a los registros *MR* especificados. Ejemplo:

MR1:0 += r0 ** r1;;

Rsd = MRmd **Registro MR (Resultado de la multiplicación)**

Carga/Transferencia

Estas instrucciones transfieren el valor del registro de origen (a la derecha del =) al registro destino (a la izquierda del =).

Ejemplo:

r1:0 = MR3:2

rsd = SMr2 ;; **Extraer palabras del registro MR**

Estas instrucciones extraen los valores en el registro de origen MRb o MRa y extraen los bits de aviso del registro MR4 después almacena los datos extraídos en el registro de destino *Rsd* o *Rsq*. Ejemplo:

r1:0 = SMR2 ;;

La instrucción *Rsd* = SMR0/1/2/3 (u); extrae los dos valores cortos de un registro simple *MR* y sus bits de aviso en MR4 hacia un registro doble.

5.1.4 Instrucciones de Desplazamiento

El desplazador realiza operaciones de bits (desplazamientos lógicos y aritméticos) y operaciones de campos de bits (depósito y extracción de campo) para el procesador. El desplazador además ejecuta operaciones de conversión de datos tales como conversiones de formato punto-fijo/flotante. Las instrucciones del desplazador se describen en la Tabla. 5.4.

Tabla. 5.4. Instrucciones del desplazador

rs =lshift rm by rn;	**Desplazamiento Aritmético/Lógico** Estas instrucciones aritméticamente o lógicamente desplazan el operando del registro *Rm* por el valor en el registro *Rn*, lo desplaza a través del valor inmediato de la instrucción. Un valor positivo de *Rn* indica un desplazamiento a la izquierda y un valor negativo un desplazamiento a la derecha. El resultado desplazado es ubicado en el registro *Rs*. Ejemplo: R5 = lshift r3 by -4;
rs = rot rm by rn;	**Rotación** Esta instrucción rota el operando en el registro *Rm* a través de un valor determinado por el operando en el registro *Rn* o por el valor del bit inmediato en la instrucción. El resultado rotado es ubicado en *Rs*. Los valores de la rotación son números de complemento a dos. Valores positivos resultan en una rotación a la izquierda, valores negativos en una rotación a la derecha. Ejemplo: R5 = rot r3 by -4;
Lrsd= fext *rmd* by *rn*	**Extracción de Campo** Esta instrucción extrae un campo de un registro *Rm* dentro de un registro *Rs* que esta especificado por la información de control en el registro *Rn*. Existen dos versiones de esta instrucción. Una toma la información de control (longitud y posición del campo) de un registro par (*Rnd*). La otra toma la información de control de un registro simple (*Rn*). La longitud del campo esta justificada a la derecha en *Rn* y su longitud es de 7 bits, permitiendo longitudes de 64 bits y cero bits inclusive. La posición del campo que es de 8 bits, permitiendo extracción a la izquierda fuera de escala.

rs +=fdep rm by rn

Depósito de campo

Esta instrucción deposita un campo justificado a la derecha de un registro *Rm* dentro de un registro *Rs*, donde la posición y longitud en el registro destino *Rs* están determinadas por la información de control en el registro *Rn*.

rs = mask rm by rn

Máscara Campo/Bit

Esta instrucción substituye varios bits en el registros *Rs* por los bits en el registro *Rm*, de acuerdo a como determine el conjunto de bits en el registro *Rn*.

rsd=getbits rmq by rnd

Obtención de bits

Esta instrucción extrae un campo de bits de una cadena de bits contigua contenida en el registro cuádruple *Rmq* y almacena el campo extraído en *Rsd*, de acuerdo a la información de control en *Rnd*. Esta instrucción es usada para implementar un bit FIFO. Ejemplo:

R1:0 = GETBITS r7:4 BY r17:16;;

rsd += putbits rmd by rnd

Ubicar bits

Esta instrucción deposita los 64 bits de *Rmd* dentro de una cadena de bits contigua contenida en el registro cuádruple compuesta de BFOTMP en la parte alta y *Rsd* en la parte baja. Los datos son insertados, empezando desde el bit apuntado por el campo BFP en *Rnd*. El campo Len7 de *Rnd* es ignorado.

BITEST *rm* by *rn*

Prueba de bits

Esta instrucción examina el bit #n en el registro *Rm*, de acuerdo a como indique el operando en *Rn* o el valor de bit inmediato en la instrucción.

rs = bclr|bset|btgl *rm* by *rn*

Encerar/Set/Conmutar

Estas instrucciones enceran, ponen en set o conmutan el bit #n en el registro *Rm* como indica el operando en *Rn* o el valor del bit inmediato en la instrucción. El resultado es ubicado en el registro *Rs*.

5.2 OPERACIONES CON UNIDAD ARITMÉTICA LÓGICA (ALU)

Ejemplos de instrucción ALU

Los ejemplos a continuación muestran instrucciones aritméticas de la ALU. Los comentarios con las instrucciones identifican las características claves de la instrucción.

OPERACIÓN 1: LR5:4 = R11:10 + R1:0 ;;

Esta operación es un suma de operandos de entrada punto fijo de 64 bits XR11:10 + XR1:0 y YR11:10 + YR1:0; el DSP ubica el resultado en XR5:4 y YR5:4

OPERACIÓN 2: YSR1:0 = R31:30 + R25:24 ;;

Esta es una operación de suma de cuatro operandos de entrada punto fijo de 16 bits YR31:30 y los cuatro operandos en YR25:24; el DSP ubicada los cuatro resultados en YR1:0.

OPERACIÓN 3: XR3 = R5 AND R7 ;;

Es una operación lógica AND de los operandos de entrada de 32 bits XR5 y XR7; el DSP ubica el resultado en XR3.

OPERACION 4: YR4 = SUM SR3:2 ;;

Esta es una suma lateral con signo de los cuatro operandos de entrada de 16 bits en YR3:2; el DSP ubica el resultado en YR4.

OPERACION 5: R9 = R4 + R8, R2 = R4 - R8 ;;

Es una instrucción doble; la primera parte de la instrucción es una suma punto-fijo de los operandos de entrada de 32 bits XR4 + XR8 y YR4 + YR8; el DSP ubica los resultados en XR9 y YR9; la segunda parte de la instrucción es una resta de los operandos de entrada de 32 bits XR4 - XR8 y YR4 - YR8; el DSP ubica los resultados en XR2 y YR2.

OPERACIÓN 6: FR9 = R4 + R8 ;;

Es una suma de operandos de entrada punto-flotante de 32 bits en XR4 + XR8 y YR4 + YR8; el DSP ubica los resultados en XR9 y YR9.

OPERACIÓN 7: XFR9:8 = R3:2 + R5:4 ;;

Es una suma de operandos punto-flotante de 40 bits (Palabra extendida) en XR3:2 y XR5:4; el DSP ubica el resultado en XR9:8.

5.3 OPERACIONES CON LA IALU

Las operaciones IALU son:

- Operaciones enteras (lógicas y aritméticas).
- Operaciones de carga, transferencia y almacenamiento.

5.3.1 Operaciones enteras

La IALU realiza operaciones lógicas y aritméticas sobre datos de punto-fijo.

Ejemplos:

OPERACIÓN 1: J2 = J1 + J0 ;;

Esta es una suma de punto-fijo de los 32-bits de los operandos de entrada J1 y J0; el DSP ubica el resultado en J2.

OPERACIÓN 2: K0 = ABS K2 ;;

El DPS ubica el valor absoluto del operando de entrada K2 en el registro K0.

OPERACIÓN 3: J2 = (J1 + J0) / 2 ;;

Esta es una suma y división para dos de punto-fijo de los operandos de entrada J1 mas J0, el DSP ubica el resultado en J2.

5.4 OPERACIONES CON EL MULTIPLICADOR

Los siguientes ejemplos proveen un número de operaciones con instrucciones de multiplicación y multiplicación-acumulación.

OPERACIÓN 1: XYR4 = R6 * R8;;

Esta instrucción es una multiplicación fraccional de 32 bits en ambos bloques de cálculo que produce un resultado redondeado de 32 bits.

OPERACIÓN 2: XYR5:4 = R6 * R8;;

Esta instrucción es una multiplicación fraccional de 32 bits en ambos bloques de cálculo que produce un resultado de 64 bits.

OPERACIÓN 3: XR11:10 = R9:8 * R7:6

Esta instrucción es una multiplicación cuádruple de 16 bits; los operandos de entrada son XR9_H x XR7_H, XR9_L x XR7_L, XR8_H x XR6_H y XR8_L x XR6_L; los resultados de 16 bits van a XR11_H, XR11_L, XR10_H y XR10_L respectivamente.

OPERACIÓN 4: XM3:2 += R1 * R0;;

Esta es una multiplicación de operandos de origen XR1 y XR0, y el resultado de la multiplicación es sumado a los contenidos actuales de los registros XMR de destino, el desbordamiento dentro XMR4_H.

OPERACIÓN 5: YM1:0 -= R3 * R2;;

Esta es una multiplicación de operandos de origen YR3 e YR2, y el resultado de la multiplicación es restado de los contenidos actuales de los registros YMR de destino, el desbordamiento dentro YMR4_L.

OPERACIÓN 6: XR7 = MR3:2, MR3:2 += R1 * R0;;

Esta instrucción ejecuta una operación de multiplicación-acumulación y transfiere los registros MR previos dentro del archivo de registro; el valor previo en los registros MR es transferido al archivo de registro.

OPERACIÓN 7: YMR3:0 += R5:4 * R7:6

Esta instrucción realiza cuatro multiplicaciones de cuatro palabras cortas de 16 bits en el registro par YR5:4 y cuatro palabras cortas de 16 bits en el par YR7:6. Los cuatro resultados son acumulados en MR3:0. Los bits de desbordamiento se escriben en el registro MR4.

OPERACIÓN 8: MR3:0 += R9:8 * R7:6;;

Esta instrucción es una multiplicación-acumulación cuádruple de 16 bits con resultados de 32 bits; los operandos de entrada son XR9_H x XR7_H, XR9_L x XR7_L, XR8_H x XR6_H y XR8_L x XR6_L; los resultados de 32 bits acumulados van a XMR3, XMR2, XMR1 y XMR0 respectivamente.

OPERACIÓN 9: XMR1:0 += R9 ** R7;;

Esta instrucción es una multiplicación del valor complejo en XR9 y el valor complejo en XR7. El resultado es acumulado en XMR1:0.

OPERACIÓN 10: XFR20 = R22 * R23 (T);;

Esta es una instrucción de multiplicación de punto-flotante de 32 bits (precisión simple) con resultado de 32 bits. Registros simples seleccionan operaciones de 32 bits.

OPERACIÓN 11 : YFR25:24 = R27:26 * R30:29 (T);;

Esta es una instrucción de multiplicación de punto-flotante de 40 bits (precisión extendida) con resultado de 40 bits. Registros dobles seleccionan operaciones de 40 bits.

5.5 OPERACIONES DE DESPLAZAMIENTO

OPERACIÓN 1: XR5 = LSHIFT R4 BY R3;;

Este es un desplazamiento lógico del registro XR4 de acuerdo al valor contenido en XR3.

OPERACIÓN 2: YR1 = ASHIFT R2 BY R0;;

Este es un desplazamiento aritmético del registro XR2 de acuerdo al valor contenido en XR0.

OPERACIÓN 3: R1:0 = ROT R3:2 BY 8;;

Esta instrucción rota el contenido de cada palabra R3:2 en ambos ALUs, X e Y, ocho espacios y ubica el resultado en XR1:0 e YR1:0.

OPERACIÓN 4: XBITEST R1:0 BY R7;;

Esta instrucción analiza el bit indicado por XR7 en XR1:0 y lo fija de acuerdo a las banderas XSZ y XSN en XSTAT.

OPERACIÓN 5: R9:8 = BTGL R11:10 BY R13;;

Esta instrucción conmuta el bit indicado por R13 en XR11:10 e YR11:10 y ubica el resultado en XR9:8 e YR9:8.

OPERACIÓN 6: XR15 = LD0 R7;;

Esta instrucción extrae el número de ceros principales de XR7 y ubica el resultado en XR15.

5.6 MANIPULACIÓN DE LEDS

Para manipular los leds de la tarjeta TigerSHARC, FLAG2 Y FLAG3, de sus respectivos núcleos se utiliza las siguientes instrucciones para los dos tipos de lenguaje:

- **Programación en lenguaje "C"**

```
#include <builtins.h>
#include <defTS201.h>
void main()
{
        int LED;
        LED=0x0f;
        __builtin_sysreg_write(__FLAGREG, LED);
}
```

El valor cargado en la variable LED es 0x0F, en la siguiente instrucción se carga el valor de LED en el registro FLAGREG, los bits b_{3-0} del registro habilitan como entrada/salida a las FLAG3-0 respectivamente (con "1" salida), y los bits b_{7-4} seleccionan los valores de salida de la bandera FLAG3-0 respectivamente.

- **Programación en lenguaje "ensamblador"**

```
#include <defTS201.h>
_start:
        xR0 = 0x10001100;;
        FLAGREG = xR0;;
```

El valor de XR0 es cargado en el registro FLAGREG, b_{3-2} activan como salida a FLAG3-2 respectivamente. Los bits b_7 =1 y b_6 =0, contienen los valores de salida de FLAG3 y FLAG2 respectivamente.

A continuación se muestra un programa que conjuga tanto la manipulación de LEDS como el empleo de los pulsadores de la tarjeta ADSP-TS201.

- **Programa para manipulación de leds y pulsadores (Archivo "ejerciciodeleds.asm")**

Cuando se ejecuta el programa, se enciende el LED 4 y después se enciende LED 6, al transcurrir un instante se apagan los LEDS, el programa se queda en un estado de verificación del estado de las banderas del registro SQSTAT el cual contiene las banderas de los pulsadores de la tarjeta.

Cuando se pulsa SW9 activa el pin de bandera FLAG0_A y enciende el LED 4, al pulsar SW8 activa el pin de bandera FLAG1_A y enciende el LED 6. Si los pulsadores no son presionados los LEDS permanecerán apagados. Cuando se pulsa SW4 habilita la interrupción IRQ0_A la cual enciende los LEDS 4 y 6.

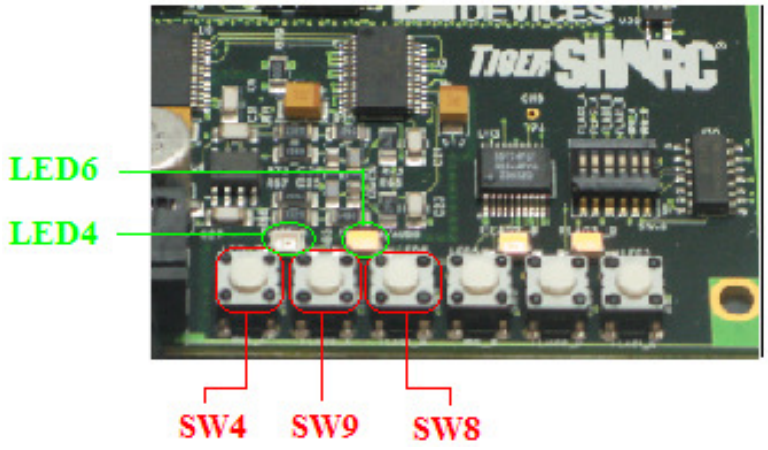

Figura. 5.1. Pulsadores y leds de la tarjeta utilizados en el programa

❖ **Archivo *"ejerciciodeleds.asm"***

```
#include <defTS201.h>
.SECTION program;
.global _main;
_main:
        xR0=0x00000000;;
        FLAGREG=xR0;;          // encera el registro FLAGREG
        xR0 = 0x0000000C;;
        FLAGREG = xR0;;        // habilita como salida FLAG3 y FLAG2
        call retardo(ABS);;    // llama a una función de retardo
        xR0 = 0x0000004C;;
        FLAGREG = xR0;;        //escribe un "1" en FLAG2, en la salida se
                               // enciende el LED4
        call retardo(ABS);;
        xR0 = 0x0000008C;;
        FLAGREG = xR0;;        //escribe un "1" en FLAG3, en la salida se
                               // enciende el LED6
        call retardo(ABS);;

        // Habilitación de la interrupción IRQ0
        j0 = _IRQ0_ISR;;
        IVIRQ0 = j0;;          // pone en set IVIRQ0 del vector interrupción
        XR0 = INT_IRQ0;;
        IMASKH = xR0;;         // interrupción IRQ0 sin máscara
        xr0 = SQCTL;;
        xr0 = bset r0 by SQCTL_GIE_P;  // habilita interrupciones globales
SQCTL = xR0;;

_testflag0:
        xr0=SQSTAT;;
        bitest r0 by 16;;
```

```
        if xSEQ, jump _apaga_flag2 (NP);;       // evalúa el estado de banderas

_enciende_flag2:
        flagregst = FLAGREG_FLAG2_OUT;;   // enciende LED4/FLAG2
        call retardo(ABS);;
jump _testflag1 (NP);;
_apaga_flag2:
flagregcl = ~(FLAGREG_FLAG2_OUT);;          // apaga LED4/FLAG2

_testflag1:
        xR0 = SQSTAT;;
        bitest r0 by 17;;
        if xSEQ, jump _apaga_flag3 (NP);;                  // evalúa las banderas
                                                          // cuando es cero
_enciende_flag3:
        flagregst = FLAGREG_FLAG3_OUT;;    // enciende LED6/FLAG3
        call retardo(ABS);;
        jump _testflag0 (NP);;
_apaga_flag3:
        flagregcl = ~(FLAGREG_FLAG3_OUT);;  // apaga LED6/FLAG3

jump _testflag0(NP);;
        _IRQ0_ISR:
        xr0 = FLAGREG;;
        xr0 = btgl r0 by FLAGREG_FLAG2_OUT_P;;  // conmuta FLAG2
        xr0 = btgl r0 by FLAGREG_FLAG3_OUT_P;;  // conmuta FLAG3
        FLAGREG = xr0;;
        call retardo(ABS);;
.align_code 4;
rti (NP) (ABS);;
```

```
retardo:
        LC1 = 0x10000000;;
        lazo:
                if NLC1E, jump lazo(NP);;
        NOP; CJMP(NP); ._main.END:
```

5.7 ACCESO HACIA Y DESDE MEMORIA / UTILIZACIÓN DE ARCHIVOS EXTERNOS

De Registro a Memoria

Permite cargar el valor de registro en una posición de memoria, el direccionamiento puede ser directo e indirecto. Para el direccionamiento indirecto se utiliza un índice y un modificador.

Cargar en el registro xR1=22 y colocar el valor de xr1 en la posición de memoria 0x80049, realizarlo con direccionamiento directo e indirecto.

- **Direccionamiento directo:**

```
xR1=0x22;;        /*carga el valor de 22 en el registro xr1*/
j0=0x80049;;      /*carga el valor de dirección a la cual va apuntar el índice j0*/
[j0+=j31]=xR1;;   /*carga el valor del registro xR1 a la dirección que está
                    apuntando j0 (80049). */
```

Para direccionamiento directo j0 esta encerado.

- **Direccionamiento Indirecto:**

```
xR1=0x22;;        /*carga el valor de 22 en el registro xr1*/
j0=0x80044;;      /*carga el valor de dirección a la cual va apuntar el índice j0*/
j1=0x5;;          /*carga el valor del modificador*/
```

[j0+j1]=xR1;; /*el valor del registro xR1 es cargado en la suma del índice j0 y modificador j1 (80049) */

5.7.1 De memoria a registro

Permite cargar el dato o valor de una determinada posición de memoria que está dada por el índice "Jx" a un registro universal. Ejemplo:

Cargar el valor de la posición de memoria 0x80010 en un registro universal. Suponiendo que el valor ubicado en la dirección 0x80010 es 24:

j1=[j31+0x80010];; /*carga el valor ubicado en la dirección 0x80010 en el índice J1*/

xr0=j1;; /*carga el valor ubicado en la dirección de memoria apuntada por el índice J1. El registro tendrá el valor de 24 */

- **Programa de aplicación de uso de acceso hacia desde memoria y la utilización de archivos externos**

A continuación se muestra un programa que se enfoca en la utilización de archivos externos y el almacenamiento de datos en memoria.

Para la comprobación del programa seguir los siguientes pasos:

- Crear un archivo "data.txt", dentro del directorio donde se ha creado el proyecto.
- Copiar *"data.txt"* en el directorio *C:\\VisualDSP Projects\ejercicioarchivos\data.txt,* y escribir en el archivo con los valores [7,6,2,1,1] que posteriormente cuando se ensamble se cargaran en *indata*.
- Dentro de Visual DSP cargar manualmente los valores del vector2 [1,2,3,4,8].
- Verificar que los valores de la operación se almacenen en el vector3.

❖ **Archivo "pruebaarchivo.asm**

```
#define N 5                        /*define el tamaño del vector*/
#include <defts201.h>
.section data1;
 .var vector2[N]={ 1,2,3,4,8};      /* vector2 de datos */
.var vector1[N] = "data.txt";       /* vector1 de datos cargados desde un
                                       archivo externo */
.SECTION program;
 .extern _imprimir;                 /* declaración de función
                                       externa */
.ALIGN_CODE 4;
.GLOBAL _main;
.global _vector3;
.var _vector3[N];                   /* vector3 de resultados */
_main:
        LC1=N;;             /*carga N en el contador de lazo*/
        j4=1;;
        j1=vector1;;        /*obtiene la dirección de vector1*/
        j2=vector2;;        /*obtiene la dirección de vector2*/
        j3=_vector3;;               /*obtiene la dirección de vector3*/
_lazo:
        k1=[j31+j1];;       /*carga datos de vector1 en k1*/
        k2=[j31+j2];;       /*carga datos de vector2 en k2*/
        k3=k1+k2;;          /*suma los elementos de los vectores*/
        xr3=k3;;        /*carga el resultado de la suma en el registro xR3*/
        [j3+=j31]=xr3;;     /*carga el dato de xR3 en la dirección de memoria
                            del  vector3*/
        j1=j1+j4;;          /*incremento del puntero del vector1*/
        j2=j2+j4;;          /*incremento del puntero del vector2*/
        j3=j3+j4;;          /*incremento del puntero del vector3*/
        if NLC1E, jump _lazo(NP);;
```

```
call _imprimir;;        /*llama la función que imprime los
                        resultados*/
NOP;;
CJMP(NP);;
._main.END:
```

Para imprimir los resultados de la operación se necesita de la ayuda de un archivo externo en C en donde se encuentra la función imprimir. Para lo cual se añade el archivo "*imp.c*" en el proyecto.

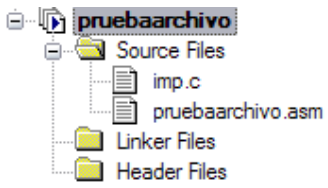

Figura. 5.2. Archivos del proyecto pruebaarchivo

- **Archivo "imp.c"**

```
#include <stdio.h>
extern int vector3[5];
 void imprimir()
{
        int i;
        for (i=0;i<5;i++)
        {
                printf("resultados [%d]= %d\n",i,vector3[i]);
                /*Imprime los elementos del vector3*/
        }
 }
```

CAPITULO VI

MANIPULACIÓN Y REALIZACIÓN DE RUTINAS DE APLICACIÓN CON LA TARJETA ADZS-TS201S

6.1 RUTINA PARA GRABACIÓN DE VOZ

Esta rutina tiene como objetivo realizar una grabación digital de una señal de voz. La rutina permite un tiempo de grabación máximo de 3 minutos. El tiempo de grabación puede ser configurado por el usuario modificando un parámetro que se explicará posteriormente.

Previa la utilización de la rutina se debe conectar al canal de audio de entrada de la tarjeta un micrófono, que permita ingresar la señal de voz, y en el canal de audio de salida unos parlantes o audífonos para verificar la señal de voz ingresada.

Una vez abierto el proyecto que corresponde a la rutina de grabación se debe proceder a construirlo. Durante la construcción, aparecerá una ventana en la cual se permite escoger el procesador en el cual queremos cargar el programa del proyecto (*Grabacion.dxe*). Cabe mencionar, que para todas las aplicaciones que involucren el canal de audio se debe cargar los programas en el procesador A (DSP A) como se indica en la Figura 6.1.

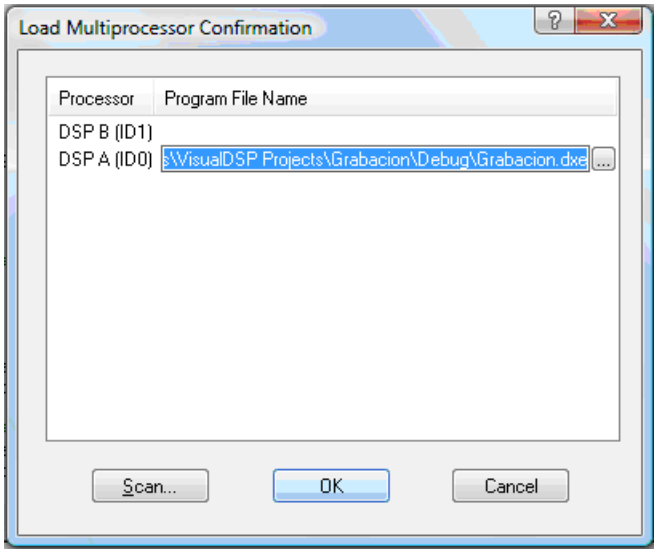

Figura. 6.1. Carga del proyecto Grabacion.dxe en el procesador A

Una vez que el programa ha sido cargado y la construcción del proyecto no ha desplegado ningún mensaje de error, se procede a ejecutar el programa que implementa la rutina de grabación.

6.1.1 Descripción de la rutina de grabación

El proyecto presenta tres estados dentro de su implementación que son los siguientes:

1. Estado *AudioPassThrough*
2. Estado Grabador
3. Estado Reproductor

Una vez que se corre el programa del proyecto, la rutina se encuentra inicialmente en el estado *PassThrough* donde las muestras que ingresen por el canal de audio de entrada son enviadas directamente al canal de salida de audio. Es decir, la señal de voz que ingrese por el micrófono se escuchará en tiempo real por los parlantes o audífonos conectados al canal de salida de audio.

Para proceder a la rutina de grabación, el botón de grabación (*FLAG1_A*) debe ser presionado. En esta etapa, las muestras que ingresen en el canal de entrada son enviadas a al canal de salida, pero además son guardadas en la SDRAM de la tarjeta. Para terminar la grabación debe presionarse nuevamente el botón *FLAG1_A*.

Para acceder a la etapa de reproducción, el botón de play (*FLAG0_A*) debe ser presionado. En esta etapa los valores almacenados previamente en SDRAM son enviados al canal de salida de audio.

En la Figura 6.2 se explica con mayor detalle el LED y los botones que controlan respectivamente las etapas en la rutina de grabación de voz:

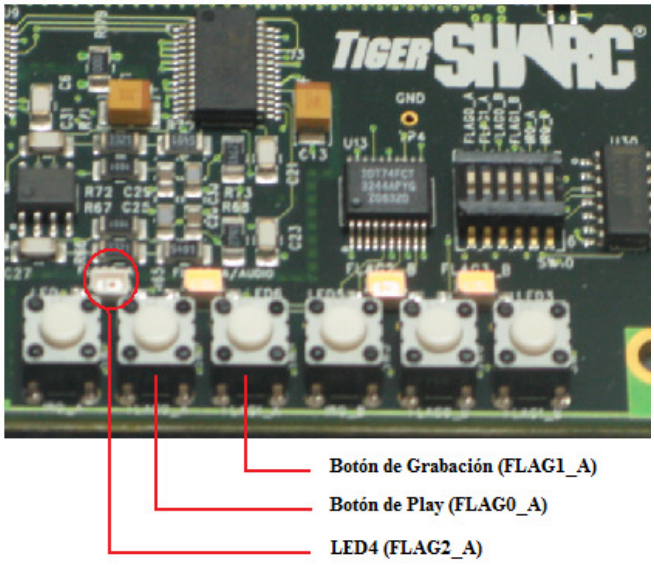

Figura. 6.2. Botones y LED que controlan las etapas de la rutina de grabación

1. ESTADO *PASS THROUGH*:

En el estado *PassTrough*, el código toma las entradas tanto del canal de audio derecho como del izquierdo y las envía directamente a los canales de salida izquierdo y derecho.

Mientras el programa se encuentre en el estado *PassThrough*, el LED4 (*FLAG2_A*) está apagado.

2. ESTADO GRABADOR:

En el estado *Grabador*, el código toma las muestras de los canales de audio de entrada izquierdo y derecho y las envía a los canales de salida izquierdo y derecho. Sólo las muestras que ingresan por el canal izquierdo son además guardadas en la SDRAM de la tarjeta. Al presionar el botón de grabación una vez, se inicia la rutina de grabación de la señal de voz que ingrese en ese momento. Al presionarlo nuevamente, se detendrá la grabación y el código regresa al estado *PassThrough*. Si la cantidad de datos a ser guardados excede el tamaño de SDRAM disponible, el DSP retornará al inicio de la SDRAM y escribirá sobres los datos guardados anteriormente.

Durante el estado Grabador, el LED4 (*FLAG2_A*) está encendido. Si se presiona el botón de *Play* durante la grabación, no ocurre nada.

- **TIEMPO DE GRABACIÓN**

La cantidad de datos que pueden guardarse en SDRAM depende del tamaño que se le dé a este *buffer*. Por defecto, el valor que tiene este *buffer* es de 0x83D600, que corresponde al tamaño máximo que se le puede dar a SDRAM. Para expresar este valor en unidad de tiempo, se emplea el siguiente cálculo:

$$tiempo = \frac{tamaño\,buffer\,SDRAM}{f_{muestreo}}$$

ESPACIO DE MEMORIA

$$tiempo = \frac{0x83D600_{(H)}}{48KHz}$$

$1024 \quad Bytes \rightarrow 1KB$

$8640000\,Bytes \rightarrow x = 8437{,}5\,KB$

$$tiempo = \frac{8640000}{48KHz}$$

$1024 \quad KB \rightarrow 1MB$

$8437{,}5 \quad KB \rightarrow x = 8MB$

$tiempo = 180\,s. = 3\,\text{min},$

El espacio de memoria de la SDRAM es de $8MB$

3. ESTADO REPRODUCTOR

En el estado reproductor, el código toma las muestras de la SDRAM de la tarjeta y las envía al canal de salida izquierdo; y a su vez, toma las muestras actuales y las envía al canal de salida derecho. Esto permitirá al usuario escuchar por el canal izquierdo los datos que fueron almacenados previamente, y por el canal derecho en tiempo real las muestras que están siendo ingresadas.

Al presionar el botón play una vez, iniciará la reproducción. Si se lo presiona nuevamente, se detendrá la reproducción y retornará al estado *Pass Through*. Si se llega al final de los datos almacenados, estos datos empezarán a reproducirse desde el inicio nuevamente. Cada vez que el botón play sea presionado, los datos se reproducirán nuevamente desde el principio de la memoria SDRAM.

Durante el estado de reproducción, el LED4 (*FLAG2_A*) está intermitente. Si se presiona el botón de grabación durante el estado de reproducción no ocurre nada.

6.1.2 Descripción de la implementación del programa

A continuación se explicará detalladamente el programa que implementa una rutina para grabación de voz.

El código, como se explicó en el ítem 6.1.1, es una máquina de tres estados. El usuario se mueve de estado a estado presionando los botones *FLAG0_A* y *FLAG1_A*. Cuando se ingrese a un estado, la variable *_mode* se configura de acuerdo al estado. Cada vez que una interrupción del CODEC ocurra, el procesador usa el valor de la variable *_mode* para decidir si éste debe pasar la muestra, grabarla o reproducir la muestra almacenada en la SDRAM.

El proyecto de rutina de grabación consta de tres archivos .asm y un archivo de enlace .ldf, y son los siguientes:

- *Grabacion.asm*
- *InitA.asm*
- *Audio.asm*
- *Grabación.ldf*

❖ **Archivo "*Grabacion.asm*"**

```
//*************************************************************************
//      Código DSP A para EZ-Kit TigerSHARC ADSP-TS201S
//      Grabacion.asm
//*************************************************************************
//SECCIÓN DE INICIALIZACIÓN
#include <defTS201.h>
```

El archivo "defTS201.h" se encuentra dentro del directorio *C:\Program Files\Analog Devices\VisualDSP 5.0\TS\include*, y mediante está instrucción el ensamblador lo incluye en el momento de la compilación. Este archivo contiene directivas de prepocesador del tipo #define. Básicamente, estas directivas sirven para colocar nombres descriptivos a cada uno de los bits de los registros de sistema DSP.

```
#define pass_through 0
```

Esta variable es usada para definir a su vez la variable *_mode* con el valor de cero para determinar que el programa se encuentra en el estado *pass through*.

```
#define play 1
```

Esta variable es usada para definir a su vez la variable *_mode* con el valor de uno para determinar que el programa se encuentra en el estado de reproducción.

#define record 2

Esta variable es usada para definir a su vez la variable *_mode* con el valor de dos para determinar que el programa se encuentra en el estado de grabación.

.extern _InitA;

Esta instrucción realiza la declaración como función externa de la función *_InitA*, que se encuentra en el archivo *InitA.asm*, la cual se explicará posteriormente.

.extern _mode;

Esta instrucción realiza la declaración como variable externa de la variable *_mode* que se encuentra declarada como global en el archivo *InitA.asm*.

.section/NO_INIT sdram;
.align 4;
.var sdram_address;

Con estas instrucciones se define una dirección dentro del mapa de memoria de datos para la variable *sdram*. Esta variable servirá como *buffer* en el cual se grabarán las muestras que ingresen por el canal de audio.

// SECCIÓN DEL PROGRAMA

.section program;
_start:
call _InitA; nop; nop;;

En este paso se llama a la función InitA. Esta función sirve para inicializar el puerto de audio. La función InitA se encuentra en el archivo *InitA.asm* que será analizado posteriormente.

```
yr30 = FLAGREG;;
yr30 = bset r30 by FLAGREG_FLAG2_EN_P;; // Enable FLAG2 as an output
FLAGREG = yr30;;
```

Con esto se habilita a la bandera FLAG2 como una salida. Primero copia en el registro yr30 el estado actual del registro FLAGREG y después limpia en el registro yr30 la posición correspondiente al bit que indica FLAGREG_FLAG2_EN_P. Finalmente, el valor modificado del registro yr30 es copiado al registro FLAGREG con lo cual se observa que el bit incializado corresponde al habilitador del puerto FLAG2 con lo cual ordena que la bandera FLAG2 se habilite como salida.

```
yr0=pass_through;;
yr1=play;;
yr2=record;;
```

Esta parte del programa carga en los registros yr0, yr1 y yr2 los valores correspondientes de “0”, “1” y “2” para usarlos posteriormente para poder identificar el estado actual en que se encuentra la rutina.

```
j0=j31+sdram_address;;
jb0=j31+sdram_address;;
jl0=0x1;;
j1=j31+sdram_address;;
jb2=j31+sdram_address;;
jl2=0x3FFFFFF;;                    //Tamaño de la SDRAM
j2=j31+sdram_address;;
```

En esta parte se configura el *buffer* circular que se usará para almacenar los datos en la SDRAM. Tómese en cuenta que el valor que se le dé a la variable jl2 (longitud del *buffer* j2) equivale al tamaño de la SDRAM deseado y por lo tanto de este valor dependerá el tiempo de grabación máximo permitido.

En la Figura 6.3 se muestra la parte del archivo Grabacion.asm donde se encuentra la instrucción que determina el tamaño de la SDRAM que controla a su vez el tiempo de grabación.

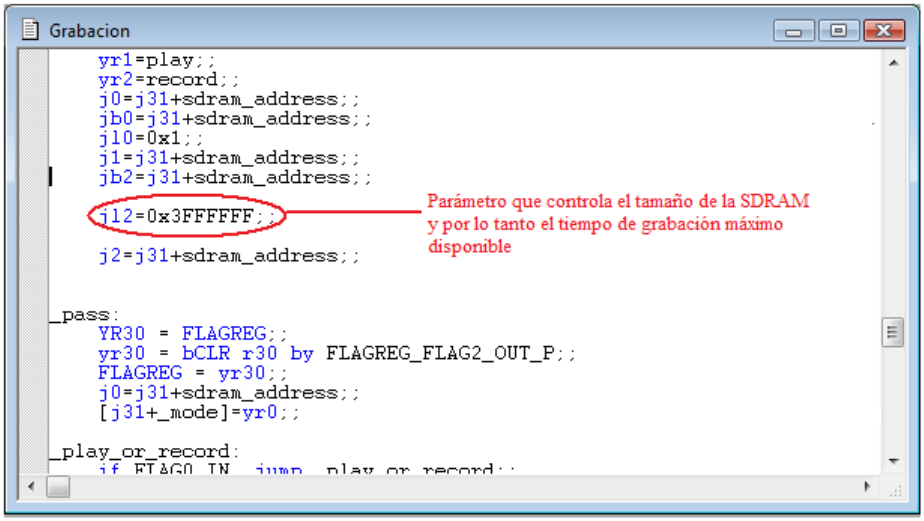

Figura. 6.3. Parámetro que controla el tamaño de la SDRAM

//ESTADO PASSTHROUGH

_pass:

En esta sección el programa se encontrará en estado *passthrough.*

YR30 = FLAGREG;;

yr30 = bCLR r30 by FLAGREG_FLAG2_OUT_P;;

FLAGREG = yr30;;

Con esto se apaga el LED4 (*FLAG2_A*). Primero copia en el registro yr30 el estado actual del registro FLAGREG y después limpia en el registro yr30 la posición correspondiente al bit que indica FLAGREG_FLAG2_OUT_P. Finalmente, el valor modificado del registro yr30 es copiado al registro FLAGREG con lo cual se observa que el bit encerado corresponde a la salida del puerto FLAG2 con lo cual ordena que el LED4 sea apagado.

j0=j31+sdram_address;;
[j31+_mode]=yr0;;

Esta parte del programa inicializa el estado de la máquina a estado *pass through*; ya que, el valor actual del registro yr0 corresponde al mismo de la variable *passthrough*.

_play_or_record:

En esta sección el programa se encontrará en estado de grabación o reproducción.

if FLAG0_IN, jump _play_or_record;;
if FLAG1_IN, jump _play_or_record;nop;nop;nop;;

Con estas instrucciones el programa se asegura que tanto el botón *FLAG0* como el *FLAG1* no estén previamente presionados antes de ingresar al lazo *_play_or_record_bounce*.

_play_or_record_bounce:

if FLAG1_IN, jump _record; nop;nop;nop;;

Si se presiona el botón *FLAG1_A* salta a *_record* en donde se realiza la rutina de grabación.

if FLAG0_IN, jump _play;nop;nop;nop;;

Si se presiona el botón *FLAG0_A* salta a *_play* en donde se realiza la rutina de reproducción.

jump _play_or_record_bounce;nop;nop;nop;;

Si ninguno de los dos botones es presionado permanence en *pass_through*.

//ESTADO REPRODUCCIÓN

_play:

A continuación se muestra la rutina de reproducción de los datos en SDRAM por el canal de audio izquierdo.

```
if FLAG0_IN, jump _play;nop;nop;nop;;
_play_bounce:

[j31+_mode]=yr1;;
```

Configura el modo en estado de reproducción

```
xr30 = FLAGREG;;
xr30 = btgl r30 by FLAGREG_FLAG2_OUT_P;;
FLAGREG = xr30;;
```

Con estas instrucciones el programa hace que el LED4 parpadee para demostrar que el código está en el estado de reproducción.

```
LC0 = 10000000;;
nop;; nop;; nop;; nop;;

_toggle_loop:
nop;; nop;; nop;; nop;;
if NLC0E, jump _toggle_loop (NP);nop;nop;nop;;
```

Con esta parte se cumple con una parte importante del programa; ya que, debido a que en el estado de reproducción el programa debe también dejar pasar las muestras actuales hacia el canal de audio derecho, el DSP necesita este lazo como una forma de espera por la interrupción.

```
if FLAG0_IN, jump _pass; nop;nop;nop;;
```

jump _play_bounce;nop;nop;nop;;

Ahora el botón play funciona como el botón de parada. El botón de grabación en este estado no cumple ninguna función.

//ESTADO GRABACIÓN

_record:

A continuación se muestra la rutina de grabación de los datos que provienen del canal de audio izquierdo en la SDRAM.

YR30 = FLAGREG;;
yr30 = bSET r30 by FLAGREG_FLAG2_OUT_P;;
FLAGREG = yr30;;

Con estas instrucciones el programa hace que el LED4 se encienda para demostrar que el código está en el estado de grabación.

if FLAG1_IN, jump _record;nop;nop;nop;;

Para asegurarse que el botón está presionado antes de ingresar al lazo _record_jump_bounce.

j2=j31+sdram_address;;

Inicializa el índice del *buffer* j2 para proceder a la grabación.

_record_jump_bounce:

[j31+_mode]=yr2;; //configura el modo a *grabación*
if FLAG1_IN, jump _store; nop;nop;;
jump _record_jump_bounce;nop;nop;nop;;

Ahora el botón de grabación funciona como el botón de parada. El botón de reproducción en este estado no cumple ninguna función.

_store:

En esta parte del programa se explica el papel que cumplen los *buffers* y sus índices dentro de las rutinas.

```
j1=j2-sdram_address;;
```

Con esta instrucción se obtiene en el *buffer* el final de la grabación para poder cumplir posteriormente con la reproducción del *buffer* circular.

```
jb0=j31+sdram_address;;
jl0=j1;;
```

Aquí j0 apunta al final del *buffer* después de que el botón de parada es presionado. La base del *buffer* j0 es siempre cero. Por su parte, jl0 guarda la longitud del *buffer* circular en base al número de puntos que han sido grabados,

```
j0=j31+sdram_address;;
```

Aquí se reinicia el vector de reproducción. Por lo tanto, si se desea reproducir el vector, éste empezará desde el inicio.

```
jump _pass;;
_wait:
jump _wait (NP); nop; nop; nop;;
CJMP(ABS)(NP);;
_main.end:
```

En este punto finaliza el archivo *Grabacion.asm*

❖ **Archivo "*Audio.asm*"**

```
/*************************************************************************
    Rutina de Interrupción de Audio para el EZ-Kit TigerSHARC ADSP-TS201s
    Audio.asm
*************************************************************************/

#include <defTS201.h>
#define CODEC          0x38000000

.global _AudioInt;
.extern _ReadDataLeft;
.extern _ReadDataRight;
.extern _WriteDataLeft;
.extern _WriteDataRight;
.extern _mode;
```

Se realiza la declaración de las variables que van a ser utilizadas dentro de este programa. Nótese que la variable *_AudioInt* es declarada como global para permitir su uso externo.

```
//SECCION DEL PROGRAMA
.section program;
_AudioInt:
        xr0=0x0;;
        xr1=0x1;;
        xr2=0x2;;
```

Los registros xr0, xr1 y xr2 serán usados como registros referencia para determinar el estado en que se encuentra el programa.

```
        xr31=[j31+_mode];;
```

El registro xr1 guarda el valor del modo actual.

```
        xr3=r31-r0;;
        if AEQ, jump pass_stage;nop;nop;nop;;
```

```
        xr3=r31-r1;;
        if AEQ, jump play_stage;nop;nop;nop;;

        xr3=r31-r2;;
        if AEQ, jump record_stage;nop;nop;nop;;
```

AEQ será la bandera que decidirá a que modo accederemos dentro de la interrupción. Cuando el resultado de una de las restas realizadas sea igual a cero, entonces AEQ se setea y el programa accede al lazo correspondiente.

```
        xr31=0x0;;
        jump _AudioInt;nop;nop;nop;;
_AudioInt.end:

// INTERRUPCIÓN ESTADO PASS THROUGH

pass_stage:
        xr4 = [j31 + _ReadDataLeft];;
```

XR4 toma la muestra del canal de entrada de audio izquierdo.

```
        [j31 + _WriteDataLeft] = xr4;;
```

Se escribe la muestra que contiene XR4 en el canal de salida de audio izquierdo.

```
        xr5 = [j31 + _ReadDataRight];;
```

XR5 toma los datos del canal de entrada de audio derecho.

```
        [j31 + _WriteDataRight] = xr5;;
```

Se escribe la muestra que contiene XR5 en el canal de salida de audio derecho.

```
jump end_of_int;nop;nop;nop;;
```

Finalmente el programa salta a la etiqueta *end_of_int* para finalizar la interrupción.

```
pass_stage.end:

// INTERRUPCIÓN ESTADO REPRODUCCIÓN

play_stage:
    xr4=CB[j0+=0x1];;
```

Esta instrucción copia en XR4 la muestra almacenada en la SDRAM.

```
    [j31 + _WriteDataLeft] = xr4;;
```

En esta parte se copia al canal de salida izquierdo el contenido de sdram para que sea reproducido.

```
    xr5 = [j31 + _ReadDataRight];;
    [j31 + _WriteDataRight] = xr5;;
    jump end_of_int;nop;nop;nop;;
```

El canal derecho se mantiene en un estado de pass through. Es decir, el canal derecho reproduce las muestras de entrada actuales. Finalmente el programa salta a la etiqueta *end_of_int* para finalizar la interrupción.

```
play_stage.end:

// INTERRUPCIÓN ESTADO GRABACIÓN

record_stage:
    xr4 = [j31 + _ReadDataLeft];;
    [j31 + _WriteDataLeft] = xr4;;  // pass through
```

```
CB[j2+=0x1]=xr4;;
```

En esta parte del programa a más de operar en el estado *pass through*, los datos de entrada son almacenados en SDRAM a través del *buffer* j2. Como se revisó anteriormente en el código, el *buffer* j2 está configurado para ser *buffer* circular y tener el mismo tamaño de SDRAM.

```
xr5 = [j31 + _ReadDataRight];;
[j31 + _WriteDataRight] = xr5;;
jump end_of_int;nop;nop;nop;;
```

El canal derecho se mantiene en un estado de *pass through*. Las muestras que ingresen por el canal derecho solamente se reproducen y no son guardadas en la SDRAM. Finalmente el programa salta a la etiqueta *end_of_int* para finalizar la interrupción.

```
record_stage.end:

nop;nop;nop;nop;;
end_of_int:
nop;nop;nop;nop;;
rti (ABS)(NP);;
```

En este punto finaliza el servicio a la rutina de interrupción.

❖ **Archivo "InitA.asm"**

En este archivo se establece el proceso de inicialización del puerto de audio. Este proceso será el mismo para todos los proyectos que empleen este puerto en el presente proyecto de grado.

```
/*************************************************************************
    Rutina de inicialización del puerto de audio TigerSHARC EZ-Kit ADSP-
    TS201s      InitA.asm
*************************************************************************/

#include <defts201.h>

.global _InitA;
.extern _AudioInt;
.extern _RcveDMALDestinTCB;
.extern _RcveDMALSourceTCB;

section program;
_InitA:

xr0 =   SYSCON_MP_WID64    | SYSCON_MEM_WID64|
        SYSCON_MSH_PIPE2   | SYSCON_MSH_WT0    | SYSCON_MSH_IDLE |
        SYSCON_MS1_PIPE1   | SYSCON_MS1_WT0    | SYSCON_MS1_IDLE |
        SYSCON_MS0_SLOW    | SYSCON_MS0_WT3    | SYSCON_MS0_IDLE;;

SYSCON = xr0;;
```

En esta sección se inicializa el registro SYSCON.

```
xr0 =   SDRCON_INIT | SDRCON_RAS2PC5    | SDRCON_PC2RAS2 |
        SDRCON_REF3700      | SDRCON_PG256    | SDRCON_CLAT2 |
        SDRCON_ENBL;;
SDRCON = xr0;;
```

En esta sección se inicializa el registro SDRCON. Los bits que se muestran configuran al registro de la siguiente forma:

```
xr0 = IMASKL;;
xr0 = bset r0 by INT_DMA0_P;;
IMASKL = xr0;;
```

Aquí se habilita la interrupción del puerto externo DMA0

```
SQCTLST = SQCTL_GIE;;
```

Esta instrucción habilita las interrupciones globales a través del bit SQCTL_GIE en el registro SQCTL.

```
xr0 = FLAGREG;;
xr0 = bset r0 by FLAGREG_FLAG3_EN_P;;  //flag3 es puesta como salida
xr0 = bclr r0 by SER_ENBL_P;;   // Bandera de habilitación serial en 0
FLAGREG = xr0;;
```

La bandera FLAG3 es usada para habilitar el puerto de comunicación serial. En esta parte del programa se configura a la bandera FLAG3 como salida y se fija su valor de salida en cero deshabilitando momentáneamente el puerto serial.

```
j4 = j31 + _AudioInt;;
IVDMA0 = j4;;
```

Configura el vector de interrupciones del canal 0 DMA para iniciar la interrupción en la dirección fijada por _AudioInt.

```
xr3:0 = Q[j31 + _RcveDMALSourceTCB];;
xr7:4 = Q[j31 + _RcveDMALDestinTCB];;
DCS0 = xr3:0;;
DCD0 = xr7:4;;
```

Esta parte indica el inicio de la cadena de DMAs

```
xr0 = FLAGREG;;
```

```
xr0 = bset r0 by SER_ENBL_P;;
FLAGREG = xr0;;
```

Aquí a través del seteo del bit SER_ENBL_P se habilita el Puerto serial.

```
CJMP(ABS)(NP);;
_InitA.end:
```

En este punto finaliza la inicialización del puerto de audio.

- **Archivo "Grabacion.ldf"**

El archivo *Grabacion.ldf* es un archivo de enlace que es creado con la única función de crear un espacio de memoria de datos para SDRAM. Este archivo con respecto a un archivo .ldf por defecto presenta dos variaciones con el objeto de hacer constar a SDRAM dentro de la memoria de la tarjeta. Las dos variaciones mencionadas se muestran en las figuras 6.4 y 6.5.

ARCHIVO .ldf POR DEFECTO

```
MEMORY
{
#if defined(__SILICON_REVISION__) && \
     ((__SILICON_REVISION__ < 0x100)|| (__SILICON_REVISION__ == 0xfff
// For errata 03-00-248 valid for revisions 0.0 and 0.1 we reserve
// memory 0x0001FFA0 through 0x0001FFF for the emulator workaround
    M0Code      { TYPE(RAM) START(0x00000000) END(0x0001FF9F) WIDTH(32) }
#else
    M0Code      { TYPE(RAM) START(0x00000000) END(0x0001FFFF) WIDTH(32) }
#endif
    M2DataA     { TYPE(RAM) START(0x00040000) END(0x0004FFFF) WIDTH(32) }
    M2DataB     { TYPE(RAM) START(0x00050000) END(0x0005FFFF) WIDTH(32) }
    M4DataA     { TYPE(RAM) START(0x00080000) END(0x0008FFFF) WIDTH(32) }
    M4DataB     { TYPE(RAM) START(0x00090000) END(0x0009FFFF) WIDTH(32) }
    M6DataA     { TYPE(RAM) START(0x000C0000) END(0x000CFFFF) WIDTH(32) }
    M6DataB     { TYPE(RAM) START(0x000D0000) END(0x000DFFFF) WIDTH(32) }
    M8DataA     { TYPE(RAM) START(0x00100000) END(0x0010FFFF) WIDTH(32) }
    M8DataB     { TYPE(RAM) START(0x00110000) END(0x0011FFFF) WIDTH(32) }
    M10DataA    { TYPE(RAM) START(0x00140000) END(0x0014FFFF) WIDTH(32) }
    M10DataB    { TYPE(RAM) START(0x00150000) END(0x0015FFFF) WIDTH(32) }

    MS0         { TYPE(RAM) START(0x30000000) END(0x37FFFFFF) WIDTH(32) }
    MS1         { TYPE(RAM) START(0x38000000) END(0x3FFFFFFF) WIDTH(32) }
    MSSD0       { TYPE(RAM) START(0x40000000) END(0x43FFFFFF) WIDTH(32) }
    MSSD1       { TYPE(RAM) START(0x50000000) END(0x53FFFFFF) WIDTH(32) }
    MSSD2       { TYPE(RAM) START(0x60000000) END(0x63FFFFFF) WIDTH(32) }
    MSSD3       { TYPE(RAM) START(0x70000000) END(0x73FFFFFF) WIDTH(32) }

// Memory blocks need to be less than 2 Gig.
    HOST        { TYPE(RAM) START(0x80000000) END(0x8FFFFFFF) WIDTH(32) }
    HOST1       { TYPE(RAM) START(0x90000000) END(0xAFFFFFFF) WIDTH(32) }
    HOST2       { TYPE(RAM) START(0xB0000000) END(0xCFFFFFFF) WIDTH(32) }
```

ARCHIVO Grabacion.ldf

```
MEMORY
{
#if defined(__SILICON_REVISION__) && \
     ((__SILICON_REVISION__ < 0x100)|| (__SILICON_REVISION__ == 0xfff
// For errata 03-00-248 valid for revisions 0.0 and 0.1 we reserve
// memory 0x0001FFA0 through 0x0001FFF for the emulator workaround
    M0Code      { TYPE(RAM) START(0x00000000) END(0x0001FF9F) WIDTH(32) }
#else
    M0Code      { TYPE(RAM) START(0x00000000) END(0x0001FFFF) WIDTH(32) }
#endif
    M2DataA     { TYPE(RAM) START(0x00040000) END(0x0004FFFF) WIDTH(32) }
    M2DataB     { TYPE(RAM) START(0x00050000) END(0x0005FFFF) WIDTH(32) }
    M4DataA     { TYPE(RAM) START(0x00080000) END(0x0008FFFF) WIDTH(32) }
    M4DataB     { TYPE(RAM) START(0x00090000) END(0x0009FFFF) WIDTH(32) }
    M6DataA     { TYPE(RAM) START(0x000C0000) END(0x000CFFFF) WIDTH(32) }
    M6DataB     { TYPE(RAM) START(0x000D0000) END(0x000DFFFF) WIDTH(32) }
    M8DataA     { TYPE(RAM) START(0x00100000) END(0x0010FFFF) WIDTH(32) }
    M8DataB     { TYPE(RAM) START(0x00110000) END(0x0011FFFF) WIDTH(32) }
    M10DataA    { TYPE(RAM) START(0x00140000) END(0x0014FFFF) WIDTH(32) }
    M10DataB    { TYPE(RAM) START(0x00150000) END(0x0015FFFF) WIDTH(32) }

    MS0         { TYPE(RAM) START(0x30000000) END(0x37FFFFFF) WIDTH(32) }
    MS1         { TYPE(RAM) START(0x38000000) END(0x3FFFFFFF) WIDTH(32) }
    SDRAM       { TYPE(RAM) START(0x40000000) END(0x43FFFFFF) WIDTH(32) }
    MSSD1       { TYPE(RAM) START(0x50000000) END(0x53FFFFFF) WIDTH(32) }
    MSSD2       { TYPE(RAM) START(0x60000000) END(0x63FFFFFF) WIDTH(32) }
    MSSD3       { TYPE(RAM) START(0x70000000) END(0x73FFFFFF) WIDTH(32) }

// Memory blocks need to be less than 2 Gig.
    HOST        { TYPE(RAM) START(0x80000000) END(0x8FFFFFFF) WIDTH(32) }
    HOST1       { TYPE(RAM) START(0x90000000) END(0xAFFFFFFF) WIDTH(32) }
    HOST2       { TYPE(RAM) START(0xB0000000) END(0xCFFFFFFF) WIDTH(32) }
```

Figura. 6.4. Modificaciones en el archivo Grabacion.ldf para crear un espacio de datos en SDRAM

ARCHIVO .ldf POR DEFECTO

```
        {
            INPUT_SECTIONS( $OBJECTS(data6a) )
        } >M6DataA

        data6b
        {
            INPUT_SECTIONS( $OBJECTS(data6b) )
        } >M6DataB

        data8a
        {
            INPUT_SECTIONS( $OBJECTS(data8a) )
        } >M8DataA

        data8b
        {
            INPUT_SECTIONS( $OBJECTS(data8b))
        } >M8DataB

        data10a
        {
            INPUT_SECTIONS( $OBJECTS(data10a))
        } >M10DataA

        data10b
        {
            INPUT_SECTIONS( $OBJECTS(data10b))
        } >M10DataB
    }
}
```

ARCHIVO Grabacion.ldf

```
        {
            INPUT_SECTIONS( $OBJECTS(data6b) )
        } >M6DataB

        data8a
        {
            INPUT_SECTIONS( $OBJECTS(data8a) )
        } >M8DataA

        data8b
        {
            INPUT_SECTIONS( $OBJECTS(data8b))
        } >M8DataB

        data10a
        {
            INPUT_SECTIONS( $OBJECTS(data10a))
        } >M10DataA

        data10b
        {
            INPUT_SECTIONS( $OBJECTS(data10b))
        } >M10DataB

        sdram SHT_NOBITS
        {
            INPUT_SECTIONS( $OBJECTS(sdram))
        } >SDRAM
    }
}
```

Figura. 6.5. Declaración de SDRAM en el archivo Grabacion.ldf

6.2 COMUNICACIÓN PUNTO A PUNTO ENTRE TARJETAS ADSP-TS201

Para la implementación del proyecto de la comunicación punto a punto entre tarjetas ADSP-TS201, es necesario conocer previamente como se realiza la comunicación entre los procesadores A y B de una misma tarjeta ADSP-TS201; ya que el criterio de su implementación es similar al que buscamos para la comunicación entre tarjetas. Es decir, durante este punto del capítulo se va a analizar la rutina para la comunicación entre ambos procesadores y los cambios que se deben realizar dentro de ésta para obtener una comunicación entre dos tarjetas ADSP-TS201.

6.2.1 Explicación de la rutina de comunicación

Para realizar la comunicación entre procesadores o tarjetas es necesario y recomendable crear un proyecto VDK en VisualDSP++ 5.0; ya que, este proyecto facilita la programación de la rutina de transferencia de datos.

Además, es necesaria la creación de dos proyectos, uno para el transmisor y otro para el receptor. Los archivos utilizados dentro de estos dos proyectos son los mismos, la diferencia radica en que cada proyecto incorpora un archivo principal para controlar sus respectivas funciones. El transmisor incorpora *Producer.cpp* y el receptor *Consumer.cpp*.

CREACIÓN DEL PROYECTO VDK PARA LA PRESENTE APLICACIÓN

Cuando se crea un proyecto VDK los archivos base del proyecto son:

- En la carpeta de archivos *Source*: ExceptionHandler-TS201.asm
- En la carpeta de archivo *Linker* : VDK-TS201.ldf
- En la carpeta de archivos *Kernel*: VDK.cpp, VDK.h, nombre_pryct.vdk

❖ **Archivo "VDK.h"**

Este archivo contiene directivas del tipo *#define* que sirven para colocar nombres descriptivos a las opciones VDK.

Para poder establecer la comunicación, es necesario habilitar varias opciones VDK que deben ser construidas en el proyecto. Las opciones se encuentran definidas en el archivo *VDK.h* y por defecto algunas de ellas, que son necesarias para este proyecto, se encuentran desactivadas (comentadas) dentro del código. Las opciones que deben ser activadas son:

- *Semaphores*
- *Memory pools*
- *Devices flags*
- *IO*
- *Messages*
- *Routing threads*
- *Routing nodes*
- *Multiproccesor messaging*
- *Marshalled messages.*
- *Thread ID*
- Puertos del procesador.

❖ **Archivo "VDK.cpp"**

En el archivo *VDK.cpp* se realiza la declaración de las variables que serán utilizadas por su respectiva opción VDK, algunas opciones VDK utilizan estructuras para la definición de las variables.

❖ **Archivo "Link2_ISR.asm"**

El archivo *Link2_ISR.asm* contiene instrucciones que sirven para etiquetar las interrupciones DMA de los canales 6 y 12 que son utilizados por el puerto de enlace para permitir la comunicación entre los procesadores A y B de una misma tarjeta ADSP-TS201.

❖ **Archivo "Link3_ISR.asm"**

El archivo *Link3_ISR.asm* contiene instrucciones que sirven para etiquetar las interrupciones DMA de los canales 7 y 11 que son utilizados por el puerto de enlace para permitir la comunicación entre tarjetas ADSP-TS201.

❖ **Archivo "LinkPort.cpp"**

En el archivo *LinkPort.cpp* se inicializa el puerto de enlace, se crean funciones para la verificación y control de acceso al puerto de enlace y registros DMA. En las funciones son utilizados los registros LRSTATX/LTSTATX que controlan el estado de Rx/Tx con su respectivo puerto de enlace. También son utilizado los registros LBUFRX/LBUFTRX que controlan los *buffers* de Rx/Tx con su respectivo puerto de enlace.

Se realiza la habilitación la interrupción DMA, para lo cual se utiliza dos canales DMA para la que la comunicación sea *full duplex*, un canal será para transmitir y otro para recibir. Los canales DMA utilizados son asociados con el puerto de enlace que va hacer utilizado para la comunicación en este caso es el *Link3*.

```
#include "LinkPort.h"
```

Se encuentran la declaración de las funciones que controlan el puerto de enlace para la comunicación (Init, Active, Open, Close, SyncRead, SyncWrite, IOCtl).

```
#pragma file_attr("OS_Component=DeviceDrivers")
#pragma file_attr("DeviceDrivers")
```

Pragma direcciona el compilador para emitir los atributos especificados cuando este compila un archivo que contiene *pragma*.

```
#ifdef VDK_INCLUDE_IO_
#define READ_MODE          (1 << 0)
#define WRITE_MODE         (1 << 1)
```

Definición de banderas para el modo de lectura/escritura

```
#ifdef __TS_BYTE_ADDRESS
#define WORD_SHIFT 2
#else
#define WORD_SHIFT 0
#endif
```

Parametriza el desplazamiento de palabras para la compilación de ambos tipos de direccionamiento bytes y palabras.

```
inline int lrstat_read (int num)
{
        int result;
        switch (num)
        {
        default:
        case 0:asm("J0 = LRSTAT0;; J1 = LRSTATC0;; %0 = J0;;"
                        : "=j" (result) : : "J0", "J1" );
                break;
        case 1:asm("J0 = LRSTAT1;; J1 = LRSTATC1;; %0 = J0;;"
                        : "=j" (result) : : "J0", "J1" );
                break;
        case 2:asm("J0 = LRSTAT2;; J1 = LRSTATC2;; %0 = J0;;"
                        : "=j" (result) : : "J0", "J1" );
                break;
        case 3:asm("J0 = LRSTAT3;; J1 = LRSTATC3;; %0 = J0;;"
```

```
                    : "=j" (result) : : "J0", "J1" );
            break;
        }
        return result;
}
```

La función *lrstat_read* toma la dirección de uno de los registros LRSTATX dependiendo del valor de *Num*, este registro verifica el estado de Recepción de su respectivo puerto de enlace.

```
inline int ltstat_read (int num)
{
        int result;
        switch (num)
        { default:
        case 0:asm("J0 = LTSTAT0;; J1 = LTSTATC0;; %0 = J0;;"
                    : "=j" (result) : : "J0", "J1" );
            break;
        case 1:asm("J0 = LTSTAT1;; J1 = LTSTATC1;; %0 = J0;;"
                    : "=j" (result) : : "J0", "J1" );
            break;
        case 2:asm("J0 = LTSTAT2;; J1 = LTSTATC2;; %0 = J0;;"
                    : "=j" (result) : : "J0", "J1" );
            break;
        case 3:asm("J0 = LTSTAT3;; J1 = LTSTATC3;; %0 = J0;;"
                    : "=j" (result) : : "J0", "J1" );
            break;
        }
        return result; }
```

La función *ltstat_read* toma la dirección de uno de los registros LTSTATX dependiendo del valor de *Num*, este registro verifica el estado de transmisión de su respectivo puerto de enlace.

```
inline __builtin_quad lbufrx_read (int num)
```

```
{
        switch (num)
        {
        default:
        case 0:return __builtin_sysreg_read4(__LBUFRX0);
        case 1:return __builtin_sysreg_read4(__LBUFRX1);
        case 2:return __builtin_sysreg_read4(__LBUFRX2);
        case 3:return __builtin_sysreg_read4(__LBUFRX3);
        }
}
```

La función *builtin_quad lbufrx_read* permite leer el registro LBUFRX el cuál verifica el estado del *buffer* de recepción para su correspondiente puerto de enlace dependiendo del valor de *Num*. El registro LBUFRX es solo de lectura.

```
inline void lbuftx_write (int num, __builtin_quad val)
{       switch (num)
        {
        default:
        case 0:__builtin_sysreg_write4(__LBUFTX0, val);
                break;
        case 1:__builtin_sysreg_write4(__LBUFTX1, val);
                break;
        case 2:__builtin_sysreg_write4(__LBUFTX2, val);
                break;
        case 3:__builtin_sysreg_write4(__LBUFTX3, val);
                break;
        }
}
```

La función *lbuftx_write* configura el registro LBUFTX con el valor que contenga la variable *val,* el registro LBUFTX verifica el estado de los *buffers* de transmisión.

```
inline void lrctl_write(int num, int val)
{
```

```
        switch (num)
        {
        default:
        case 0:__builtin_sysreg_write(__LRCTL0, val);
                break;
        case 1:__builtin_sysreg_write(__LRCTL1, val);
                break;
        case 2:__builtin_sysreg_write(__LRCTL2, val);
                break;
        case 3:__builtin_sysreg_write(__LRCTL3, val);
                break;
        }
}
```

La función *lrctl_write* establece las características de recepción mediante el valor de la variable *val* que es escrito en el registro LRCTLX, El registro configura LRCTLX la habilitación de recepción así como el tamaño de los datos que van a ser recibidos.

```
inline void ltctl_write(int num, int val)
{
        switch (num)
        {
        default:
        case 0:__builtin_sysreg_write(__LTCTL0, val);
                break;
        case 1:__builtin_sysreg_write(__LTCTL1, val);
                break;
        case 2:__builtin_sysreg_write(__LTCTL2, val);
                break;
        case 3:__builtin_sysreg_write(__LTCTL3, val);
                break;
        }
}
```

La función *ltctl_write* establece las características de transmisión mediante el valor de la variable *val* que es escrito en el registro LTCTLX, El registro LRCTLX configura la habilitación de transmisión, así como el tamaño de los datos que van a ser transmitidos y establece la velocidad de transferencia.

```
inline void dc4_write(int num, __builtin_quad val)
{
        switch (num)
        {
        default:
        case 0:__builtin_sysreg_write4(__DC4, val);
                break;
        case 1:__builtin_sysreg_write4(__DC5, val);
                break;
        case 2:__builtin_sysreg_write4(__DC6, val);
                break;
        case 3:__builtin_sysreg_write4(__DC7, val);
                break;
        }
}
```

La función *dc4_write* configura el canal DMA con su respectivo registro TCB transmisor del puerto de enlace mediante el valor de la variable *val*.

```
inline void dc8_write(int num, __builtin_quad val)
{
        switch (num)
        {
        default:
        case 0:__builtin_sysreg_write4(__DC8, val);
                break;
        case 1:__builtin_sysreg_write4(__DC9, val);
```

```
            break;
        case 2:__builtin_sysreg_write4(__DC10, val);
            break;
        case 3:__builtin_sysreg_write4(__DC11, val);
            break;
        }
}
```

La función *dc8_write* configura el canal DMA con su respectivo registro TCB receptor del puerto de enlace mediante el valor de la variable *val*.

```
void*
LinkPort::DispatchFunction(VDK::DispatchIDinCode,VDK::DispatchUnion &
inUnion)
{
  int   ret_val = 0;
  switch(inCode)
  {
    case VDK::kIO_Init:
      ret_val = Init(inUnion);
      break;
    case VDK::kIO_Activate:
      ret_val = Activate(inUnion);
      break;
    case VDK::kIO_Open:
      ret_val = Open(inUnion);
      break;
    case VDK::kIO_Close:
      ret_val = Close(inUnion);
      break;
    case VDK::kIO_SyncRead:
      ret_val = SyncRead(inUnion);
      break;
    case VDK::kIO_SyncWrite:
      ret_val = SyncWrite(inUnion);
```

```
            break;
        case VDK::kIO_IOCtl:
            ret_val = IOCtl(inUnion);
            break;
        default:
            break;
    }
    return reinterpret_cast<void *>(ret_val);
}
```

La función *DispatchFunction* llama a una región crítica cuando un dispositivo comienza a inicializarse y activarse.

```
int LinkPort::Init (const VDK::DispatchUnion &inUnion)
{
        // Encerado de bits de banderas
        m_readFlags  = 0;
        m_writeFlags = 0;

        // Creación de banderas de dispositivos
        m_DevFlagRead  = VDK::CreateDeviceFlag();
        m_DevFlagWrite = VDK::CreateDeviceFlag();

        //Coloca el número de enlace de arranque de inicialización de objetos de I/O.
        m_linkNo = *((long *)inUnion.Init_t.pInitInfo);
        return 0;
}
```

En la función *Init* se inicializa el Puerto de enlace, se configura y se crea banderas que verificarán las lecturas y escrituras en el puerto de enlace.

```
int LinkPort::Open (const VDK::DispatchUnion &inUnion)
{
        unsigned mode = 0;
        unsigned size = 0;              // Modo de 1 bit de tamaño.
```

```
unsigned speed = LTCTL_TCLKDIV2; // velocidad es ½ de CCLK

// Evalúa un carácter para determinar el modo
char *pStr = inUnion.OpenClose_t.flags;
while (*pStr)
{
        switch (toupper(*pStr))
        {
                default:
                        return -1;      // opción desconocida
                case 'R': case 'r':
                        mode |= READ_MODE;
                        break;
                case 'W': case 'w':
                        mode |= WRITE_MODE;
                        break;
          case 'N': case 'n':
                        size = 0x00000010; // Modo de 4 bits para Tx y Rx
                        break;
          case 'B': case 'b':
                        size = 0;
                        break;
          case '1':
                // velocidad de transmisión CCLK dividido por 1.5.
                        if ('.' == pStr[1] && '5' == pStr[2])
                        {
                                pStr += 2;
                                speed = LTCTL_TCLKDIV1P5;
                        }
           // velocidad de transmisión CCLK dividido por 1.
                        else
                                speed = LTCTL_TCLKDIV1;
                        break;
          // velocidad de transmisión CCLK dividido por 2.
```

```
            case '2':
                        speed = LTCTL_TCLKDIV2;
                        break;
          // velocidad de transmisión CCLK dividido por 4.
            case '4':
                        speed = LTCTL_TCLKDIV4;
                        break;
                  case ' ': case '\t': case ',':
                        break;
        }
        ++pStr;
  }
  unsigned currentMode = m_readFlags | m_writeFlags;
```

La función *Open* analiza y valida el Modo *Open*, además habilita el *hardware* del Puerto de enlace.

```
  if (mode & currentMode & (READ_MODE | WRITE_MODE))
        return -1;                    // Se abre en este modo
  if (0 == currentMode)
  {
  }
```

Este controlador soporta *"split-open"*, para que el Puerto sea abierto para lectura y escritura.

```
  *inUnion.OpenClose_t.dataH = reinterpret_cast<void*>(mode);

  // Lee las especificaciones iniciales
  if (mode & READ_MODE)
  {
        // Habilita el Puerto de enlace para lectura
        lrctl_write(m_linkNo, LRCTL_REN | size);
        m_readFlags = mode;
```

```
        }

        // Escribe las especificaciones iniciales
        if (mode & WRITE_MODE)
        {
                // Habilita la escritura en el Puerto de enlace
                ltctl_write(m_linkNo, LTCTL_TEN | size | speed);
                m_writeFlags = mode;
        }
        return 0;
}
```

Almacena el modo en los datos del descriptor, cuando el puerto es accedido mediante el descriptor se puede conocer el modo en el que fue abierto.

```
int LinkPort::Close (const VDK::DispatchUnion &inUnion)
{
Unsigned mode = reinterpret_cast <unsigned>(*inUnion.OpenClose_t.dataH);

        if (mode != (mode & (m_readFlags | m_writeFlags)))
                return -1;

        if (mode & READ_MODE)
        {
        // Deshabilita la lectura en el Puerto de enlace
                m_readFlags = 0;
        }

        if (mode & WRITE_MODE)
        {
        // Deshabilita la escritura en el Puerto de enlace
                m_writeFlags = 0;
        }
```

```
        // Borrar los datos del descriptor
        *inUnion.OpenClose_t.dataH = 0;

        return 0;
}
```

La función *Close* borra el estado de las banderas de lectura y escritura dependiendo del modo en el que se encuentre el puerto de enlace.

```
int LinkPort::SyncRead(const VDK::DispatchUnion &inUnion)
{
// Coloca el modo OPEN desde el descriptor de datos

Unsigned mode = reinterpret_cast<unsigned> (*inUnion.ReadWrite_t.dataH);

        if (0 == (mode & READ_MODE))
                return -1;

// Almacena los argumentos de transferencia en estas variables
   m_readDataSize = inUnion.ReadWrite_t.dataSize;
   m_pReadData  = reinterpret_cast<unsigned long
*>(inUnion.ReadWrite_t.data);

//Se ejecuta si lee una palabra cuádruple y el buffer de recepción esta lleno
        if (sizeof(__builtin_quad) == m_readDataSize &&
                0 != (lrstat_read(m_linkNo) & 0x1))
        {
// Se lee la palabra directamente desde el Puerto de enlace
__builtin_quad *pQData = reinterpret_cast<__builtin_quad *>
                                (inUnion.ReadWrite_t.data);

  __builtin_quad tmp = lbufrx_read(m_linkNo);
```

```
memcpy(pQData, &tmp, sizeof(__builtin_quad));   //*pQData = tmp;

            return sizeof(__builtin_quad);
        }
```

La función *SyncRead* lee directamente desde el *buffer* del Puerto de enlace de Recepción si suceden dos condiciones, primero si el tamaño de la lectura es igual a una palabra cuádruple y segundo si el *buffer* de recepción se encuentra lleno. De otra manera establece una transferencia DMA y espera a que el bloque sea completado.

```
// Establece la recepción DMA para la transferencia

   __builtin_quad quadWord;

   static union
   {
        unsigned long vWord[4];
   } dma;

   dma.vWord[0] = reinterpret_cast<unsigned long>(m_pReadData);
   dma.vWord[1] = (m_readDataSize << (16 - WORD_SHIFT)) | 4;
   dma.vWord[2] = 0;
   dma.vWord[3] = TCB_INTMEM | TCB_QUAD | TCB_INT |
                  TCB_DMAR;

// Obtiene el valor de TCB dentro de una palabra cuádruple alineada
   memcpy (&quadWord, &dma, sizeof(dma));

   VDK::LogHistoryEvent((VDK::HistoryEnum)(VDK::kUserEvent+1),
                        m_readDataSize);

        VDK::PushCriticalRegion();
```

```
// Escribe una palabra cuádruple TCB en el DMA
        dc8_write(m_linkNo, quadWord);

 // Espera mientras DMA sea completado
   VDK::PendDeviceFlag (m_DevFlagRead, inUnion.ReadWrite_t.timeout);

        return inUnion.ReadWrite_t.dataSize;
}
```

La función *SyncWrite* escribe directamente en el *buffer* del Puerto de enlace de transmisión si suceden dos condiciones, primero si el tamaño de la escritura es igual a una palabra cuádruple y segundo si el *buffer* de transmisión se encuentra vacío. De otra manera establece una transferencia DMA y espera a que el bloque sea completado.

```
int LinkPort::SyncWrite(const VDK::DispatchUnion &inUnion)
{

// Coloca el modo OPEN desde el descriptor de datos
unsigned mode = reinterpret_cast<unsigned>(*inUnion.ReadWrite_t.dataH);

        if (0 == (mode & WRITE_MODE))
                return -1;

  // Almacena los argumentos de transferencia en estas variables
   m_writeDataSize = inUnion.ReadWrite_t.dataSize;
   m_pWriteData=reinterpret_cast<unsigned long *>(inUnion.ReadWrite_t.data);

  //Se ejecuta si se escribe una palabra cuádruple y el buffer de transmisión esta vació
        if (sizeof(__builtin_quad) == m_writeDataSize &&
                (ltstat_read(m_linkNo) & LTSTAT_TVACANT))
```

```
        {

// Escribe la palabra directamente en el Puerto de enlace
        __builtin_quad *pQData =
                reinterpret_cast<__builtin_quad*>(inUnion.ReadWrite_t.data);
                lbuftx_write(m_linkNo, *pQData);
                return sizeof(__builtin_quad);
        }


// Establece la transmisión DMA para la transferencia
   __builtin_quad quadWord;
           static union
   {
        unsigned long vWord[4];
   } dma;


   dma.vWord[0] = reinterpret_cast<unsigned long>(m_pWriteData);
   dma.vWord[1] = (m_writeDataSize << (16 - WORD_SHIFT)) | 4;
   dma.vWord[2] = 0;
   dma.vWord[3] = TCB_INTMEM | TCB_QUAD | TCB_INT | TCB_DMAR;

// Coloca el valor de TCB en una palabra cuádruple alineada
   memcpy (&quadWord, &dma, sizeof(dma));

   VDK::LogHistoryEvent(VDK::kUserEvent, m_writeDataSize);

        VDK::PushCriticalRegion();

// Escribe una palabra cuádruple TCB en el DMA engine

        dc4_write(m_linkNo, quadWord);
```

```
  // Espera mientras DMA sea completado

   VDK::PendDeviceFlag (m_DevFlagWrite, inUnion.ReadWrite_t.timeout);
        return inUnion.ReadWrite_t.dataSize;
}
```

La función *Activate* activa el controlador del Puerto de enlace, realiza un llamado a un segundo nivel *Kernel* seguido de una activación ISR. También verifica el estado DMA y el estado posterior de las banderas que indican si un DMA ha sido completado.

```
int LinkPort::Activate (const VDK::DispatchUnion &inUnion)
{
        // Lee el estado de los registros DMA

        union
        {
                unsigned long long dblWord;
                unsigned long vWord[2];
        } dstat;
        dstat.dblWord = __builtin_sysreg_read2(__DSTAT);

// Extrae los campos de estado de recepción y transmisión de los registros de estado
        int rxStat = (dstat.vWord[1] >> (m_linkNo * 3))  & 0x7;
        int txStat = (dstat.vWord[0] >> (m_linkNo * 3 + 12)) & 0x7;
        int lrStat = lrstat_read(m_linkNo);
        int ltStat = ltstat_read(m_linkNo);

// Si esta transmitiendo y la Tx DMA esta completa
        if (DSTAT_DONE == txStat)
        {
                // la bandera indica que esta escribiendo
                VDK::PostDeviceFlag(m_DevFlagWrite);
```

```
        }

// Si esta recibiendo y la Rx DMA esta completa
        if (DSTAT_DONE == rxStat)
        {
                // la bandera indica que esta leyendo
                VDK::PostDeviceFlag(m_DevFlagRead);
        }
        return lrStat + ltStat;  // mantiene el compilador para evitar lecturas en sysreg
}

#endif /* VDK_INCLUDE_IO_ */
```

ARCHIVO PRINCIPAL DEL TRANSMISOR

❖ **Archivo "Producer.cpp"**

El archivo *Producer.cpp* se encarga de generar los mensajes que van hacer transmitidos, en este archivo se encuentra la función principal que controla el proceso de la transmisión.

```
#include "Producer.h"
#include <new>
#include <stdlib.h>
#include <stdio.h>
#include "../ProdCons.h"

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")

#define MSG_LIMITP 10   // Número total de mensajes a transmitir
int datos_transmitidos[MSG_LIMITP];
```

```
//Función local que genera el ítem de salida
static void produce_item(void **ppItem)
{
        int val = rand();
        datos_transmitidos[w++] = val;
        *ppItem = reinterpret_cast<void*>(val);
}
```

La función *Producer::Run* es la función principal del programa que genera el mensaje para ser transmitido. Los mensajes transmitidos son números randómicos enteros que a su vez son guardados en el vector global *datos_transmitidos* para su posterior comparación y comprobación una vez que el receptor haya recibido los datos.

```
void
Producer::Run()
{
   bool produceMessages = true;
   while (produceMessages)
   {
                // Genera un item para poner en el mensaje
                void *item;
                produce_item(&item);

                // Espera por un canal para colocar el mensaje
VDK::MessageID msg = VDK::PendMessage(MSG_CHANNEL, 0);

                // Averigua de donde viene el mensaje
                VDK::MsgChannel channel;
                VDK::ThreadID sender;
                VDK::GetMessageReceiveInfo(msg, &channel, &sender);
                // Obtiene la parte valida del mensaje
                int type;
                unsigned int size;
```

```
            void *mem_block;
            VDK::GetMessagePayload(msg, &type, &size, &mem_block);

            if(type != END_MSG)
            {
        // Resetea el espacio de memoria y copia el siguiente mensaje dentro de este
        // espacio de memoria
              memset(mem_block, 0, MBSIZE);
              memcpy(mem_block, &item, sizeof(int));

                  // Configura y envía el nuevo mensaje
            VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE,
mem_block);
            VDK::PostMessage(sender, msg, MSG_CHANNEL);
          }
          else
          produceMessages = false;
   }

   printf("Producer Finished\n");
   exit(0);
}
```

Es el controlador de error del productor

```
int
Producer::ErrorHandler()
{
   /* Por defecto ErrorHandler mata la cadena cuando existe un error */
   return (VDK::Thread::ErrorHandler());
}
```

PROYECTO PARA EL RECEPTOR

Para la recepción se crea un proyecto y se añaden los archivos descritos en el desarrollo de la rutina de transmisión, estos archivos añadidos son:

- *Link2_ISR*
- *Link3_ISR*
- *LinkPort.cpp*

El archivo *Consumer.cpp* controla la recepción, cuando son recibidos los mensajes desde el transmisor imprime un mensaje confirmando la recepción del mensaje y cuando no recibe el mensaje se imprime un mensaje de error.

Debido a que el proyecto de recepción contiene varios de los archivos ya analizados en la proyecto de transmisión, se explicará únicamente los archivos e instrucciones que no fueron revisadas anteriormente.

❖ **Archivo "Consumer.cpp"**

```
#include "../ProdCons.h"
```

El archivo de cabecera *ProdCons.h* contiene la declaración de constantes las cuales son requeridas en ambas clases de cadenas.

```
#define MSG_LIMIT 10     // Número total de los mensajes que serán recibidos

// Función local que manipula el mensaje entrante
static void consume_item (void *item)
{
        static int expected = 1;
        int *current = reinterpret_cast<int*>(item);

        // Si el ítem del mensaje es recibido
```

```
        if (*current == expected)
        {
                printf("Consumer ID %d Received %d from Producer\n",
                VDK::GetThreadID(), *current);
        }
        // Si el mensaje no es recibido
        else
        {
                printf("Consumer Error: Expecting %d Received %d\n",expected,
*current);
        }

        expected++;
        VDK::Yield();
}
```

La función Consumer::Run() es la función principal del programa de recepción

```
void
Consumer::Run()
{
        // Envía VDK_kMaxNumActiveMessages vacíos
        for (int i = 0; i < VDK_kMaxNumActiveMessages; ++i)
        {
                // Ubica he inicializa los espacios de memoria
                void*mem_block=(void*)
                VDK::MallocBlock(kMarshalledMessages);
                strcpy((char *)mem_block,"empty block");

        // Crea el mensaje, establece la parte valida del mensaje y envía el mensaje
        VDK::MessageID msg = VDK::CreateMessage(MSG_TYPE, 0, NULL);
        VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE, mem_block);
        VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
        }
```

```
    for(int i = 0; i < MSG_LIMIT; i++)
    {
        // Capta los mensajes que contiene un ítem
        VDK::MessageID msg = VDK::PendMessage(MSG_CHANNEL, 0);

        // Extrae el ítem desde el mensaje
        int type;
        unsigned int size;
        void *item;
        VDK::GetMessagePayload(msg, &type, &size, &item);

        // Realiza cualquier operación con el ítem
        consume_item(item);


        if (i < MSG_LIMIT-1) // Si no se alcanza el limite del mensaje
        {
        // Reenvía mensaje para su reutilización
        strcpy((char *)item,"empty block");
        VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE, item);
        VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
        }
        else    // Si se alcanza el limite del mensaje
    {
    // Establece la parte válida del mensaje para mostrar que este es el último mensaje
                VDK::SetMessagePayload(msg, END_MSG, 0, 0);
    // Envía un mensaje al Producer para indicar el fin del ejemplo
                VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
        }
    }

    printf("Consumer Finished\n");
    exit(0);
}
```

6.2.2 Ejecución del programa de comunicación entre dos tarjetas ADSP-TS201

Una vez cargados los proyectos en sus respectivos computadores, se debe modificar los siguientes parámetros en la pestaña de *Kernel* de cada uno de los proyectos. En la herramienta *Messages* → *Routing Threads* → *InRouter(OutRouter)* → *I/O Objects*, seleccionar *Link3* como se muestra en la Figura 6.6.

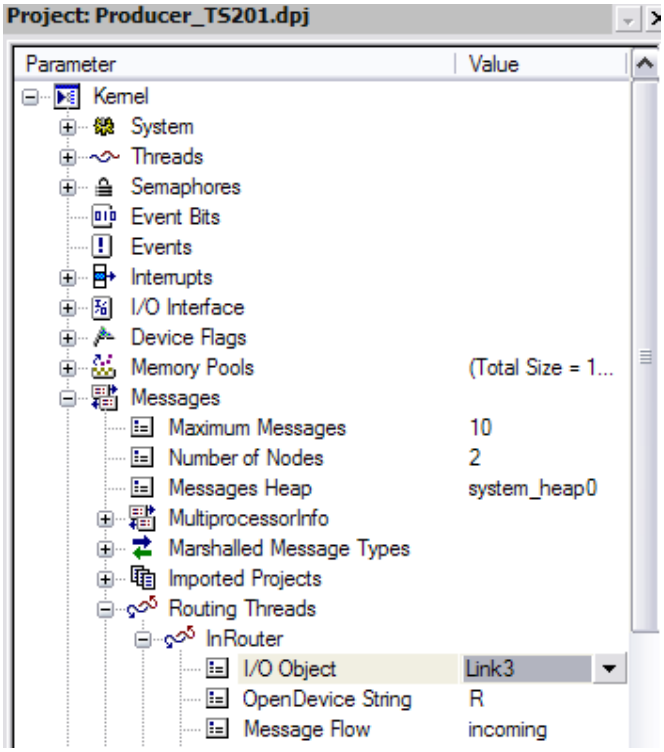

Figura. 6.6. Modificación I/O de puerto enlace

Una vez compilados y ejecutados simultáneamente los programas en los correspondientes computadores, en la ventana de salida del Receptor se imprimirán los datos recibidos que sirven como confirmación para el transmisor de que se recibió satisfactoriamente los mensajes enviados. La ventana de salida en el receptor se muestra en la Figura 6.7.

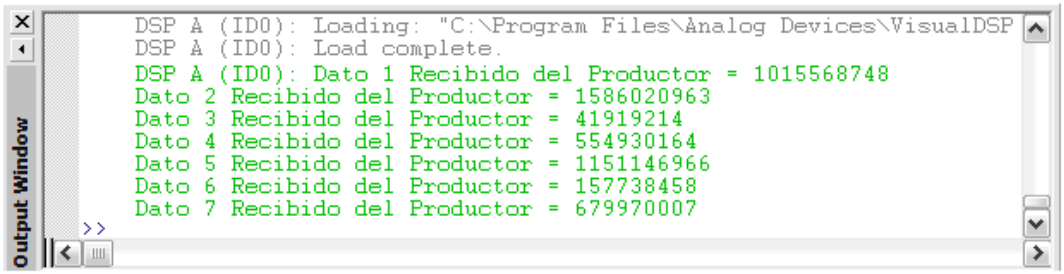

Figura. 6.7. Ventana de salida en el receptor

Cuando se recibe la cantidad total de mensajes que fueron configurados en el receptor, el receptor indica mediante un mensaje que la recepción ha finalizado como se muestra en la Figura 6.8.

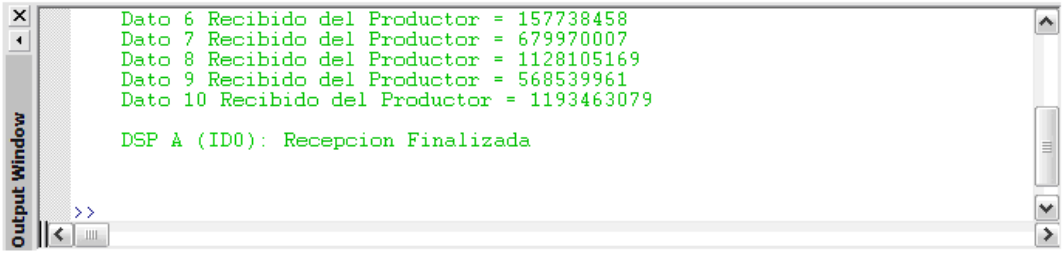

Figura. 6.8. Ventana de salida en el transmisor

Una vez que se envían todos los mensajes desde el transmisor, El transmisor mediante un mensaje en la ventana de salida indica que el proceso de transmisión finalizó.

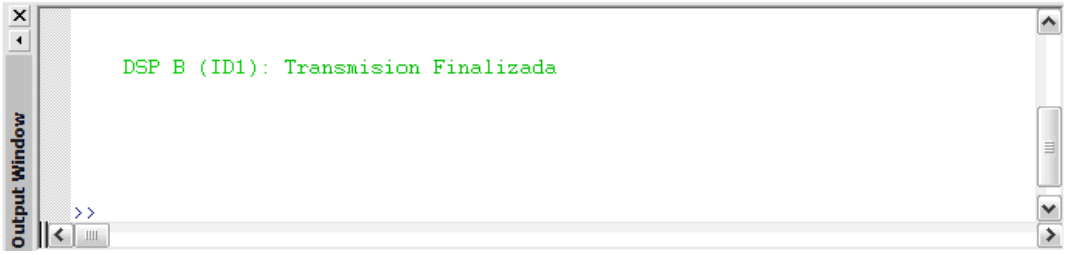

Figura. 6. 9. Mensaje de finalización de comunicación en el receptor

6.2.3 Ejecución del programa de comunicación entre procesadores A y B de una misma tarjeta ADSP-TS201

Para la comunicación entre procesadores se debe seguir los siguientes pasos:

1. Se debe abrir ambos proyectos en forma simultánea. Para este efecto se debe abrir el proyecto en forma de *"grupo"* como se indica en la Figura 6.10.

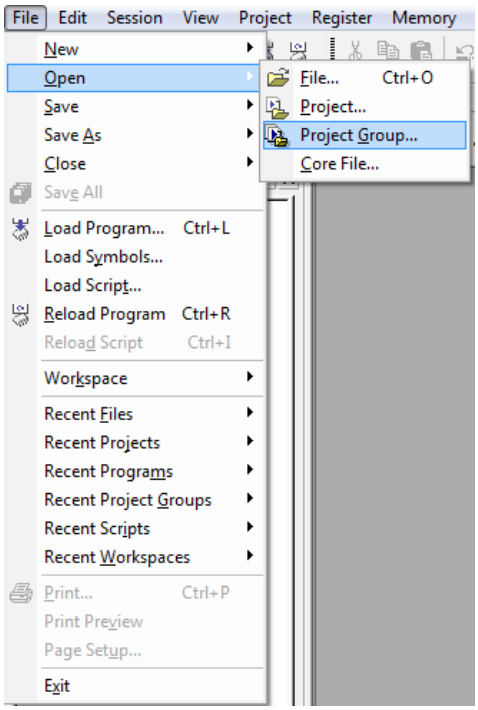

Figura. 6.10. Carga de proyectos en forma de grupo

2. En la ventana que aparecerá se debe escoger y cargar el proyecto grupal deseado.

3. Una vez cargado el proyecto grupal, se debe modificar los siguientes parámetros en la pestaña de Kernel de cada uno de los proyectos. En la herramienta *Messages* → *Routing Threads* → *InRouter(OutRouter)* → *I/O Objects*, seleccionar *Link2* como se muestra en la Figura 6.11.

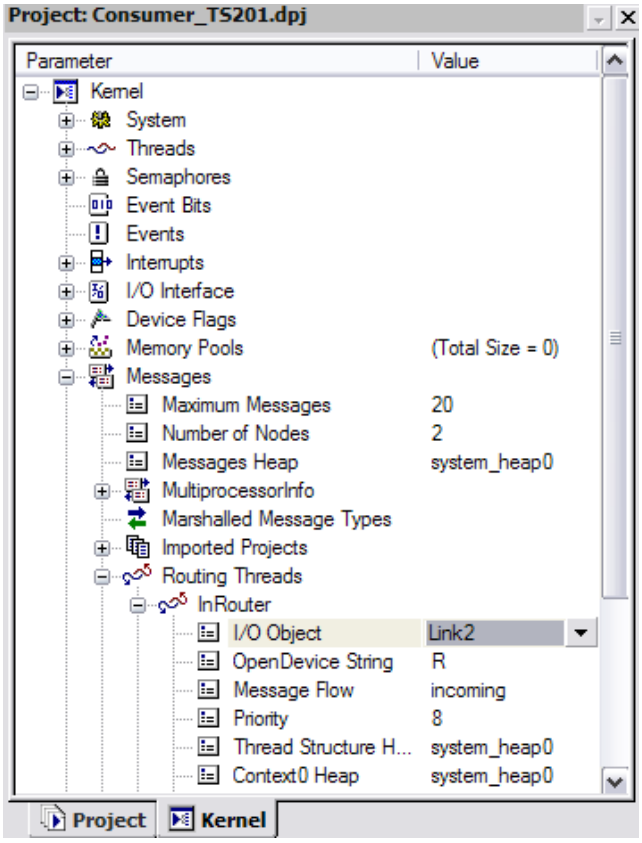

Figura. 6.11. Selección de I/O del puerto de enlace (Link 2)

4. Se debe compilar el programa, cargando los programas respectivos de cada etapa (transmisor y receptor) uno en cada procesador de la tarjeta de la siguiente manera:

5. Una vez compilado el proyecto grupal se debe correr simultáneamente los programas en los correspondientes procesadores. Para este efecto se debe ejecutar el proyecto en forma de multiprocesamiento como se indica en la figura:

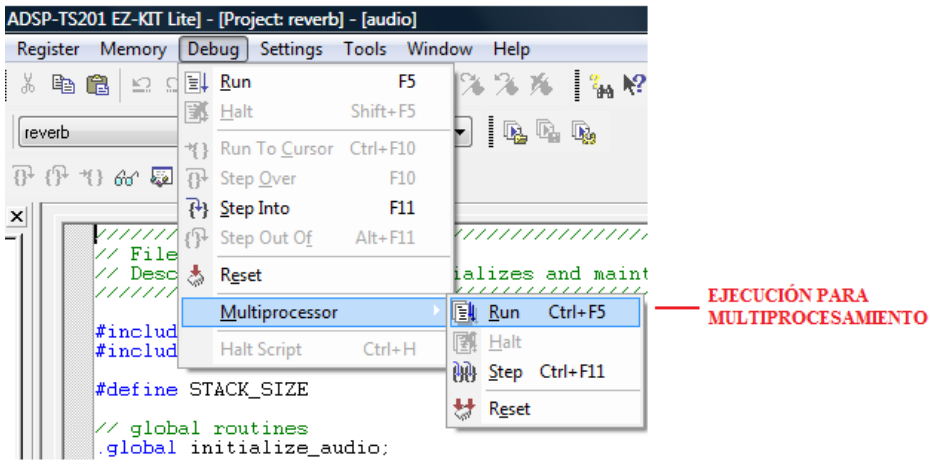

Figura 6. 12 Ventana para ejecución de multiprocesamiento

6. En la ventana de salida se imprimirán los datos recibidos por el procesador A que sirven como confirmación para el transmisor (Procesador B) de que se recibió satisfactoriamente los mensajes enviados. Además, una vez finalizada la transmisión, en la misma ventana, se imprimirán adicionalmente dos mensajes de confirmación. El primer mensaje lo imprime el procesador B para indicar que la ha transmisión ha finalizado y el segundo mensaje lo imprime el procesador A indicando que la recepción ha finalizado.

La ventana de salida se muestra en la Figura 6.13.

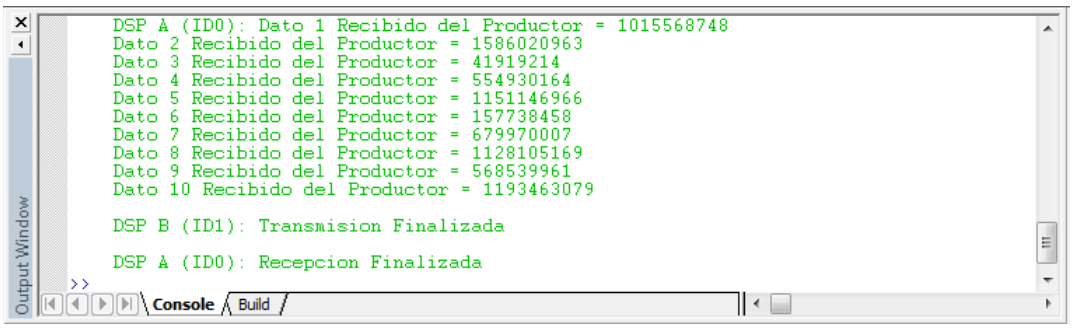

Figura. 6. 13. Ventana de salida del receptor en comunicación entre procesadores A y B

CAPITULO VII

DESARROLLO DE PRÁCTICAS DE PROCESAMIENTO DIGITAL DE SEÑALES ADSP-TS201

7.1 TRANSFORMADA DISCRETA DE FOURIER (DFT)

Para la realización de esta práctica, en base a la tarjeta EZ-KIT Lite ADSP-TS201s, se diseñó previamente un código en lenguaje C y ensamblador para la implementación de un programa que calcule la transformada FFT de raíz 2. Este código implementado deberá ser usado durante toda la práctica para realizar los procedimientos de todos cálculos de FFT solicitados.

7.1.1 Fundamento teórico

Una de las herramientas más útiles para el análisis y diseño de sistemas LTI (lineales e invariantes en el tiempo), es la transformada de Fourier. Esta representación de señales implica la descomposición de las mismas en términos de componentes sinusoidales o exponentes complejas. Con esta descomposición, se dice que una señal está representada en el dominio de la frecuencia.

La representación en series de Fourier de una señal periódica en tiempo continuo puede tener infinitas componentes en frecuencia; mientras que, la representación en series de Fourier de una señal periódica en tiempo discreto contendrá un máximo de N componentes en frecuencia. Sin embargo, para calcular el espectro de señales en el tiempo continuo así como discreto, se requieren los valores de la señal para cada instante de tiempo, lo cual es en realidad imposible dado que nuestra señal es finita y el tiempo no, pero con la DFT o Transformada Discreta de Fourier se puede lograr una excelente aproximación.

La DFT es un algoritmo fundamental en procesamiento digital de señales y es usada en muchas aplicaciones, incluyendo análisis en frecuencia y procesamiento en el dominio de la frecuencia.

La DFT se asemeja a la operación de correlación en que ésta mide la similitud entre una señal desconocida y un exponencial complejo. El espectro resultante lleva la información compleja (amplitud y fase) para N frecuencias.

Una DFT de N puntos calcula una secuencia X[k] de N números complejos dada otra secuencia de datos x[n] de longitud N de acuerdo con la fórmula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad k = 0,1,2,\ldots, N-1$$

$$W_N = e^{-j2\pi/N}$$

El cálculo de cada punto de la DFT implica N multiplicaciones complejas y (N-1) sumas complejas; por lo tanto, los *N* puntos de la DFT pueden obtenerse tras N^2 multiplicaciones complejas y N(N-1) sumas complejas. El elevado número de cálculos hace a la DFT ineficiente debido a que no explota las propiedades de simetría y periodicidad del factor de fase W_N.

Debido a estos requerimientos computacionales, el algoritmo de la DFT no se utiliza usualmente en procesamiento de señales en tiempo real.

Para este efecto se utiliza el algoritmo FFT (Transformada Rápida de Fourier), el cual provee un algoritmo eficiente para implementar la DFT.

El algoritmo para la FFT explota las propiedades de simetría y periodicidad de la exponencial compleja discreta en el tiempo para reducir significativamente el número de cálculos que requiere la DFT. En una implementación de FFT de las componentes real e imaginario de W_N se llaman frecuentemente factores *"twiddle"*. Estas constantes son usualmente precalculadas y almacenadas en una tabla.

La base de la FFT es que la DFT puede ser dividida en DFTs más pequeñas. Una FFT de raíz 2 divide la DFT en dos DFTs más pequeñas, cada una de las cuales se divide en dos DFTs más pequeñas, y así sucesivamente, resultando en una combinación final de DFTs de dos puntos.

Una DFT de N puntos puede ser calculada ejecutando dos DFTs de N/2 puntos y combinando las salidas de las DFTs más pequeñas para dar el mismo resultado que la DFT original. Al dividir la DFT en dos DFTs más pequeñas se reduce el número de cálculos en un 50%. Es decir, si los cálculos de N puntos son divididos en DFTs más pequeñas hasta que solo hayan DFTs de dos puntos, el número total de multiplicaciones y sumas complejas se reduce a $N\log_2 N$.

Para evaluar una transformada discreta de Fourier con N muestras el algoritmo de la FFT encuentra su eficiencia cuando N es una potencia de 2.

Se utilizan dos métodos para dividir las DFTs en cálculos principales más pequeños (de dos o cuatro puntos).

- La FFT de decimación en el tiempo (DIT) que divide la secuencia de entrada (tiempo) en dos grupos: uno de muestras pares y otro de muestras impares.
- La FFT de decimación en frecuencia (DIF) divide la secuencia de salida (frecuencia) en porciones pares e impares.

Cada una de las estructuras elementales que se dan en la implementación del algoritmo FFT (DFT de dos puntos) se denomina *butterfly* o *mariposa*. Las ecuaciones para estas estructuras elementales son:

$$x_0' = x_0 + (Cx_1 + Sy_1)$$
$$x_1' = x_0 - (Cx_1 + Sy_1)$$
$$y_0' = y_0 + (Cy_1 - Sx_1)$$
$$y_1' = x_0 - (Cy_1 - Sx_1)$$

Las variables x_n y y_n representan las partes real e imaginaria, respectivamente, de una muestra de datos, mientras que los coeficientes C y S corresponden al factor *"twiddle"* que corresponde a esa *mariposa*. La Figura 7.1 representa este cálculo de mariposa.

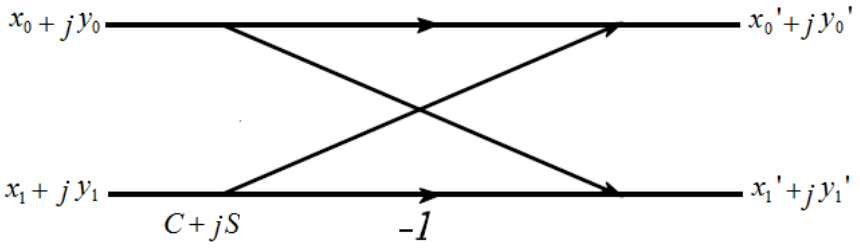

Figura. 7. 1. Calculo mariposa

La FFT de raíz 2 está dividida en estructuras repetidas llamadas etapas como también en mariposas individuales. Existen $\log_2(N)$ etapas en una FFT de raíz 2 de N puntos. Por ejemplo, la FFT de 64 puntos contiene 6 etapas.

Nótese que para la implementación, el arreglo de salida x[k] está en un orden secuencial normal y el arreglo de entrada x[n] está en el orden de *"bit reversed"*. Este orden no secuencial es un resultado de la subdivisión repetida de las secuencias de datos en el algoritmo de la FFT. La inversión de bits es una técnica de direccionamiento usada en los cálculos de la FFT para ordenar los resultados secuencialmente. Una dirección de bits invertidos se genera invirtiendo el orden de los bits en una

representación binaria de las direcciones. La inversión de bits de un arreglo de datos se realiza mediante el intercambio de cada posición en el arreglo con la posición de su correspondiente dirección con los bits invertidos.

7.1.2 IMPLEMENTACIÓN DE LA FFT DE RAÍZ DOS

ESTRUCTURA NO OPTIMIZADA

Esta implementación permite el cálculo de la FFT para los siguientes datos:

- Datos reales con un número de bits (N) de entre 64 y 32768 bits. N debe ser una potencia de 2.
- Datos complejos con un número de bits (N) de entre 32 y 16384 bits. N debe ser una potencia de 2.

En el CD que contiene la información el proyecto de grado se incluye el programa para el cálculo de la FFT de raíz 2.

A continuación se explicará detalladamente el proyecto que implementa una FFT de raíz 2. Este proyecto incluye los siguientes archivos:

- *FFTDef.h*
- *main.c*
- *fft.asm*
- *init.c*
- *variables.asm*

❖ **Archivo "FFTDef.h"**

En este archivo es en donde el número de bits (N) y el tipo de FFT deben ser definidos.

#define N 128 // Tamaño de la FFT

#define FFT_Real

En esta sección es precisamente en donde se define N y el tipo de FFT. N representa el número de bits de entrada de la FFT, ya sea que correspondan a datos reales o complejos. Para definir el tipo de FFT (real o complejo), se debe incluir o comentar la instrucción ***#define FFT_Real*** como se indica en la Figura 7.2. Si se incluye esta instrucción, entonces la FFT será de tipo real, y si se la comenta la FFT será de tipo complejo.

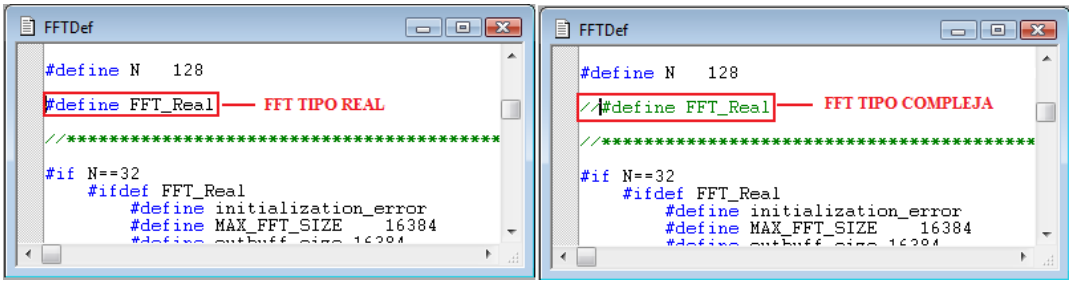

Figura. 7.2. Definición de FFT real o compleja

Para el caso de FFT real, N es válido para valores entre 64 y 32768, siendo N siempre una potencia de 2.

Para el caso de FFT compleja, N es válido para valores entre 32 y16384, siendo N siempre una potencia de 2.

Para todo el resto de casos, un mensaje de error se desplegará en la ventana de salida después de compilar el proyecto.

```
#if N==32
  #ifdef FFT_Real
    #define initialization_error
    #define      MAX_FFT_SIZE    16384
    #define outbuff_size 16384
  #else
    #define      MAX_FFT_SIZE    32
```

```
      #define outbuff_size 64
   #endif
#elif N==64
   #define MAX_FFT_SIZE    64
   #ifdef FFT_Real
      #define outbuff_size 64
   #else
      #define outbuff_size 128
   #endif
.
.
.
#elif N==32768
   #ifdef __ADSPTS201__
      #ifdef FFT_Real
         #define    MAX_FFT_SIZE    32768
         #define outbuff_size 32768
      #else
         #define initialization_error
         #define outbuff_size 8192
         #define    MAX_FFT_SIZE    8192
      #endif
   #else
    #define initialization_error
    #define outbuff_size 16384
    #define        MAX_FFT_SIZE    16384
   #endif
#else
   #define initialization_error
   #define MAX_FFT_SIZE    8192
   #define outbuff_size 8192
#endif
```

Estas instrucciones establecen un lazo condicional que sirve para definir las siguientes variables:

- *MAX_FFT_SIZE* : Tamaño máximo del *buffer* del dato de entrada.
- *outbuff_size* : Tamaño del *buffer* del dato de salida.
- *initialization_error* : Si se ingresa un valor de N no válido, esta variable es definida al compilar el proyecto.

Estas variables son definidas de acuerdo al valor de N y al tipo de FFT fijados anteriormente. Por ejemplo, si el valor de N es 64 y el tipo de FFT es real, estas variables se definen de la siguiente forma:

- *MAX_FFT_SIZE* = 64
- *outbuff_size* = 64
- *initialization_error* no es definido ya que el valor de N es válido.

El tamaño de la constante *outbuff_size* será el doble de N cuando se trabaje con una FFT tipo compleja. Al ingresar un valor no válido de N, la variable *initialization_error* es definida. Esta definición hace que el programa al ser corrido despliegue un mensaje de error, indicando que el valor de N debe ser modificado.

```
#define REAL      0
#define COMPLEX   1
```

Estas variables son definidas para su utilización posterior en la función que realiza el cálculo de la FFT.

❖ **Archivo "*main.c*"**

En este archivo se inicializa la variable de entrada y se llama a la función que ejecuta el cálculo de la FFT.

#include "FFTDef.h"

Mediante esta instrucción el ensamblador incluye este archivo en el momento de la compilación.

extern FFT32(float (*)[], float (*)[], float (*)[], float (*)[], int, int);

Aquí se define como externa la función FFT32; función que se encuentra en el archivo fft.asm.

```
#pragma align 4
section ("data1ab")
float output[outbuff_size];
```

Estas instrucciones definen la variable *output* en donde se guardarán los cálculos de la FFT.

```
#if N==32
 #ifndef FFT_Real
    float input[2*N] = {
            #include "inputs/input64.dat"
            };
 #endif
#elif N==64
 #ifdef FFT_Real
    float input[N] = {
            #include "inputs/input64.dat"
            };
 #else
    float input[2*N] = {
            #include "inputs/input128.dat"
            };
 #endif
.
.
.
#elif N==32768
 #ifdef __ADSPTS201__
    #ifdef FFT_Real
```

```
        float input[N] = {
            #include "inputs/input32768.dat"
            };
    #endif
 #endif
#endif
```

Estas instrucciones establecen un lazo condicional que sirve para inicializar el vector *input* con los datos de entrada. Los datos que se carguen a este vector dependerán del valor que se le dé a N.

```
#ifdef initialization_error
    float input[16384];
#endif
```

Si la constante *initialization_error* fue definida (al dar un valor no válido a N) entonces el vector *input* no es inicializado.

```
volatile int
    i,
    tmp_i0,
    tmp_i1;
```

Se declaran las variables enteras *tmp_i0* y *tmp_i1* como volátiles. Estas variables guardarán el valor de contador de ciclo inicial y contador de ciclo final respectivamente.

```
void main( void )
{
#ifdef initialization_error
    printf("ERROR: Esta funcion FFT trabaja solo en las siguientes
condiciones:\n");
    printf("      -N debe ser una potencia de 2\n");
```

```
printf("     -Para entradas reales,   64<=N<=32768\n");
printf("     -Para entradas complejas, 32<=N<=16384\n");
printf("Cambie la configuracion de N en el archivo FFTDef.h\n");
```

Esta sección explica la parte del código al que accede el programa cuando la constante *initialization_error* fue definida a causa de un ingreso de un valor de N no válido. Como se observa, se imprimirán varios mensaje en la ventana de salida indicando al usuario las condiciones sobre las cuales trabaja esta función FFT.

```
#else
#ifdef __ADSPTS201__
 asm("#include <defts201.h>");
 asm("#include <cache_macros.h>" );
 asm("#include <ini_cache.h>");
 asm("#include <fftdef.h>");
 asm("preload_cache;");
#endif
```

Si la constante *initialization_error* no fue definida, entonces el programa accede a esta sección del programa. Mediante estas instrucciones el ensamblador incluye estos archivos al proyecto en el momento de la compilación. Observe que las instrucciones contienen el comando ***asm("…").*** Este commando permite que las funciones o códigos en lenguaje ensamblador sean reconocidos por el lenguaje C.

```
tmp_i0 = __builtin_sysreg_read( __CCNT0 );
```

Mediante esta instrucción se realiza una lectura del contador de ciclo inicial CCNT0 y se guarda este valor en la variable *tmp_i0*.

```
#ifdef FFT_Real
  FFT32(&(input), &(ping_pong_buffer1), &(ping_pong_buffer2),
 &(output),  N,  REAL);
#else
```

```
FFT32(&(input), &(ping_pong_buffer1), &(ping_pong_buffer2),
&(output), N, COMPLEX);
#endif
```

Esta sección comprende la llamada de la función *FFT32* que es la encargada de realizar los cálculos para la FFT. Obsérvese que si la constante *FFT_Real* está definida, el programa ejecuta la función FFT para datos reales. Por el contrario, si la constante *FFT_Real* no está definida, el programa ejecuta la función FFT para datos complejos. Nótese además que la única diferencia entre ambas funciones radica en el argumento final (REAL o COMPLEJO) que se usa para cada función. Estas constantes REAL y COMPLEJO están definidas en el archivo *FFTDef.h* que se revisó previamente.

```
tmp_i1 = __builtin_sysreg_read( __CCNT0 );
```

Mediante esta instrucción se realiza una lectura del contador de ciclo inicial CCNT0 y se guarda este valor en la variable tmp_i0.

```
printf("cycle count = %d", tmp_i1 - tmp_i0);
```

Finalmente se imprime en la ventana de salida el valor total del contador de ciclo. Este valor será igual al valor del contador final menos el valor del contador inicial.

Este valor final de contador de ciclo corresponde al número de ciclos total que al procesador le tomó para realizar el cálculo de la FFT. Como ejemplo, operando a 600 MHz, el núcleo del procesador ADSP-TS201S ejecuta el cálculo de 256 puntos complejos FFT en 585 ciclos de de reloj que equivalen a 0.975 us.

- **Archivo "fft_flp32_TS201.asm"**

Este archivo contiene la rutina de la función FFT32 que realiza el cálculo de la FFT de raíz 2.

FFT32(&(input), &(ping_pong_buffer1), &(ping_pong_buffer2), &(output), N, TIPO);

I. Descripción de la llamada de la Función

1. Entradas:
   j4 -> input (*ping-pong buffer* 1)
   j5 -> *ping-pong buffer 1*
   j6 -> *ping-pong buffer 2*
   j7 -> output (Salida)
   j27+0x18 -> N = Número de puntos
   j27+0x19 -> TIPO: REAL o COMPLEX

2. Limitaciones:
   a. Todos los *buffers* deben estar alineados sobre espacios de memoria que sean múltiplo de 4.
   b. N deber estar entre 32 y MAX_FFT_SIZE.
   c. Si se requiere optimizar espacio de memoria y la entrada no necesita ser conservada, el *buffer ping_pong_buffer1* puede ser usado como el *buffer* de entrada.
   d. Si se requiere optimizar espacio de memoria el *buffer ping_pong_buffer2* puede ser usado como el buffer de salida.

3. El valor máximo de MAX_FFT_SIZE puede ser escogido a través de #define. Un mayor valor permite tener más opciones de N, pero sus *twiddles* ocuparán más espacio de memoria.

4. Esta función puede procesar hasta 64K bloques de datos debido a que el entorno C necesita memoria para sí mismo. Un bloque de memoria tiene palabras con un tamaño de 128K, entonces el valor máximo de N puede ser de 128K para datos reales o 64K para datos complejos.

II. Descripción del algoritmo FFT COMPLEJO.

1. El dato de entrada es tratado como un complejo de N puntos.
2. Debido al reordenamiento, ninguna etapa puede ser implementada en su lugar.
3. El *"bit-reversed"* y las dos primeras etapas están combinadas en un lazo simple. Este lazo toma los datos de la entrada y los guarda en el *buffer ping-pong buffer1*.
4. Cada etapa subsiguiente realiza un proceso de *ping-pong* con los datos entre los dos buffer *ping-pong*. La última etapa usa el *buffer* de salida FFT para su salida.
5. Aunque la FFT está diseñada para ser llamada con cualquier tamaño de N <= MAX_FFT_SIZE a través del submuestreo de los factores *twiddle*, la mejor optimización de ciclo es lograda cuando MAX_FFT_SIZE=N.

III. Descripción del algoritmo FFT REAL.

1. Los datos de entrada son tratados como N/2 puntos complejos intercalados. La FFT de N/2 de puntos complejos será calculada primero. Por lo tanto, N es reducido a la mitad, ahora el número de puntos es de N/2.

2. Recombinación Real

Aquí la FFT calculada en el paso anterior es recombinada para producir la FFT real de N puntos.

❖ **Archivo *variables.asm***

Este archivo contiene un lazo condicional que sirve para cargar los factores *twiddle*. Los factores que se carguen al vector correspondiente también dependerán del valor que se le dé a N (número de puntos).

ESTRUCTURA OPTIMIZADA

Para estar apto para re-estructurar el algoritmo para alcanzar su optimización sobre el ADSP-TS201, nosotros tenemos que entender porque el rendimiento de extensas FFTs usando la estructura convencional FFT es muy pobre.

La memoria del ADSP-TS201 está optimizada para lecturas secuenciales. La cache está diseñada para ayudar con los algoritmos donde las lecturas no son secuenciales. En el algoritmo convencional FFT, cada *butterfly* de una etapa se realiza dos veces; por lo que, las lecturas no son secuenciales. La solución consiste en re ordenar la salida de una etapa para asegurar que las lecturas de la siguiente etapa sean secuenciales.

La nueva *butterfly* tendrá las operaciones necesarias para realizar una sóla butterfly compleja en lugar de las dos que existían en la estructura convencional. Debido a que el ADSP-TS201 es un procesador SIMD (puede doblar todos los cálculos), las dos butterflies que se encuentren adyacentes son calculadas en paralelo, una en el bloque de cálculo X y la otra en el bloque de cálculo Y.

El desarrollo de la práctica para la DFT se encuentra en el ANEXO 5.

7.2 FILTRO FIR

7.2.1 Fundamento teórico

Un filtro FIR es una combinación lineal de un conjunto finito de entradas. La ecuación para un filtro FIR es:

$$y[n] = \sum_{k=0}^{m-1} a_k x[n-k]$$, donde:

$x[n-k]$ es la historia pasada de las entradas

$y[n]$ es la salida del filtro en el tiempo discreto

a_k es el vector de coeficientes del filtro

El código de un filtro FIR es una implementación por *software* de esta ecuación.

El filtraje de tipo FIR es una convolución en el tiempo. La ecuación de un filtro FIR es similar a la ecuación de convolución:

$$y[n] = \sum_{k=0}^{\infty} h_k x[n-k]$$

Los filtros FIR tienen varias ventajas que los hacen más deseables que los filtros IIR (*Infinite Impulse Response*) para ciertos criterios de diseño:

- Los filtros FIR pueden ser diseñados para que tengan fase lineal. En muchas aplicaciones, la fase es un componente crítico de la salida.

- Los filtros FIR son siempre estables por que únicamente tienen ceros en el plano complejo.

- Los errores de sobreflujo no son problemáticos por que la suma de productos en los filtros FIR se realiza sobre un conjunto finito de datos.
- Los Filtros FIR son fáciles de entender e implementar. Pueden proporcionar soluciones rápidas a problemas de ingeniería.

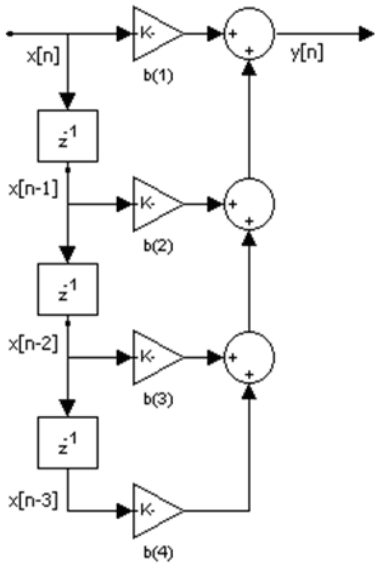

Figura. 7. 3. Estructura de un filtro FIR

7.2.2 IMPLEMENTACIÓN DEL FILTRO FIR

En esta sección se explicara el proceso para realizar la implementación de filtros FIR. La estructura general para los programas que son implementadas para los filtros se muestra en la Figura 7.4.

EXPLICACIÓN DEL PROGRAMA GENERAL DEL FILTRO FIR IMPLEMENTADO

En el CD que contiene la información del Proyecto de Grado se incluye el programa del filtro FIR pasabajos FIR.dpj. A continuación se explicará detalladamente el proyecto que implementa un filtro FIR pasabajos. El proyecto será el mismo para los demás tipos filtros ya que solo cambian los coeficientes.

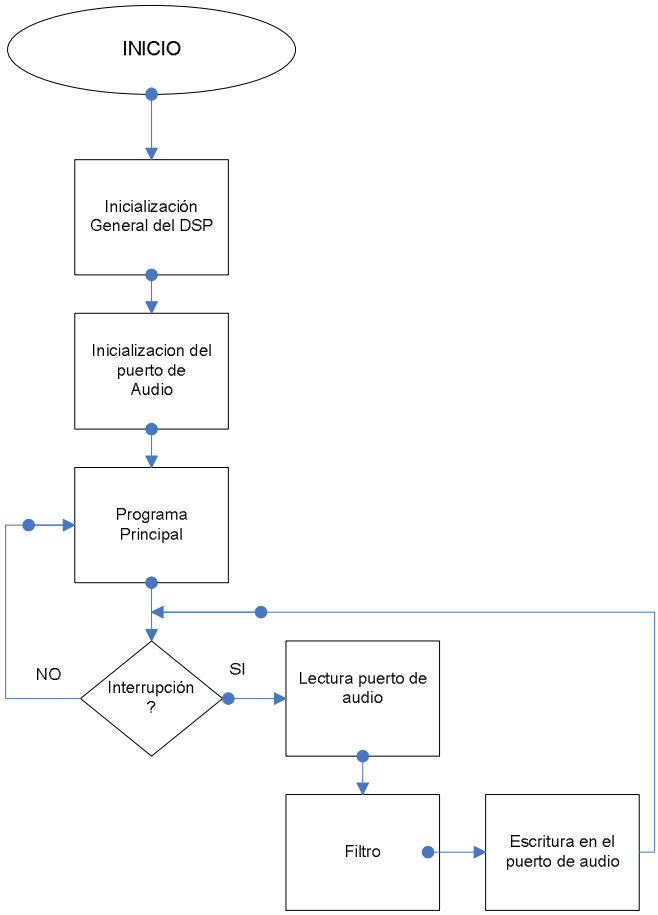

Figura. 7. 4. Estructura general de los programas de los filtros

❖ **PROGRAMA PRINCIPAL (Archivo *"FIR.asm")***

#include <defTS201.h>

El archivo <defTS201.h> se encuentra dentro del directorio *C:\Program Files\Analog Devices\VisualDSP 5.0\TS\include*, y mediante esta instrucción el ensamblador lo incluye en el momento de la compilación. Este archivo contiene directivas de preprocesador del tipo #define, enteramente similares a las del lenguaje C. Básicamente, estas directivas sirven para colocar nombres descriptivos a cada uno de los bits de los registros del DSP.

#include "tabs.h"

El archivo *"tabs.h"* incluye la constante TAPS que contiene el valor del orden del filtro. Si se desea cambiar el orden del filtro en el proyecto, se debe cambiar únicamente el valor de esta constante. Mediante esta instrucción el ensamblador incluye este archivo en el momento de la compilación.

```
.SECTION data1;
.align 4;
.var dline[TAPS];
```

Esta parte del programa define la línea de retardo para las muestras del filtro FIR.

```
.align 4;
.var coefs[TAPS]="coeficientes_filtros.dat";
```

Esta sección inicializa el *buffer* que contiene los coeficientes que corresponden al filtro en cuestión extrayéndolos del archivo *coeficientes_filtros.dat.* Cabe recalcar que para realizar otro tipo de filtro solamente se debe cambiar el nombre del archivo de coeficientes.

```
_main:
    call _InitA;;
```

Esta instrucción llama a la rutina de inicialización del puerto de audio. La explicación de esta rutina se la realizó en capítulos anteriores.

```
    xr0 = FLAGREG;;
    xr0 = bset r0 by FLAGREG_FLAG2_EN_P;;
    xr0 = bclr r0 by 6;;
    FLAGREG = xr0;;
```

Con estas instrucciones se apaga el LED que corresponde a FLAG 2.

```
begin:
    j0=j31+dline;;
    jb0=j31+dline;;
    jl0=TAPS;;

    j1=j31+coefs;;
    jb1=j31+coefs;;
    jl1=TAPS;;
```

En esta sección se inicializan los *buffers* de retardo y de coeficientes del filtro FIR. Se observa que los registros base están inicializados con el mismo valor de los registros índice. Además, nótese que estos *buffers* se van a tratar como *buffers* circulares con longitud igual al orden del filtro.

```
    xr6=TAPS;;
    call _fir_init;;
```

Aquí se llama a la rutina de inicialización del filtro FIR que se explicará posteriormente. Además, se carga el registro xr6 con el valor de TAPS.

```
wait:
        nop;;
        nop;;
        nop;;
jump wait;
```

Con estas instrucciones se incluye dentro del programa una forma para que el DSP espere por una interrupción.

Fin de programa principal

❖ **Archivo "*Audio.asm*"**

Este archivo contiene la rutina de interrupción.

```
_AudioInt:

  nop;;
  nop;;
  nop;;
```

Se escriben tres instrucciones *nop* ya que ninguna instrucción IALU es permitida en los tres primeros ciclos de una rutina de interrupción.

```
  xr4 = [j31+_ReadDataLeft];;     //Lee el canal izquierdo
  xr9 =-16;;
  xfr7=float r4 by r9;;
```

En esta sección se toma la muestra del canal izquierdo y luego se la convierte a punto flotante multiplicándola primero por 2-16 ya que se sabe que el conversor

analógico a digital proporciona muestras codificadas en punto fijo y en complemente a 2 de 16 bits.

```
xr10=TAPS-1;;
xfr0=pass r7;;
call _filtro;;
```

Esta parte coloca la muestra actual de entrada en xfr0 y carga el número de veces que se va a ejecutar el ciclo del filtro FIR en xr10. Finalmente, se llama a la rutina de cálculo del filtro FIR, rutina que se verá posteriormente.

```
xfr7=pass r0;;
xr9=16;;
xr4=fix fr7 by r9;;
[j31+ _WriteDataLeft] = xr4;;        // Escribe en canal izquierdo
[j31+ _WriteDataRight] = xr4;;     // Escribe en canal derecho
rti (ABS)(NP);;
```

En esta sección se toma la muestra de salida que viene en xfr0 y se la convierte a punto fijo. Luego se carga este valor en los canales de salida izquierdo y derecho para que la muestra de salida se despliegue por los dos canales. Después de este proceso, finaliza el servicio a la interrupción.

❖ **Archivo "*FIR1.asm*"**

Este archivo contiene tanto la rutina de inicialización del filtro como la rutina de cálculo del filtro.

```
.global _fir_init;
.global _filtro;
```

A través de estas instrucciones se declara como globales las rutinas de inicialización y de cálculo del filtro FIR.

```
_fir_init:
    lc0=xr6;;
    xr11=0;;
 zero:
    nop;;
    nop;;
    nop;;
    CB[j0+=1]=xr11;;
 if nlc0e, jump zero(NP);;

 CJMP(NP);;
._fir_init.END:
```

Aquí se realiza la inicialización del filtro FIR, que no es otra cosa que poner en cero toda la línea de retardo del filtro.

```
_filtro:
    xr12=r12 xor r12; CB[j0+=1]=xr0;;
```

En este punto comienza la parte del programa que ejecuta el filtro FIR. Se inicializa el registro xr12 con el valor de cero y se almacena el valor de la muestra de entrada que viene en xr0 en la línea de retardo.

Obsérvese que el puntero de línea de retardo modifica una vez que ingresa la muestra de entrada en la línea de retardo. Por tal motivo, los coeficientes deben estar arreglados de manera que el último coeficiente sea el primero en el arreglo

```
    xr8=r8 xor r8; xr0=CB[j0+=1];;
    xr11=CB[j1+=1];;
```

Se inicializa el registro xr8 con cero porque este registro va a funcionar como acumulador. Además, se obtienen muestras desde la línea de retardo y desde el *buffer* de coeficientes.

```
    lc0=xr10;;
macs:
    xfr12=r0*r11; xfr8=r8+r12; xr0=CB[j0+=1];;
    xr11=CB[j1+=1];;

    if nlc0e, jump macs;;

    xfr12=r0*r11; xfr8=r8+r12;;
    xfr0=r8+r12;;

    CJMP(ABS)(NP);;
_filtro.END:
```

Con este grupo de instrucciones se realiza todo el proceso de multiplicación y acumulación. Nótese que se realiza una multiplicación, una suma y un acceso a memoria de datos en forma simultánea. Obsérvese también que se utilizan los mismos registros (xfr0 y xfr12) tanto como fuente como destino. Debe notarse que los valores de estos registros que se utilizan en la línea de instrucción *xfr12=r0*r11* son los que se leyeron en la instrucción anterior y no en la presente.

El desarrollo de la práctica para los filtros FIR se encuentra en el ANEXO 6.

7.3 FILTRO IIR

7.3.1 Fundamento teórico

A causa de que los filtros digitales IIR (Infinite Impulse Response) corresponden directamente a los filtros analógicos, una manera de diseñar un filtro IIR es crear una función de transferencia deseada en el dominio analógico y entonces transformarla al dominio Z. De esta manera los coeficientes de un filtro IIR pueden ser calculados a partir de la ecuación en el dominio Z. La siguiente ecuación es la ecuación de diferencias bicuadrada de forma directa para un filtro IIR:

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + a_1 y[n-1] + a_2 y[n-2]$$

La forma directa se usa continuamente en el diseño de los filtros IIR. Otra forma llamada la forma canónica, (también llamada la forma directa II) usa la mitad de las etapas de retardo, y por lo tanto utiliza menos memoria aunque esto no es un requerimiento crítico. Asumiendo un sistema lineal invariante con el tiempo (LTI), la forma directa puede ser manipulada matemáticamente para obtener la forma canónica. La ecuación para la forma canónica es:

$$w[n] = x[n] + a_1 w[n-1] + a_2 w[n-2]$$
$$y[n] = w[n] + b_1 w[n-1] + b_2 w[n-2]$$

La muestra de entrada es x[n] y y[n] es la muestra de salida. El termino w[n] se denomina valor intermedio; w[n-1] es el valor previo y w[n-2] el anterior a aquél. Las variables a y b son los coeficientes del filtro. El tipo de filtro IIR que se usara en el diseño de los filtros IIR se denomina bicuadrado y es un filtro de segundo orden. Los filtros de orden superior se pueden construir conectando en cascada varios filtros bicuadrados. La estructura general de los filtros IIR se muestra en la Figura 7.5.

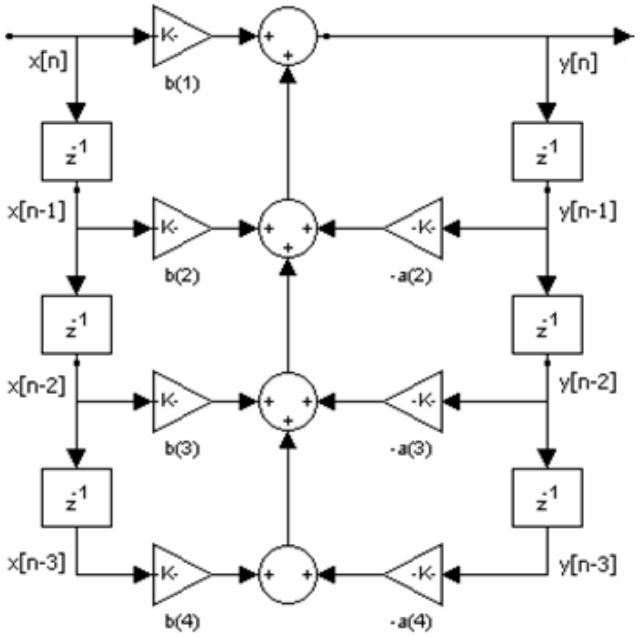

Figura. 7.5. Estructura del filtro IIR

Los filtros IIR tienen algunas ventajas sobre los filtros FIR:

- Los filtros IIR requieren menos memoria y menos instrucciones que los filtros FIR para implementar una función de transferencia especificada.

- Los filtros IIR contienen polos y ceros en el plano complejo. Los polos les dan a los filtros IIR una habilidad para implementar ciertas funciones de transferencia que los filtros FIR no pueden realizar.

- Los filtros IIR se traducen bien en modelos analíticos. Un ejemplo de esto, que se encuentra en la teoría del control, es la ecuación del controlador proporcional integro-diferencial (PID). Una implementación de esta este algoritmo utiliza una ecuación que corresponde a un filtro IIR.

- Los filtros IIR no son necesariamente estables, es tarea del diseñador asegurar la estabilidad.

El sobreflujo del procesador es algo que debe considerarse. Los filtros IIR se implementan con una operación de suma de productos que esta basada en la suma infinita. Esta estructura puede producir resultados que excedan el máximo valor que puede representar el procesador.

7.3.2 EXPLICACIÓN DEL PROGRAMA GENERAL DEL FILTRO IIR IMPLEMENTADO

En el CD que contiene la información del Proyecto de Grado se incluye el programa del filtro IIR pasabajos IIR_ING.dpj.

Se puede ver que el programa es muy similar al de los filtro FIR; por lo tanto, se explicará únicamente la parte del programa que implementa el filtro IIR que difiera del filtro FIR.

Como en el proyecto de los filtros FIR, el mismo proyecto sirve para los demás filtros de este tipo ya que únicamente deben cambiar los coeficientes.

❖ **ARCHIVO PRINCIPAL: *IIR_ING.asm***

```
.section data2;
.align 4;
.var dline[SECTIONS*2];
 .align 4;
.var coefs[SECTIONS*4+1]="coeficientes_iir.dat";
```

El archivo principal comienza por declarar e inicializar los buffers de retardo y de coeficientes utilizados por el filtro. El *buffer* ***dline*** mantiene la línea de retardo que contiene w[n-1] y w[n-2] para cada etapa bicuadrada del filtro. Ya que existen solamente dos valores intermedios anteriores por cada etapa bicuadrada, la longitud del filtro es dos veces el número de etapas. La línea de retardo está ordenada de la siguiente manera:

w[n-2] de la etapa 1, w[n-1] de la etapa 1, w[n-2] de la etapa 2, w[n-1] de la etapa 2, …

El *buffer* de coeficientes, que es el próximo *buffer* inicializado en el programa, debe estar ordenado de manera que coincida con el *buffer* de línea de retardo. El orden es:

a2 de la etapa 1, a1 de la etapa 1, b2 de la etapa 1, b1 de la etapa 1, a2 de la etapa 2, ….

```
j0=dline;;
jb0=dline;;
jl0=SECTIONS*2;;
j1=dline;;
k0=coefs;;
kb0=coefs;;
kl0=SECTIONS*4+1;;
```

Después de declarar los *buffers*, el programa asigna punteros a cada uno de ellos. La longitud del *buffer* ***dline*** es SECTIONS*2 y la longitud de ***coefs*** es SECTIONS*4 + 1.

```
call _iir_init;;
```

Antes de comenzar la filtración, la subruitina *_iir_int,* similar a la rutina _fir_init del filtro FIR, es llamada para llenar el *buffer* de la línea de retardo con ceros. Este *buffer* debe ser encerado debido a que se obtienen datos a partir de éste para realizar adiciones, la primera vez que se ingresa al lazo QUADS. Si el *buffer* no es encerado, las primeras muestras dentro del filtro serán incorrectas. La rutina *_iir_init* se encuentra en el archivo *IIR.asm*.

❖ **Archivo "*Audio.asm*"**

Este archivo contiene la rutina de interrupción, que es similar a la utilizada en los filtros FIR.

❖ **Archivo "*IIR.asm*"**

Este archivo contiene tanto la rutina de inicialización del filtro (_iir_init) como la rutina de cálculo del filtro (_filtro).

```
_iir_init:
    xr2=0;;
    lc0=xr20;;

    clear:
    CB[j0+=1]=xr2;; //coloca w[n-2]=0
    CB[j0+=1]=xr2;; //coloca w[n-1]=0
    if nlc0e, jump clear;;
    nop;;
    nop;;
```

```
    nop;;
    CJMP(ABS)(NP);;
_iir_init.end:
```

Esta parte se encarga de la inicialización de los *buffers dline* y *coefs*. El lazo *clear* se realiza tantas veces como haya secciones bicuadradas en el filtro. Cada vez que se realiza este lazo se escriben dos ceros en el *buffer*. Esto corresponde a escribir ceros en los dos valores intermedios (w[n-2] y w[n-1]) que se almacenan en cada etapa.

```
_filtro:
    nop;;
    nop;;
    nop;;
    j1=j0;;//j1 se usa para actualizar la linea de retardo
```

La rutina *_filtro* se llama dentro de la interrupción y se ejecuta cada vez que llega una muestra. La muestra de entrada se carga en el registro xr8. Los punteros a la línea de retardo y al *buffer* de coeficientes reciben la dirección del comienzo de los *buffers*. Una vez terminada la rutina *_filtro,* el valor de salida del filtro está en xr8. Esta rutina utiliza la instrucción j1=j0 para inicializar el puntero j1, el cual es el puntero de la línea de retardo que actualiza los valores intermedios.

```
    xr12=r12-r12;;
    xr2=cb[j0+=1];; // obtiene w[n-2]
    xr4=cb[k0+=1];; //coeficiente a2
    lc0=xr20;;
```

El filtraje comienza con este grupo de instrucciones. Estas instrucciones colocan xr12 en cero, extraen un valor de la línea de retardo y un coeficiente. Estos dos valores representan w[n-2] y a2 para la primera etapa bicuadrada. Estas instrucciones se ejecutan fuera del lazo QUADS de tal manera que los datos estén disponibles para ser multiplicados en la primera instrucción del lazo.

```
quads:
    xfr12=r2*r4; xfr8=r8+r12; xr3=cb[j0+=1]; xr4=cb[k0+=1];;
        // a2*w[n-2]; x[n]+0; obtiene w[n-1]; obtiene a1
    xfr12=r3*r4; xfr8=r8+r12; cb[j1+=1]=xr3; xr4=cb[k0+=1];;
        // a1*w[n-1]; x[n]+a2*w[n-2]; almacena w[n-2]; obtiene b2
    xfr12=r2*r4; xfr8=r8+r12; xr2=cb[j0+=1]; xr4=cb[k0+=1];;
        // b2*w[n-2]; nuevo w[n]; w[n-2] para nueva sección; obtiene b1

    xfr12=r3*r4; xfr8=r8+r12; cb[j1+=1]=xr8; xr4=cb[k0+=1];;
        // b1*w[n-1]; w[n]+{b2*w[n-2]}; almacenar nuevo w[n-1];
        // obtiene a2 para la próxima sección.

    if nlc0e, jump quads;;
```

Cada iteración del lazo QUADS representa una etapa bicuadrada del filtro. El registro *lc0* (al cual se le asignó un valor igual al número de etapas del filtro) se utiliza para indicar el número de veces que el lazo se repite.

Los registros para el cálculo utilizados por el lazo son:

- xr2 para w[n-2]
- xr3 para w[n-1]
- xr4 para coeficientes
- xr8 para suma de las multiplicaciones
- xr12 para multiplicaciones entre retardos y coeficientes.

```
xfr8=r8+r12;;            // nuevo y[n]
xr10=cb[k0+=1];;         //GANANCIA
xfr8=r8*r10;;
nop;nop;nop;;

    CJMP(ABS)(NP);;
_filtro.end:
```

Una vez que las etapas bicuadradas del filtro terminan, se requiere una suma adicional para obtener el resultado de la última etapa del filtro. Es necesario además, multiplicar este resultado por la ganancia total del filtro para obtener el resultado final.

Una vez cumplidas estas últimas dos operaciones finaliza la rutina para el cálculo del filtro IIR.

El desarrollo de la práctica para los filtros IIR se encuentra en el ANEXO 7.

7.4 FILTRO ADAPTATIVO

7.4.1 Fundamento teórico

Se han desarrollado muchos algoritmos computacionalmente eficientes para el filtraje adaptativo en los últimos tiempos. Están basados en un método estadístico, como el algoritmo LMS, o un método determinístico, como el algoritmo RLS. La principal ventaja del algoritmo LMS es su simplicidad computacional. El algoritmo RLS, por el contrario, ofrece una convergencia mas rápida, pero con un grado mayor de complejidad computacional.

El algoritmo LMS esta implementado con una estructura de filtro FIR. Ya que en los filtros FIR adaptativos solo se pueden ajustar los ceros, estos filtros están libres de los problemas de estabilidad que están asociados con los filtros IIR adaptativos donde tanto los polos como los ceros son ajustables.

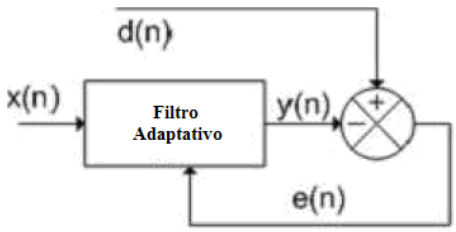

Figura. 7.6. Estructura de filtro adaptativo

Los filtros adaptativos se utilizan ampliamente en telecomunicaciones, sistemas de control, sistemas de radar, y otros sistemas en donde existe poca información disponible acerca de la señal de entrada. Algunas aplicaciones típicas de los filtros adaptativos son las siguientes:

En el diseño de controles para un sistema dinámico se debe tener un modelo que describa el sistema en acción. Sin embargo, modelar fenómenos físicos complejos, no es fácil. Uno puede obtener información acerca del sistema a ser controlado a partir de la recolección de datos experimentales y estimar los mejores valores para los parámetros desconocidos. Este proceso de construcción de modelos para fenómenos físicos se le conoce como identificación de sistemas y puede realizarse mediante filtros adaptativos que simulen el desempeño del sistema en cuestión.

Los filtros adaptativos se utilizan ampliamente en ecualización en modems que transmiten datos sobre la banda de voz y canales con mayor ancho de banda. Un ecualizador adaptativo se emplea para compensar la distorsión causada por el medio de transmisión. Su operación involucra un modo de entrenamiento seguido por un modo de rastreo.

En redes telefónicas, existe un dispositivo llamado “hibrido” que permite una operación de tipo *full-duplex* en una línea telefónica de dos cables. Debido al desacoplo entre el hibrido y el canal telefónico, se genera un eco, el cual puede ser suprimido mediante canceladores adaptativos de eco instalados en la red telefónica.

7.4.2 Filtro LMS

Algoritmo LMS implementado con una estructura de filtro FIR transversal

Las ecuaciones que se implementan en este algoritmo son las siguientes:

1. $y[n] = w[n].u[n], y[n] = salida\ del\ filtro\ FIR$

Donde:

$$w[n] = [w_0[n]\, w_1[n].....w_{N-1}[n]] = pesos\ del\ filtro$$

$$u[n] = [u[n]u[n-1].....u[n-N+1]] = muestras\ de\ entrada\ \ en\ la\ línea\ de\ retardo$$

$$n = índice\ de\ tiempo, N = número\ de\ pesos\ del\ filtro\ (TAPS)$$

2. $e[n] = d[n] - y[n], e[n] = señal\ de\ error\ y\ d[n] = salida\ deseada$

3. $w_i[n+1] = w_i[n] + STEPSIZE * e[n] * u[n-i], 0 \le i \le N-1$

EXPLICACIÓN DEL PROGRAMA GENERAL DEL FILTRO LMS IMPLEMENTADO

El objetivo del programa es predecir la muestra de entrada en base a un conjunto de muestras anteriores, y con un cierto retardo con la muestra actual.

Una vez que se recibe los datos de entrada se envía al filtro adaptativo que se encarga de calcular la muestra de salida de acuerdo a su correspondiente algoritmo. El algoritmo LMS se encuentra en al archivo *LMS.asm* y se describe a continuación.

❖ Archivo "LMS.asm"

```
#define TAPS 120                 //Orden del filtro
#define STEPSIZE 0.005           // Tamaño del peso

.section data1;
.var deline_data[TAPS];
```

Este *buffer* define la línea retardo para las muestras de la señal de entrada.

.var weights [TAPS];

La variable *weights* define un *buffer* para los pesos del filtro

_lms_init:

xr0=0x0;;
xr1=0x0;;
xr2=0x0;;
xr3=0x0;;
xr7:4=xr3:0;;
xr11:8=xr3:0;;
xr15:12=xr3:0;;

Encerado de los registros de propósito general

lc0=TAPS;; // Contador de lazo LC0 con el valor del orden del filtro
j0=deline_data;;
jb0=deline_data;;
jl0=TAPS;;

Se asigna la base y la longitud del *buffer* circular de la variable *deline_data*, observe que la longitud del *buffer* de la línea de retardo es igual al orden del filtro.

k1=weights;;
kb1=weights;;
kl1=TAPS;;

En esta sección se realiza la asignación del *buffer* circular para los pesos del filtro, la longitud del *buffer* de pesos es igual al orden del filtro.

k2=k1;;
kb2=kb1;;
kl2=kl1;;

El registro K2 es inicializado igual que el registro k1, es decir contiene la misma base, longitud y apunta a la misma dirección de la variable *weights*.

```
            xr7=STEPSIZE;;
            xr0=0.0;;
 clear_bufs:
            CB[j0+=-1]=xr0;;
            CB[k1+=1]=xr0;;
  if nlc0e, jump clear_bufs;;
```

Se realiza la inicialización del filtro LMS, se encera la línea de retardo y los pesos del filtro.

```
lms_alg:
            CB[j0+=-1]=xr0;;    //almacena u(n) en la línea de retrasos,
            xr4=CB[k1+=1];;     //f4=w0(n)
            xfr8=r0*r4;xr0=CB[j0+=-1];xr4=CB[k1+=1];;
            // f8=u(n)*w0(n), f0=u(n-1), f4=w1(n)
            xfr12=r0*r4;xr0=CB[j0+=-1];xr4=CB[k1+=1];;
            //f12=u(n-1)*w1(n), f0=u(n-2), f4=w2(n)
            lc0= TAPS-3;;
macs:       xfr12=r0*r4;xfr8=r8+r12;xr0=CB[j0+=-1];xr4=CB[k1+=1];;
            // f12=u(n-i)*wi(n), f8= suma de productos, f0=u(n-i-1), f4=wi+1(n)
            if nlc0e, jump macs;;
            NOP;;
            NOP;;
            xfr12=r0*r4;xr8=r8+r12;;       f12=u(n-N+1)*wN-1(n)
            xfr13=r8+r12;;          // f13=y(n)
            xfr6=r1-r13;;                   // f6=e(n)
            xfr1=r6*r7;xr4=CB[j0+=-1];;

    //f1=STEPSIZE*e(n),f4=u(n)
```

```
xfr0=r1*r4;xr12=CB[k1+=1];;  //f0=STEPSIZE*e(n)*u(n),f12=w0(n)
lc1=TAPS-1;;

update_weights:
xfr8=r0+r12;xr4=CB[j0+=-1];xr12=CB[k1+=1];;
//f8=wi(n+1),f4=u(n-i-1),f12=w0(n)
xfr0=r1*r4;CB[k2+=1]=xr8;;
//f0=STEPSIZE*e(n)*u(n-i-1), almacena wi(n+1)

if nlc1e, jump update_weights;;
```

Realiza la actualización de los coeficientes del filtro

```
NOP;;
NOP;;
xfr8=r0+r12;xr0=CB[j0+=1];;
//apunta a u(n+1) posición de la línea de retardo
CB[k2+=1]=xr8;;
// almacena wN-1(n+1)
```

7.4.3 Filtro NLMS

Algoritmo NLMS normalizado implementado con una estructura de filtro FIR transversal.

1. $y[n] = \underline{w}[n].\underline{u}[n],\ y[n] = salida\ del\ filtro\ FIR$

   Donde:

   $\underline{w}[n] = \left[w_0[n] w_1[n] ..... w_{N-1}[n]\right] = pesos\ del\ filtro$

   $\underline{u}[n] = \left[u[n] u[n-1] ..... u[n-N+1]\right] = muestras\ de\ entrada\ en\ la\ línea\ de\ retardo$

   $n = índice\ de\ tiempo, N = número\ de\ pesos\ del\ filtro\ (TAPS)$

2. $e[n] = d[n] - y[n], e[n] = señal\ de\ error\ y\ d[n] = salida\ deseada$

3. $w_i[n+1] = w_i[n] + STEPSIZE\,(normalizado) * e[n] * u[n-1], 0 \le i \le N-1$

4. $STEPSIZE\,(normalizado) = (ALPHA) / GAMMA + E[n])$

Donde:

$$E[n] = \underline{u}[n].\underline{u}[n] = energía\ \ de\ la\ línea\ de\ retardo$$

$$E[n]\ se\ calcula\ recursivamente\ como\ sigue:$$

5. $E[n] = E[n-1] + u[n] **2 - u[n-N] **2$

EXPLICACIÓN DEL PROGRAMA GENERAL DEL FILTRO NLMS IMPLEMENTADO

❖ **Archivo "NLMS.asm"**

```
#define TAPS 241          // Orden del filtro
#define STEPSIZE 0.005    // Tamaño del peso
#define ALPHA 0.5         // Parámetros para velocidad de convergencia
#define GAMMA 0.5
.GLOBAL _calcular;

.section data1;
.var indata[TAPS]= "ejer\input_3000hz.dat";
.var deline_data[TAPS];
```

Este *buffer* define la línea retardo para las muestras de la señal de entrada.

```
.section data2;
.var desired[TAPS]= "ejer\desired_3000hz.dat";
```

Carga los datos de la señal de deseada

```
.VAR output[TAPS];
```

La variable *output* define un *buffer* los resultados de la salida del filtro

```
.section data10a;
.align 4;
```

.var weights [TAPS];

La variable *weights* define un *buffer* para los pesos del filtro

.section program;

```
xr0=0x0;;
xr1=0x0;;
xr2=0x0;;
xr3=0x0;;
xr7:4=xr3:0;;
xr11:8=xr3:0;;
xr15:12=xr3:0;;
xr19:16=xr3:0;;
xr0=0.0;;
xr1=0.0;;
```

Encerado de los registros de propósito general

```
lc0=TAPS;;   // Contador de lazo LC0 con el valor del orden del filtro
j0=deline_data;;
jb0=deline_data;;
jl0=TAPS;;
```

Se asigna la base y la longitud del *buffer* circular de la variable *deline_data*, observe que la longitud del *buffer* de la línea de retardo es igual al orden del filtro.

```
k0=weights;;
kb0=weights;;
kl0=TAPS;;
```

En esta sección se realiza la asignación del *buffer* circular para los pesos del filtro, la longitud del *buffer* de pesos es igual al orden del filtro.

```
k1=weights;;
kb1=weights;;
kl1=TAPS;;
```

El registro K1 es inicializado igual que el registro k0, es decir contiene la misma base, longitud y apunta a la misma dirección de la variable *weights*.

```
xr5=ALPHA;;
xr11=GAMMA;;   //f11=E(0)+GAMMA
xr9=2.0;;      // Valor utilizado para la división
r13=0.0;;
```

Se asigna la base y la longitud del *buffer* circular de la variable *indata*, la longitud del *buffer* de entrada es igual al orden del filtro.

```
j1=indata;;
jb1=indata;;
jl1=TAPS;;
```

Se asigna la base y la longitud del *buffer* circular de la variable *output*, la longitud del *buffer* de salida es igual al orden del filtro.

```
k2=output;;
kb2=output;;
kl2=TAPS;;
```

Se asigna la base y la longitud del *buffer* circular de la variable *output*, la longitud del *buffer* de salida es igual al orden del filtro.

```
j2=desired;;
jb2=desired;;
jl2=TAPS;;
```

Se asigna la base y la longitud del *buffer* circular de la variable *desired*, la longitud del *buffer* es igual al orden del filtro.

```
j1 = j31 + indata;;          // apunta a la dirección de inicio de indata
k2 = k31 + output;;          // apunta a la dirección de inicio de output
j2 = j31 + desired;;         // apunta a la dirección de inicio de desired
clear_buf:
CB[j0+=-1]=xr13;;
CB[k0+=1]=xr13;;
```

if nlc0e, jump clear_buf(NP);;

Se realiza la inicialización del filtro LMS, se encera la línea de retardo y los pesos del filtro.

nlms_alg:

xr0 = [j1+=1];;

f0= u(n)

xr1 = [j2+=1];;

f1= d(n)

xfr14=r0*r0;CB[j0+=-1]=xr0;xr4=CB[k0+=1];;

f14= u(n)**2, almacena u en la línea de retardo, f4=w0(n)

xfr8=r0*r4;xfr11=r11+r14;xr0=CB[j0+=-1];xr4=CB[k0+=1];;

f8=u(n)*w0(n), f11=E(n-1)+u(n)*2

xfr12=r0*r4;xfr11=r11-r13;xr0=CB[j0+=-1];xr4=CB[k0+=1];;

f12=u(n-1)*w1(n), f11=E(n), f0=u(n-2), f4=w2(n)

lc1=TAPS-3;;

NOP;;

macs1:

xfr12=r0*r4;xfr8=r8+r12;xr0=CB[j0+=-1];xr4=CB[k0+=1];;

f12=u(n-i)*wi(n), f8=suma de productos, f0=u(n-i-1), f4=wi+1(n)

if nlc1e, jump macs1(NP);;

NOP;;

NOP;;

xfr12=r0*r4;xfr8=r8+r12;xr14=xr11;;

f12=u(n-N+1)*wN-1(n), f14=E(n)

xfr2=r8+r12;; f2=y(n)

xfr6=r1-r2;xr4=CB[j0+=-1];; f6=e(n), f4=u(n)

xfr7=r6*r5;; f7=ALPHA*e(n)

Realiza la división de dos números flotantes

xr20=0.0;; //xr20=xr7

xr21=0.0;;

xr22=0.0;;

```
            xr21=xr7;;   //numerador
            xr22=xr14;;   // denominador
            xr20=xr21;;
            xfr21=RECIPS r22;;
            xfr22=r21*r22;;
            xfr20=r21*r20; xfr21=r9-r22;;
            xfr22=r21*r22;;
            xfr20=r21*r20; xfr21=r9-r22;;
            xfr22=r21*r22;;
            xfr20=r21*r20; xfr21=r9-r22;;
            xfr1=r21*r20;;        f1=STEPSIZE(Normalizado)*e(n)
            xfr0=r1*r4;xr12=CB[k0+=1];;
            f0=f1*u(n), f12=w0(n)
            lc0=TAPS-1;;
            NOP;;
update_wei:
            xfr8=r0+r12;xr4=CB[j0+=-1];xr12=CB[k0+=1];;
            f8=wi(n+1), f4=u(n-i-1), f12=wi+1(n)
            xfr0=r1*r4;CB[k1+=1]=xr8;;
            f0=STEPSIZE(normalizado)*e(n)*u(n-i-1), almacena wi(n+1)
if nlc0e, jump update_wei(NP);;
            NOP;;
            NOP;;
            xfr8=r0+r12;xr0=CB[j0+=1];;
            f8=wN-1(n+1), u(n+1) posición en la línea de retardo
            xfr13=r4*r4;CB[k1+=1]=xr8;;
            f13=u(n-N)*2, almacena wN-1(n+1)
[k2+=1] = xr2;;            //almacena y (n)
if NLC1E, jump nlms_alg;
NOP;;
NOP;;
CJMP(NP);;
._calcular.END
```

El desarrollo de la práctica para los filtros adaptativos se encuentra en el ANEXO 8.

7.5 ALGORITMOS PARA SISTEMAS DISCRETOS

En el sistema de Procesamiento Digital de Señales de la tarjeta EZ-KITLite ADSP-TS201s se realiza dos tipos de algoritmos sobre las muestras de una señal analógica de entrada digitalizada. Los dos algoritmos a los que nos referiremos en particular son:

- Algoritmo de eco.
- Algoritmo de reverberación.

7.5.1 Fundamento teórico

Un sistema de Procesamiento Digital de Señales (PDS) es aquel que realiza algún tipo de transformación (algoritmo) sobre las muestras de una señal analógica de entrada digitalizada.

En muchos casos, el resultado de esa transformación es otra señal digital que puede pasarse nuevamente, si así es requerido, a formato analógico, aunque, en general, la salida del sistema puede ser cualquier tipo de información obtenida de la señal de entrada. Un esquema de un sistema de procesamiento digital de señal sería el siguiente:

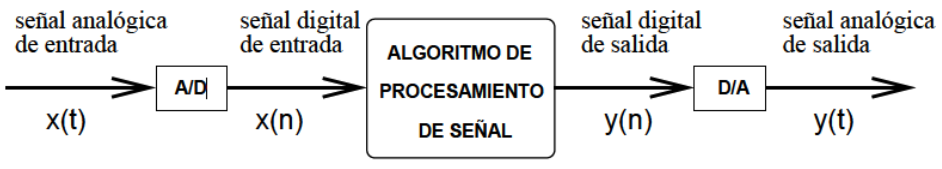

Figura. 7.7. Esquema de un procesamiento digital de señales

Efectos que utilizan retardos o delays

Muchos efectos se consiguen sumando a la señal original, varias copias retardadas y modificadas de diversas formas. De entre todos ellos, los más típicos son los de eco y reverberación, aunque, como se conoce, no son los únicos. Dependiendo del tipo de efecto buscado, los tiempos de estos retardos pueden valer entre las pocas milésimas y los varios segundos.

Naturaleza del eco y la reverberación

En una sala, la reverberación se produce de forma natural porque los sonidos que nos llegan a los oídos no proceden de un único punto emisor, sino que recibimos también "copias" reflejadas por las paredes, el techo, el suelo, y otros objetos. Cuando más distantes de nosotros estén estos reflectores, más retardadas y también más atenuadas recibiremos las copias.

El tiempo de reverberación es una propiedad de las salas, y se define como el lapso que debe transcurrir para que el sonido inicial se atenúe en 60 dB. En salas grandes, este tiempo puede durar varios segundos. Otro factor que influye en la reverberación es la absorción de los materiales reflectantes. Superficies poco absorbentes, como el cristal, acrecientan el tiempo de reverberación, mientras que otras, como las cortinas o el propio público, hacen que este valor disminuya.

Normalmente, la absorción varía también con la frecuencia (los agudos se absorben más que los graves). Todo ello hace que un buen algoritmo digital de reverberación deba incluir muchos parámetros configurables.

Cuando estos retardos son suficientemente grandes como para oírse de forma aislada se denominan ecos, y sólo pueden producirse en espacios muy amplios y con pocos obstáculos (frente a una montaña, etc.). Si nos encontramos situados entre dos obstáculos distantes, el sonido sufrirá varias reflexiones antes de extinguirse, por lo que oiremos varios ecos sucesivos de intensidades decrecientes.

Reverberación y eco digitales

Dado que un alto porcentaje de la música grabada se escucha en pequeñas salas particulares con tiempos de reverberación muy cortos, la mayoría de grabaciones comerciales incorporan la reverberación y otros efectos que emulan la acústica de grandes salas y las hacen más gratas al oído. Pero los estudios de grabación tampoco pueden ofrecer reverberaciones naturales convincentes, por lo que desde hace décadas se ha investigado mucho en sistemas de reverberación artificiales.

En los años 60 y 70 estos efectos se reproducían de forma analógica (utilizando cintas magnetofónicas) o incluso mecánica (mediante la disposición de placas metálicas, micrófonos y altavoces), pero desde hace unos años, los ecos y reverberaciones digitales han desbancado por completo a los antiguos sistemas.

7.5.2 Algoritmo de Eco

a) EXPLICACIÓN DEL PROGRAMA GENERAL

En el CD que contiene la información del Proyecto de Grado se incluye el proyecto que implementa el algoritmo de eco para sistemas discretos.

Se puede ver que el programa es similar en su parte de configuración del canal de audio; por lo tanto, se explicará únicamente la parte del programa que implementa el algoritmo que difiera del resto de proyectos ya analizados previamente.

❖ **ARCHIVO PRINCIPAL**

```
.global _DelayLine;
.extern _InitA;
.section data2a;
.align 4;
.var _DelayLine[24000];
```

```
.section program;
_start:
    call _InitA (NP);;          // Inicializa el canal de audio
    j0  = _DelayLine;;
    jb0 = j0;;
    jl0 = 24000;;
```

En esta sección se inicializa el *buffer* de la línea de retardo para su uso en el servicio de interrupción. Se da como dirección de inicio a *_DelayLine* y su longitud será igual al tamaño del mismo vector.

```
_MainLoop:
    jump _MainLoop (NP);;
```

Esta sección es un lazo que sirve al programa como espera por la rutina del servicio de interrupción ISR.

- **Archivo "InitA.asm"**

Este archivo inicializa y configura el canal de audio. Su explicación detallada fue realizada en capítulos anteriores.

- **Archivo "Audio.asm"**

En este archivo radica la parte fundamental de este algoritmo. Como ya se conoce, el archivo *Audio.asm* contiene el servicio de rutina de interrupción. Sin embargo, para este proyecto, el archivo *Audio.asm* va a contener dos partes.

La primera parte corresponde a la implementación del algoritmo de eco pero su lectura va a estar condicionada a la definición de una macro llamada DELAY. Es decir, si el usuario define la macro DELAY dentro del proyecto, en la parte de la rutina de interrupción el proyecto va a realizar la parte correspondiente al algoritmo

de eco. Por el contrario, si la macro DELAY no es definida, el programa va a realizar la segunda parte de la rutina de interrupción que corresponde a un Audio *Pass Through* normal.

DEFINICIÓN DE LA MACRO DE PREPROCESADOR *'DELAY'*

Una definición de preprocesador para la macro DELAY es incluida en las opciones del proyecto y es implementado en la instrucción de código condicional "#ifdef DELAY", dentro del archivo *Audio.asm*. Si la macro es definida, entonces el procesamiento para línea de retraso será ejecutado. Por su parte, si esta macro no existe, las muestras de audio serán pasadas directamente a la salida sin procesamiento alguno exactamente como un programa original de Audio *Pass Through*.

Para definir la macro de preprocesador *DELAY*, se debe acceder al menú *Project* y seleccionar *Project Options*. Presione sobre la etiqueta *Assemble* y después en el cuadro de diálogo que corresponde a *Preprocessor Definitions* se debe ingresar los caracteres de definición, "*DELAY*" (sin las comillas) para poder definir esta macro.

Si se desea quitar la definición de la macro, simplemente se debe borrar la palabra DELAY del cuadro de diálogo que corresponde a *Preprocessor Definitions*.

EXPLICACIÓN DE LA RUTINA *_AudioInt*

```
_AudioInt:
    nop;;
    nop;;
    nop;;

#ifdef DELAY
```

La instrucción *#ifdef DELAY* establece el condicional para determinar que parte del programa va a realizarse de acuerdo a si la macro DELAY se encuentra definida o no.

Si la macro DELAY se encuentra definida el siguiente grupo de instrucciones serán realizadas:

```
xr4 = [j31 + _ReadDataLeft];;          // Lee canal izquierdo
xr5 = [j31 + _ReadDataRight];;         // Lee canal derecho
```

Aquí se lee las muestras de entrada del canal izquierdo y derecho respectivamente.

```
xr1 = CB [j0 += j31];;
```

Con esta instrucción se obtiene la última muestra del *buffer* de la línea de retardo y se lo almacena en xr1.

```
CB [j0 += 1] = xr4;;
```

Aquí se ubica las muestras de entrada del canal izquierdo dentro del *buffer* circular de la línea de retardo.

```
xr6 = r5 + r1;;
```

En el registro xr6 se va a cargar la suma de la muestra de entrada del canal derecho con la muestra de entrada del canal izquierdo retrasada.

```
[j31 + _WriteDataLeft] = xr4;;     // Canal izquierdo (pass through)
[j31 + _WriteDataRight] = xr6;;    // Canal derecho (muestra retrasada)
rti (ABS)(NP);;                    // Retorno de la interrupción
```

Finalmente, en el canal de salida izquierdo se va a escribir el dato original que estaba almacenado en xr4; es decir cumple una función de *PassTrough*. Por su parte, en el canal de salida derecho se va a escribir la muestra almacenada en xr6; es decir, el canal derecho va a reproducir la muestra original y un tiempo después una muestra idéntica a la original, con lo cual se simula el efecto de eco deseado.

Si la macro *DELAY* no se encuentra definida el siguiente grupo de instrucciones serán realizadas:

```
#else
    xr4 = [j31 + _ReadDataLeft];;          // Lee canal izquierdo
    [j31 + _WriteDataLeft] = xr4;;         // Escribe en canal izquierdo
    xr5 = [j31 + _ReadDataRight];;         // Lee en canal derecho
    [j31 + _WriteDataRight] = xr5;;        // Escribe en canal derecho
    rti (ABS)(NP);;                        // Retorno de la interrupción
```

Esta parte del programa ejecuta simplemente una función de Audio *PassTrough* que ya fue analizada en programas anteriores.

```
#endif
_AudioInt.end:
```

Aquí finaliza el servicio a la rutina de interrupción.

7.5.3 Algoritmo de reverberación

EXPLICACIÓN DEL PROGRAMA GENERAL

El programa configura un controlador de audio DMA y realiza un algoritmo de reverberación para simular un efecto de teatro. El programa esta implementado también para tener la funcionalidad *passthrough* (sin el efecto de reverberación).

El botón IRQ_A controla la activación o desactivación del algoritmo de reverberación.

- El LED4 está encendido cuando la reverberación está activada.
- El LED4 está apagado cuando la reverberación está desactivada.

Una FFT compleja de 64K y una convolución con la respuesta de impulso del teatro son efectuadas en tiempo real. Al usar este algoritmo de reverberación, la señal de audio suena como si estuviera justamente en una casa de teatro.

MUESTREO DE REVERBERACIÓN

- DSP A realiza la reverberación para el canal izquierdo.
- DSP B realiza la reverberación para el canal derecho.

Cada DSP realiza una FFT compleja de 64 K usando el método de superposición y suma. La respuesta impulsiva es una grabación del teatro.

PROYECTO

El algoritmo de reverberación del DSPA contiene los siguientes archivos:

- *audio.asm*
- *convolve_a.asm*
- *data_a.asm*
- *fft_flp32_a.asm*
- *initialize_a.asm*
- *main_a.asm*
- *toggle.asm*

El algoritmo de reverberación del DSPB contiene los siguientes archivos:

- *convolve_b.asm*
- *data_b.asm*
- *fft_flp32_b.asm*
- *initialize_b.asm*
- *main_b.asm*

Aparte constan en el proyecto los archivos *reverb.h* y *cache_macros.h*.

EXPLICACION DEL ALGORITMO

- **Archivo "Reverb.h"**

#define VOLUME 19

Configura el volumen que tendrá el canal de salida audio. El valor máximo posible para el volumen es de 23.

#define BUFFER 65536

Esta instrucción configure el tamaño de los *buffer* de audio.

#define SCALE 1.0 / (2.0 * BUFFER)

La constante SCALE guarda el factor de escalamiento para el cálculo de la FFT inversa.

#define START_CONVOLUTION 0
#define BYPASS_CONVOLUTION 1

Estas constantes sirven para realizar el control de la convolución.

#define CODEC 0x38000000

Esta constante sirve para guardar la dirección del codec.

#define REAL 0
#define COMPLEX 1

Las constantes REAL y COMPLEX sirven para determinar el tipo de filtro FFT que se está usando.

DSP A

A continuación se analiza los aspectos más importantes que componen los archivos del algoritmo de reverberación del canal izquierdo.

- **Archivo "*main_a.asm*"**

  Este archivo realiza la configuración para el proceso de convolución en el DSPA y también en el DSPB.

  xr0 = [j31 + convolution_control];;

  Carga en el registro xr0 el valor del control de convolución actual.

  xbitest r0 by START_CONVOLUTION;;
  if nxseq; do, xr0 = bclr r0 by START_CONVOLUTION (nf);
  do, j28 = j31 + convolve_right - P1_OFFSET_LOC (nf);;
  // configura convolucion para canal derecho.

  if nxseq; do, [j31 + P1_OFFSET_LOC + VIRPT_LOC] = j28;;
  // inicia convolucion en DSPB

  if nxseq; do, [j31 + convolution_control] = xr0;;
  // actualiza el control de convolucion

  if nxseq, call convolve_left (p);; // inicia la convolucion en el DSP A

  Con la primera instrucción el programa compara si el *buffer* está lleno o no. Si es que el *buffer* está lleno el programa procede a vaciarlo. Si el *buffer* está vacío, el programa configura e inicia la convolución en ambos canales. Finalmente, el programa actualiza el control de convolución.

❖ **Archivo initialize_a.asm**

Este archivo se encarga de inicializar el procesador DSP_A y sus periféricos. En este archivo se llama a las funciones que se encargarán de la inicialización del puerto de audio y de la inicilización de la IRQ0 que se encarga del monitoreo de la activación o desactivación del proceso de convolución.

```
call initialize_toggle (np);;  //  INICIALIZA IRQ0
call initialize_audio (np);;  // INICIALIZA EL PUERTO DE AUDIO
```

❖ **Archivo "*audio.asm*"**

Este archivo inicializa y mantiene la rutina de audio para ambos canales. Aquí se puede encontrar la rutina de la función ***initialize_audio (np)*** que es la encargada del proceso de inicialización de los canales de audio izquierdo y derecho.

❖ **Archivo "fft_flp32_a.asm"**

Este archivo contiene la rutina para el cálculo de la FFT Compleja de raíz 2. Esta rutina fue analizada en la práctica que corresponde al tema de la FFT.

❖ **Archivo "data_a.asm"**

Este archivo contiene la declaración de las variables que serán usadas por el procesador A (DSPA). Entre estas variables se encuentran el buffer para el canal izquierdo y los factores *twiddle* para el cálculo respectivo de la FFT.

❖ **Archivo "convolve_a.asm"**

Este archivo realiza la convolución de la entrada del canal de audio izquierdo con la respuesta impulsiva.

❖ **Archivo "toggle_a.asm"**

Este archivo monitorea el estado de IRQ0 cuando se activa o desactiva el proceso de convolución. Aquí se puede encontrar la rutina de la función ***call initialize_toggle (np)*** que es la encargada del proceso de monitoreo.

DSP B

- **Archivo "*main_b.asm*"**

  El archivo principal del DSPB simplemente tiene en su código un lazo infinito con el propósito de esperar por una interrupción.

- **Archivo "initialize_b.asm"**

  Este archivo se encarga de inicializar el procesador DSP_B y sus periféricos.

- **Archivo "fft_flp32_b.asm"**

  Este archivo contiene la rutina para el cálculo de la FFT Compleja de raíz 2. Esta rutina fue analizada en la práctica que corresponde al tema de la FFT.

- **Archivo "data_b.asm"**

  Este archivo contiene la declaración de las variables que serán usadas por el procesador B (DSPB). Entre estas variables se encuentran el *buffer* para el canal derecho y los factores *twiddle* para el cálculo respectivo de la FFT.

- **Archivo "convolve_b.asm"**

  Este archivo realiza la convolución de la entrada del canal de audio derecho con la respuesta impulsiva.

El desarrollo de la práctica para los filtros adaptativos se encuentra en el ANEXO 9.

CAPITULO VIII

CONCLUSIONES Y RECOMENDACIONES

8.1 CONCLUSIONES

Se ha realizado el estudio de la tarjeta "ADZS-TS201S" de *Analog Devices*, Familia *TigerSHARC*, así como también el análisis y estudio de la arquitectura del procesador ADSP-TS201.

Se verificó el adecuado funcionamiento de la tarjeta mediante la implementación de programas para generar aplicaciones como por ejemplo, manipulación de LEDS y pulsadores, accesos hacia y desde memoria, grabación de voz y un programa para establecer comunicación entre tarjetas ADZS-TS201.

Se describió el software de desarrollo VisualDSP++ 5.0 que constituye la herramienta de programación dentro de este estudio. Este software presenta un entorno que facilita la utilización de herramientas para la creación, edición y ejecución de proyectos por parte del programador.

Con el estudio de la tarjeta se dejan las bases necesarias para seguir con el desarrollo de futuros proyectos, debido a que están establecidas y dotadas las herramientas para desarrollar innumerables implementaciones, lo cual implica que no se tendrá que realizar un estudio de la tarjeta desde un inicio sino que simplemente se requerirá una investigación particular en base a una necesidad específica.

Se llegaron a entender las diferentes utilidades que tienen los DSP, como la realización de cálculos complejos en tiempo real, debido a que su arquitectura aumenta la capacidad de procesamiento. Se logró comprender las principales herramientas con las que dispone la tarjeta de acuerdo al desarrollo del Proyecto de Grado.

Se logró implementar varios algoritmos de filtros digitales para el desarrollo de las prácticas de laboratorio para el procesamiento de señales digitales en tiempo real.

Se logró implementar filtros digitales con una respuesta en frecuencia, y se comprobó el desempeño de los filtros para su posterior uso durante el desarrollo de las prácticas de laboratorio.

Se realizó la comunicación punto a punto entre tarjetas y entre núcleos de una misma tarjeta. La comunicación fue implementada en base al envió de números randómicos generados en el transmisor (DSPA) y recibidos por el receptor (DSPB).

Se comprobó experimentalmente el desempeño de algoritmos avanzados como son los algoritmos de filtros adaptativos LMS y NLMS.

Los programas que utilicen la interfaz de audio solo pueden ser ejecutados en el procesador A (DSPA).

8.2 RECOMENDACIONES

La licencia del VisualDPS++ 5.0 tiene una duración de 90 días. Una vez que la licencia haya expirado se recomienda desinstalar el *software* y reinstalarlo nuevamente para poder renovar el tiempo de expiración de licencia. Si no se procede a la desintalación de la herramienta, Visual DSP++ 5.0 no podrá ser usado.

Se recomienda que VisualDSP++ 5.0 sea instalado en computadores que tengan sistemas operativos Windows XP y Windows Vista para obtener un mejor resultado de funcionamiento y compatibildad entre el software y la tarjeta ADZS-TS201S.

Se debe tomar muy en cuenta la configuración del formato de los registros de los bloques de cálculo previo a su utilización dentro de las operaciones. Tomar en consideración el signo, el tipo y la longitud de las operandos al implementar estas operaciones.

Se recomienda tener especial cuidado en la configuración de los interruptores del DIP *switch* SW1 de la tarjeta debido a que solamente la correcta configuración de estos permitirá realizar implementaciones con señales de audio. Se debe tener en cuenta además, que cualquier modificación que se quiera realizar en la configuración de los DIP *switch* debe ser realizada mientras la tarjeta se encuentra desconectada tanto de la fuente de poder como de la PC.

Cuando se compila un proyecto, se debe colocar el directorio del archivo ejecutable *.dxe* correspondiente al proyecto en el procesador en el cual se requiere que sea ejecutado. Por ejemplo, los proyectos que contengan rutinas de audio deberán ser ejecutados exclusivamente en el procesador DSPA para que puedan ser validados.

Los archivos externos deben ser colocados en la carpeta en la cuál se ha creado el proyecto de VisualDSP++ 5.0 para que estos puedan ser incluidos y reconocidos por el proyecto una vez realizada la compilación.

Cuando se implementa un lazo dentro de un programa es necesario ubicar instrucciones *NOPs* inmediatamente después de finalizado dicho lazo debido a que el valor que retorna del lazo puede afectar a la siguiente línea de instrucción dentro del programa. También es recomendable no emplear instrucciones IALU dentro de las tres primeras líneas del código del programa debido a que esto podría ocasionar conflictos en el procesamiento del mismo.

ANEXOS

ANEXO 1

PROGRAMA PARA MANIPULACIÓN DE LEDS Y PULSADORES

```
//*****************************************************************
//
//      Código de Ejemplo para la Manipulación de LEDS y Pulsadores
//      ejerciciodeleds.asm
//
//*****************************************************************

 #include <defTS201.h>

.SECTION program;
.global _main;

_main:
        xR0=0x00000000;;
        FLAGREG=xR0;;           // encera el registro FLAGREG
        xR0 = 0x0000000C;;
        FLAGREG = xR0;;         // habilita como salida FLAG3 y FLAG2
        call retardo(ABS);;
        xR0 = 0x0000004C;;
        FLAGREG = xR0;;         // escribe un "1" en FLAG2,en la salida se enciende
                                        // el LED4
        call retardo(ABS);;
        xR0 = 0x0000008C;;
        FLAGREG = xR0;;         // escribe un "1" en FLAG3, en la salida se
                                // enciende el LED6
        call retardo(ABS);;

//*****************************************************************
// Habilitación de la interrupción IRQ0

        j0 = _IRQ0_ISR;;
        IVIRQ0 = j0;;           // seteo IVIRQ0 del vector interrupción
        xR0 = INT_IRQ0;;
        IMASKH = xR0;;          // interrupción IRQ0 sin máscara
        xr0 = SQCTL;;
        xr0 = bset r0 by SQCTL_GIE_P;  // habilita interrupciones globales sin máscara
        SQCTL = xR0;;
_testflag0:
```

```
        xr0=SQSTAT;;
        bitest r0 by 16;;
        if xSEQ, jump _apaga_flag2 (NP);;
        _enciende_flag2:
        flagregst = FLAGREG_FLAG2_OUT;;   // enciende LED 4/FLAG2
        call retardo(ABS);;
        jump _testflag1 (NP);;
        _apaga_flag2:
        flagregcl = ~(FLAGREG_FLAG2_OUT);; // apaga LED 4/FLAG2

_testflag1:
        xR0 = SQSTAT;;
        bitest r0 by 17;;
        if xSEQ, jump _apaga_flag3 (NP);;
        _enciende_flag3:
        flagregst = FLAGREG_FLAG3_OUT;;    // enciende LED 6/FLAG3
        call retardo(ABS);;
        jump _testflag0 (NP);;
        _apaga_flag3:
        flagregcl = ~(FLAGREG_FLAG3_OUT);;  // apaga LED 6/FLAG3

        jump _testflag0(NP);;

//*****************************************************************
// Verificación del pulsador IRQ
// Activación por interrupción de IRQ

_IRQ0_ISR:

        xr0 = FLAGREG;;
        xr0 = btgl r0 by FLAGREG_FLAG2_OUT_P;; // cambia FLAG2
        xr0 = btgl r0 by FLAGREG_FLAG3_OUT_P;; // cambia FLAG3
        FLAGREG = xr0;;
        call retardo(ABS);;
        .align_code 4;
        rti (NP) (ABS);;

retardo:
        LC1 = 0x10000000;;

lazo:
        if NLC1E, jump lazo(NP);;
        NOP;;
        CJMP(NP);;
        ._main.END:
```

ANEXO 2

PROGRAMA DE UTILIZACIÓN DE ARCHIVOS EXTERNOS Y ACCESO HACIA Y DESDE MEMORIA

ARCHIVO pruebaarchivo.asm

```
//*********************************************************************
//  Código de Ejemplo de Utilización de archivos externos y acceso
//  hacia y desde memoria
//
//  pruebaarchivo.asm
//*********************************************************************

#define N 5       /*define el tamaño del vector*/

#include <defts201.h>

.section data1;
.var vector2[N]={4,3,2,1,2}; /* vector2 de datos */
.var vector1[N] = "data.txt";  /* vector1 de datos cargados desde un
                                  archivo externo */

//*********************************************************************

.SECTION program;
.extern _imprimir;              /* declaración de función externa */

.ALIGN_CODE 4;
.GLOBAL _main;
.global _vector3;
.var _vector3[N];               /* vector3 de resultados */

//*********************************************************************

_main:
    LC1=N;;                     /*carga n en el contador de lazo*/
    j4=1;;
    j1=vector1;;                /*obtiene la dirección de vector1*/
    j2=vector2;;                /*obtiene la dirección de vector2*/
    j3=_vector3;;               /*obtiene la dirección de vector3*/
```

```
_lazo:
     k1=[j31+j1];;              /*carga datos de vector1 en k1*/
     k2=[j31+j2];;              /*carga datos de vector2 en k2*/
     k3=k1+k2;;                 /*suma los elementos de los vectores*/
     xr3=k3;;                   /*carga el resultado de la suma en
                                  el registro xR3*/
     [j3+=j31]=xr3;;            /*carga el dato de xR3 en la dirección
                                  de memoria del vector3*/

     j1=j1+j4;;          /*incremento del puntero del vector1*/
     j2=j2+j4;;          /*incremento del puntero del vector2*/
     j3=j3+j4;;          /*incremento del puntero del vector3*/

if NLC1E, jump _lazo(NP);;
   call _imprimir;;      /*llama la funcion que imprime los
                                  resultados*/
   NOP;;
   CJMP(NP);;
._main.END:
```

ARCHIVO imprimir.c

```
//*********************************************************************
//  Código para imprimir los resultados obtenidos
//  imprimir.c
//*********************************************************************

#include <stdio.h>

extern int vector3[5];

  void imprimir()
  {
         int i;

         for (i=0;i<5;i++)
         {
                  printf("resultados [%d]= %d\n",i,vector3[i]);
          }
  }
```

ANEXO 3

PROGRAMA PARA RUTINA DE GRABACIÓN

ARCHIVO: *Grabacion.asm*

```
//*****************************************************************************
//      Código de Grabación para DSP A en el EZ-KIT TigerSHARC
//      Grabacion.asm
//
//*****************************************************************************

#include <defts201.h>

#define SER_ENBL_P      FLAGREG_FLAG3_OUT_P
#define CODEC           0x38000000
#define pass_through 0
#define play 1
#define record 2

//*****************************************************************************
.global _main;

.extern _InitA;
.extern _mode;

.section/NO_INIT sdram;
.align 4;
.var sdram_address;

//*****************************************************************************

.section program;
_main:

        call _InitA; nop; nop;;  //Llama a la rutina InitA

        YR30 = FLAGREG;;
        yr30 = bset r30 by FLAGREG_FLAG2_EN_P;;   //Habilita FLAG2 como  salida
        FLAGREG = yr30;;
        yr0=pass_through;;
        yr1=play;;
```

```
        yr2=record;;
        j0=j31+sdram_address;;      //dirección de inicio
        jb0=j31+sdram_address;;     //buffer circular empezará en 0, con longitud de 1
        jl0=0x1;;
        j1=j31+sdram_address;;              //dirección de parada
        jb2=j31+sdram_address;;

        jl2=0x99cf00;;                      //Tamaño de la SDRAM

        j2=j31+sdram_address;;


_pass:
        YR30 = FLAGREG;;
        yr30 = bCLR r30 by FLAGREG_FLAG2_OUT_P;;
        FLAGREG = yr30;;            //Apaga los leds
        j0=j31+sdram_address;;
        [j31+_mode]=yr0;;           //inicializa el estado del programa en passthrough

_play_or_record:
        if FLAG0_IN, jump _play_or_record;;
        if FLAG1_IN, jump _play_or_record;nop;nop;nop;;
        _play_or_record_bounce:
        if FLAG1_IN, jump _record; nop;nop;nop;;
        if FLAG0_IN, jump _play;nop;nop;nop;;

        jump _play_or_record_bounce;nop;nop;nop;; //Si ni play o grabación es
                                                  //presionado, permanece en
                                                  //estado passthrough.
_play:
        if FLAG0_IN, jump _play;nop;nop;nop;;
        _play_bounce:

        [j31+_mode]=yr1;;                   //fija el modo en play
        xr30 = FLAGREG;;
        xr30 = btgl r30 by FLAGREG_FLAG2_OUT_P;;   //Led 2 se enciende en forma
                                                   //intermitente para indicar
                                                   //estado de reproducción.
        FLAGREG = xr30;;
        LC0 = 10000000;;
        nop;; nop;; nop;; nop;;

        _toggle_loop:
        nop;; nop;; nop;; nop;;
        if NLC0E, jump _toggle_loop (NP);nop;nop;nop;;
```

```
        if FLAG0_IN, jump _pass; nop;nop;nop;;    //ahora el botón play funciona como
                                                  //botón de parada

        jump _play_bounce;nop;nop;nop;;          //el botón de grabacíón no hace nada

_record:
        YR30 = FLAGREG;;
        yr30 = bSET r30 by FLAGREG_FLAG2_OUT_P;;
        FLAGREG = yr30;;       //Se enciende el Led 2 para indicar estado de grabación
        if FLAG1_IN, jump _record;nop;nop;nop;;

        j2=j31+sdram_address;;
        _record_jump_bounce:

        [j31+_mode]=yr2;;                        //fija el modo en grabación
        if FLAG1_IN, jump _store; nop;nop;;
        jump _record_jump_bounce;nop;nop;nop;;

_store:
        //obtiene el final de la grabación para realizar el playback del buffer circular

        j1=j2-sdram_address;;         //j0 apunta al final del buffer después que se
                                      //detiene la grabación.
        jb0=j31+sdram_address;;
        jl0=j1;;
        j0=j31+sdram_address;;        //resetea el vector de reproducción para escucharlo
                                      //desde su inicio
        jump _pass;;

//*****************************************************************************
_wait:
        jump _wait (NP); nop; nop; nop;;
        CJMP(ABS)(NP);;
_main.end:
```

ARCHIVO: ***Interrupcion.asm***

```
/*****************************************************************************
    Rutina de Interrupción de audio para el EZ-Kit TigerSHARC
    Interrupcion.asm
*****************************************************************************/

#include <defTS201.h>
```

```
//******************** Declaraciones globales ****************************
.global _AudioInt;

//******************** Declaraciones externas ****************************
.extern _ReadDataLeft;
.extern _ReadDataRight;
.extern _WriteDataLeft;
.extern _WriteDataRight;
.extern _mode;


//************************ Sección del Programa ***************************
.section program;

_AudioInt:
        xr0=0x0;;
        xr1=0x1;;
        xr2=0x2;;

        xr31=[j31+_mode];; //esta es la bandera que indica el modo en que se encuentra
        xr3=r31-r0;;
        if AEQ, jump pass_stage;nop;nop;nop;;

        xr3=r31-r1;;
        if AEQ, jump play_stage;nop;nop;nop;;

        xr3=r31-r2;;
        if AEQ, jump record_stage;nop;nop;nop;;

        xr31=0x0;;
        jump _AudioInt;nop;nop;nop;;

_AudioInt.end:


//Etapa de Passthrough
pass_stage:

        XR14=0XFF;;
        xr4 = [j31 + _ReadDataLeft];;
        [j31 + _WriteDataLeft] = xr4;;
        xr5 = [j31 + _ReadDataRight];;
        [j31 + _WriteDataRight] = xr5;;
        jump end_of_int;nop;nop;nop;;
```

```
pass_stage.end:

//Etapa de Reproducción

play_stage:
        XR27=0XFF;;
        xr4=CB[j0+=0x1];;
        [j31 + _WriteDataLeft] = xr4;; //reproduce el contenido de sdram
        xr5 = [j31 + _ReadDataRight];;
        [j31 + _WriteDataRight] = xr5;;
        jump end_of_int;nop;nop;nop;;
play_stage.end:


//Etapa de Grabación

record_stage:
        XR29=0XFF;;
        xr4 = [j31 + _ReadDataLeft];;
        [j31 + _WriteDataLeft] = xr4;; //pass through
        CB[j2+=0x1]=xr4;; //Almacena en sdram.
                                //j2 está configurado para ser un buffer
                                //circular del mismo tamaño que SDRAM.

        xr5 = [j31 + _ReadDataRight];;
        [j31 + _WriteDataRight] = xr5;;
        jump end_of_int;nop;nop;nop;;
record_stage.end:

nop;nop;nop;nop;;
end_of_int:
nop;nop;nop;nop;;
rti (ABS)(NP);;
```

ARCHIVO: ***Init_Audio.asm***

```
/*****************************************************************************
    Rutina de inicialización para el DSPA del EZ-KIT TigerSHARC
    Init_Audio.asm
*****************************************************************************/

#include <defts201.h>
#define SER_ENBL_P      FLAGREG_FLAG3_OUT_P     // Posición del bit para
                                                // habilitar el Puerto Serial
```

```
//********************** Mapa de memoria ****************************

#define CODEC          0x38000000

//********************** Declaraciones Globales ************************

.global _ReadDataLeft, _ReadDataRight, _WriteDataLeft, _WriteDataRight;
.global _XmitDMALSourceTCB, _XmitDMALDestinTCB, _XmitDMARSourceTCB,
_XmitDMARDestinTCB;
.global _RcveDMALSourceTCB, _RcveDMALDestinTCB, _RcveDMARSourceTCB,
_RcveDMARDestinTCB;
.global _mode;
.global _InitA;

//********************** Declaraciones Externas ************************
.extern _AudioInt;

//******************** Transmisor y Receptor de Audio ********************
.section data1;
.align 4;
.var _WriteDataLeft;
.align 4;
.var _WriteDataRight;
.align 4;
.var _ReadDataLeft;
.align 4;
.var _ReadDataRight;
.align 4;
.var _mode;

//***************************** TCBs ***********************************
.section data1;
.align 4;
.var _RcveDMALSourceTCB[4] =   CODEC,
                               0x00010000,
                               0,
                               TCB_EXTMEM | TCB_NORMAL | TCB_DMAR
                                     | TCB_CHAIN | _XmitDMALSourceTCB >> 2;

.align 4;
.var _RcveDMALDestinTCB[4] =   _ReadDataLeft,
                               0x00010000,
                               0,
                               TCB_INTMEM | TCB_NORMAL | TCB_DMAR |
                                     TCB_CHAIN | _XmitDMALDestinTCB>> 2;
```

```
.align 4;
.var _XmitDMALSourceTCB[4] =   _WriteDataLeft,
                               0x00010000,
                               0,
                               TCB_INTMEM | TCB_NORMAL | TCB_INT |
                               TCB_CHAIN |_RcveDMARSourceTCB >> 2;
.align 4;
.var _XmitDMALDestinTCB[4] =   CODEC,
                               0x00010000,
                               0,
                               TCB_EXTMEM | TCB_NORMAL | TCB_INT |
                               TCB_CHAIN | _RcveDMARDestinTCB >> 2;
.align 4;
.var _RcveDMARSourceTCB[4] =   CODEC,
                               0x00010000,
                               0,
                               TCB_EXTMEM | TCB_NORMAL |TCB_CHAIN
                               | _XmitDMARSourceTCB >> 2;
.align 4;
.var _RcveDMARDestinTCB[4] =   _ReadDataRight,
                               0x00010000,
                               0,
                               TCB_INTMEM | TCB_NORMAL | TCB_CHAIN
                               | _XmitDMARDestinTCB >> 2;

.align 4;
.var _XmitDMARSourceTCB[4] =   _WriteDataRight,
                               0x00010000,
                               0,
                               TCB_INTMEM | TCB_NORMAL | TCB_CHAIN
                               | _RcveDMALSourceTCB >> 2;

.align 4;
.var _XmitDMARDestinTCB[4] =   CODEC,
                               0x00010000,
                               0,
                               TCB_EXTMEM | TCB_NORMAL |TCB_CHAIN
                               | _RcveDMALDestinTCB >> 2;


//****************** Sección del Programa ***************************
.section program;
_InitA:

// Inicializa SYSCON and SDRCON
```

```
        xr0 =   SYSCON_MP_WID64         | SYSCON_MEM_WID64|
                SYSCON_MSH_PIPE2        | SYSCON_MSH_WT0        |
                SYSCON_MSH_IDLE         | SYSCON_MS1_PIPE1      |
                SYSCON_MS1_WT0          | SYSCON_MS1_IDLE |
                SYSCON_MS0_SLOW         | SYSCON_MS0_WT3        |
                SYSCON_MS0_IDLE;;
        SYSCON = xr0;;

        xr0 =   SDRCON_INIT             | SDRCON_RAS2PC5 |
                SDRCON_PC2RAS2          | SDRCON_REF3700  |
                SDRCON_PG256            | SDRCON_CLAT2   | SDRCON_ENBL;;
        SDRCON = xr0;;

        xr0 = IMASKL;;
        xr0 = bset r0 by INT_DMA0_P;;           // habilita interrupción del DMA0
        IMASKL = xr0;;

// Habilita las interrupciones globales

        SQCTLST = SQCTL_GIE;;  // Habilita en SQCTL el bit para las interrupciones
                               // por hardware.
        xr0 = FLAGREG;;
        xr0 = bset r0 by FLAGREG_FLAG3_EN_P;;    // fija la bandera 3 como salida
        xr0 = bclr r0 by SER_ENBL_P;;            // fija el habilitador serial en cero
        FLAGREG = xr0;;

        j4 = j31 + _AudioInt;;                   // fija el vector de interrupción DMA
        IVDMA0 = j4;;
        xr2 = (2 << 8) | (0xf);;


//************************ Inicia los DMAs ******************************
        xr3:0 = Q[j31 + _RcveDMALSourceTCB];;// inicia la cadena de DMAs...
        xr7:4 = Q[j31 + _RcveDMALDestinTCB];; // ... iniciando...
        DCS0 = xr3:0;;
        DCD0 = xr7:4;;

//********************** Habilitador del Puerto Serial***********************
        xr0 = FLAGREG;;                          // fija el habilitador serial en uno
        xr0 = bset r0 by SER_ENBL_P;;
        FLAGREG = xr0;;

//*************************** Rutina de salida  ***************************
        CJMP(ABS)(NP);;
   _InitA.end:
```

ANEXO 4

PROGRAMA PARA RUTINA DE COMUNICACIÓN

ARCHIVO: *Consumer.cpp*

```
/*============================================================
 *
 *  Descripción: Esta es una implementación en C++ para la Cadena de Recepción
 *  del Consumidor (Receptor)
 *
 *============================================================*/

#include "Consumer.h"
#include <new>
#include <stdlib.h>
#include <stdio.h>
#include "../ProdCons.h"

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")

#define MSG_LIMIT 10       // Número total de mensajes a recibir

int datos_recibido[MSG_LIMIT];


/*****************************************************************
 *  Función local para manejar un ítem entrante
 */
 static void consume_item (void *item)
{
        static int expected = 1;
        static int x = 0;

        int *current = reinterpret_cast<int*>(item);
        printf("Dato %d Recibido del Productor = %d\n", expected, *current);

        datos_recibido[x]=*current;
        x++;
        expected++;

        VDK::Yield();
}
```

```
/*****************************************************************
 *  Función de Corrida del Consumidor (Función principal de recepción main{})
 */

void
Consumer::Run()
{
        // Envía mensajes VDK_kMaxNumActiveMessages vacíos
        for (int i = 0;  i < VDK_kMaxNumActiveMessages;  ++i)
        {
                // Inicializa y ubica el bloque de memoria en el campo de memoria
                void *mem_block = (void *) VDK::MallocBlock(kMarshalledMessages);
                strcpy((char *)mem_block,"empty block");

                // Crea un mensaje, fija la carga del mensaje y lo envía
                VDK::MessageID msg = VDK::CreateMessage(MSG_TYPE, 0, NULL);
                VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE, mem_block);
                VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
        }

        for(int i = 0; i < MSG_LIMIT; i++)
        {
                // Toma el mensaje que contiene un ítem
                VDK::MessageID msg = VDK::PendMessage(MSG_CHANNEL, 0);

                // Extrae el ítem del mensaje
                int type;
                unsigned int size;
                void *item;
                VDK::GetMessagePayload(msg, &type, &size, &item);

                // El ítem cumple un proceso determinado
                consume_item(item);

                if (i < MSG_LIMIT-1) // Si el límite de mensajes no ha sido alcanzado...
                {
                        // Regresa el mensaje para su reuso.
                        strcpy((char *)item,"empty block");
                        VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE, item);
                        VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
                }
                else // Si el límite de mensajes ha sido alcanzado...
      {

        // Fija la carga del mensaje para mostrar que este es el último mensaje
```

```
                VDK::SetMessagePayload(msg, END_MSG, 0, 0);

        // Envía el mensaje al Productor (Transmisor) para señalar el fin de la recepción
                VDK::PostMessage(kProducer1, msg, MSG_CHANNEL);
            }
        }
        printf("Recepcion Finalizada\n");
        exit(0);
}
```

ARCHIVO: *Consumer.h*

```
/*============================================================
 *
 *  Descripción: Esta es un archivo de cabecera C++ para la cadena de Recepción
 *  Consumer.h
 *
 *============================================================*/

#ifndef _Consumer_H_
#define _Consumer_H_

#pragma diag(push)
#pragma diag(suppress: 177,401,451,826,831,1462)

#include "VDK.h"

#pragma diag(pop)

class Consumer : public VDK::Thread
{

public:
   Consumer(VDK::Thread::ThreadCreationBlock&);
   virtual ~Consumer();
   virtual void Run();
   virtual int ErrorHandler();
   static VDK::Thread* Create(VDK::Thread::ThreadCreationBlock&);

};

#endif /* _Consumer_H_ */
```

ARCHIVO: ***Producer.cpp***

```
/* ============================================================
 *
 *  Descripción: Esta es una implementación en C++ para la cadena de Transmisión
 *  (Producer)
 *
 * ============================================================*/

#include "Producer.h"
#include <new>
#include <stdlib.h>
#include <stdio.h>
#include "../ProdCons.h"

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")

#define MSG_LIMITP 10     // Número total de mensajes para producir y guardar

int datos_transmitidos[MSG_LIMITP];
int w=0;

/*********************************************************************
 *  Función local para generar un ítem de salida
 */

static void produce_item(void **ppItem)
{
        int val = rand();
        *ppItem = reinterpret_cast<void*>(val);
}

/*********************************************************************
 *  Función de corrida de Transmisión (Función main{} de Producer)
 */

void
Producer::Run()
{
        bool produceMessages = true;

   while (produceMessages)
   {
                // Genera un ítem para colocarlo en el mensaje
```

```
                void *item;
                produce_item(&item);
                // Espera por un mensaje vacío para colocar el ítem
                VDK::MessageID msg = VDK::PendMessage(MSG_CHANNEL, 0);

                // Averigua de donde viene el mensaje
                VDK::MsgChannel channel;
                VDK::ThreadID sender;
                VDK::GetMessageReceiveInfo(msg, &channel, &sender);

                // Obtiene la carga del mensaje
                int type;
                unsigned int size;
                void *mem_block;
                VDK::GetMessagePayload(msg, &type, &size, &mem_block);

                if(type != END_MSG)
                {
                        // Resetea este espacio de memoria y copia el siguiente ítem en él.
                        memset(mem_block, 0, MBSIZE);
                        memcpy(mem_block, &item, sizeof(int));

                // Configura y envía el nuevo ítem
                VDK::SetMessagePayload(msg, MSG_TYPE, MBSIZE, mem_block);
                VDK::PostMessage(sender, msg, MSG_CHANNEL);

                }
                else
                        produceMessages = false;
   }

   printf("Transmision Finalizada\n");
   exit(0);
}

/*********************************************************************
 *  Manejo de error del Transmisor (Producer)
 */

int
Producer::ErrorHandler()
{
   /* La función ErrorHandler (llamada abajo) termina la cadena*/
   return (VDK::Thread::ErrorHandler());
}
```

ARCHIVO: *Producer.h*

```
/* ============================================================
 *
 * Descripción: Esta es un archivo de cabecera C++ para la cadena de Transmisión
 *
 * ============================================================*/

#ifndef _Producer_H_
#define _Producer_H_

#pragma diag(push)
#pragma diag(suppress: 177,401,451,826,831,1462)

#include "VDK.h"
#pragma diag(pop)

class Producer : public VDK::Thread
{
public:
   Producer(VDK::Thread::ThreadCreationBlock&);
   virtual ~Producer();
   virtual void Run();
   virtual int ErrorHandler();
   static VDK::Thread* Create(VDK::Thread::ThreadCreationBlock&);
};

#endif /* _Producer_H_ */

/* ============================================================*/
```

ANEXO 5

DESARROLLO DE PRÁCTICA DE LABORATORIO No.1
TRANSFORMADA DISCRETA DE FOURIER

PRÁCTICA No. 1

TEMA: Transformada discreta de Fourier

1. INTRODUCCIÓN

Para la realización de esta práctica, en base a la tarjeta EZ-KIT Lite ADSP-TS201, se diseñó previamente un código en lenguaje ensamblador para la implementación de un filtro digital tipo FIR. Este código implementado deberá ser usado durante toda la práctica para realizar los procedimientos de filtrado solicitados.

2. OBJETIVO

- Analizar el proceso matemático realizado para el cálculo de la FFT.
- Realizar un algoritmo con código en lenguaje C que realice el proceso matemático para el cálculo de la FFT, y proporcione resultados bajo ciertas condiciones.
- Realizar la optimización del programa para simplifique el cálculo de la FFT.

3. EQUIPOS

- Tarjeta EZ-KIT Lite ADSZ-TS201 Familia TigerSHARC.
- PC con sistema operativo Windows 2000 o Vista
- Visual DSP++ 5.0.

4. PROCEDIMIENTO GENERAL

4.1 CONEXIÓN DE HARDWARE

a) Conectar la tarjeta EZKIT-Lite a la PC.

b) Conectar la tarjeta EZKIT-Lite a la fuente de poder.

4.2 SOFTWARE

Realizar el siguiente procedimiento para efectuar los cálculos de la FFT sobre las siguientes señales de entrada:

- Caso I: señal de 64 bits reales.
- Caso II: señal de 32 bits complejos.

CASO I: 64 BITS REALES

EJECUCIÓN APLICACIÓN DEL PROGRAMA PARA EL CÁLCULO DE LA FFT REAL DE RAÍZ 2

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto ***FFT_FLP32.dpj*** que se encuentra en la carpeta FFT como se realizo en el proceso del Capitulo IV ítem 4.2.2.

c) Abrir dentro del proyecto el archivo FFTDef.h, ya que aquí se encuentran definidas las constantes *N* y *FFT_Real* que son las constantes que guardan el tamaño de los datos de entrada y el tipo de FFT respectivamente (Figura A.1).

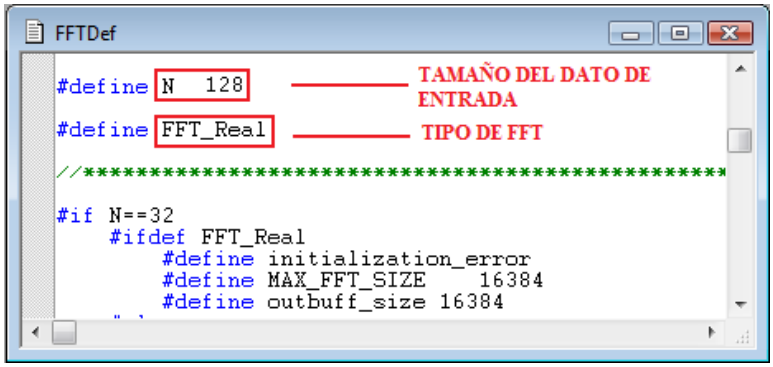

Figura A. 1. Archivo FFTDef.h de VisualDSP++.

d) Fijar el tamaño de datos de entrada en 64 bits, definiendo a la variable N con el valor de 64 como se indica en la Figura A.1.

e) Fijar el tipo de FFT en *REAL*, *"descomentando"* la instrucción ("sin //") que contiene la definición de la constante FFT_Real como se indica en la Figura A.2.

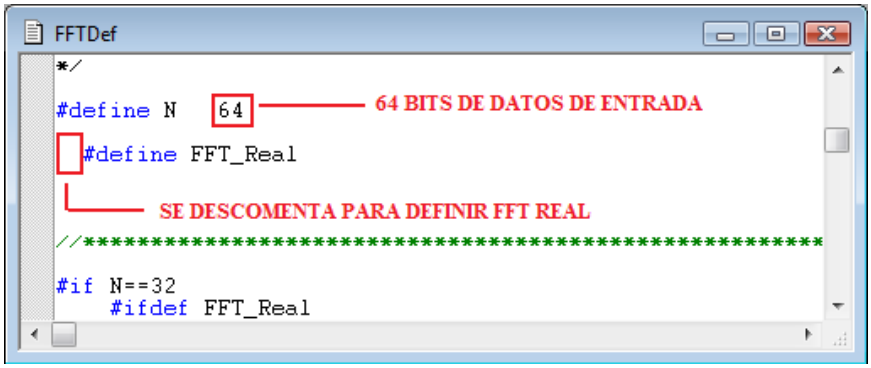

Figura A. 2. Definición de FFT real.

f) Finalmente compilar y ejecutar el proyecto, observar el resultado que se obtiene en la ventana de salida de VisualDSP++ (Figura A.3).

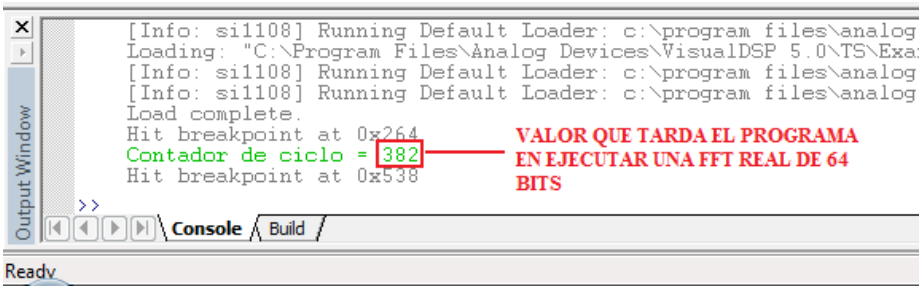

Figura A. 3. Ventana de salida FFT real.

g) Anotar el valor que se muestra en la ventana de salida, ya que servirá para realizar el análisis final de la práctica.

Al emplear una FFT Compleja de N datos, en realidad la estructura de la FFT está realizando la operación sobre 2*N datos, ya que los datos complejos están constitutitos por una parte real y una imaginaria. Por lo tanto, los análisis y comparaciones que se vayan a realizar en base a los dos tipos de FFT deberán hacerse empleando siempre en la FFT Compleja como número de datos de entrada, la mitad del número de datos de entrada de la FFT Real.

CASO II: 32 BITS COMPLEJOS

EJECUCIÓN DEL PROGRAMA PARA EL CÁLCULO DE LA FFT COMPLEJA DE RAÍZ 2

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto ***FFT_FLP32.dpj*** que se encuentra en la carpeta FFT como se realizo en el proceso del Capitulo IV ítem 4.2.2.

c) Abrir dentro del proyecto el archivo FFTDef.h (Figura A.1)

d) Fijar el tamaño de datos de entrada en 32 bits, definiendo a la variable N con el valor de 32 como se indica en la Figura A.4.

e) Fijar el tipo de FFT en *COMPLEJA*, *"comentando"* la instrucción (" // ") que contiene la definición de la constante FFT_Real como se indica en la Figura A.4.

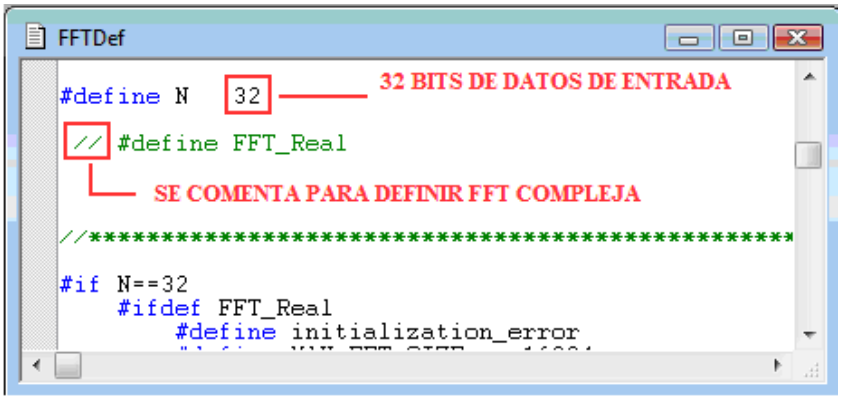

Figura A. 4. Definición de FFT compleja.

f) Finalmente a compilar y ejecutar el proyecto, observar el resultado que se obtiene en la ventana de salida de VisualDSP++ (Figura A.5).

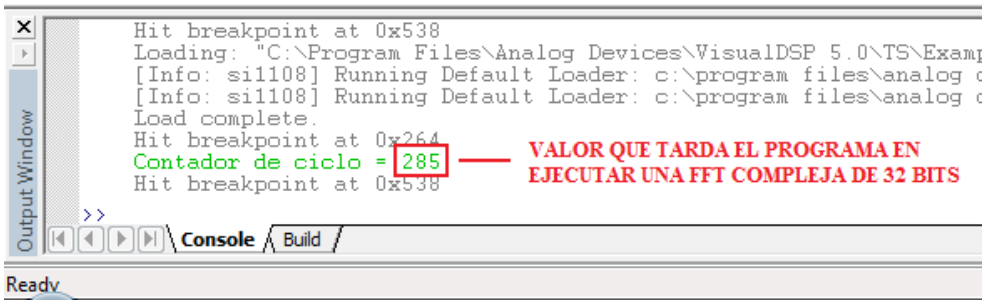

Figura A. 5. Ventana de salida FFT compleja.

g) Anotar el valor que se muestra en la ventana de salida, ya que servirá para realizar el análisis final de la práctica.

EJERCICIO I

ESTRUCTURA CONVENCIONAL

1. De acuerdo a su criterio y sin emplear el programa de cálculo de la FFT responda las siguientes preguntas:

a) ¿Qué cálculo toma más tiempo en ejecutarse: Una FFT Compleja de N datos o una FFT Real de 2*N datos? ¿Explique, Por qué?

b) ¿Considera que el programa implementado toma a todos los datos de entrada, sean estos reales o complejos, siempre como datos complejos? Si su respuesta es afirmativa mencione ¿cómo el programa diferencia entre ambos tipos de datos para realizar el cálculo? Si su respuesta es negativa, mencione ¿cómo consigue el programa realizar el cálculo de los dos tipos de FFT empleando las misma etapas?

c) Proponer un método de cálculo para convertir el contador de ciclos del programa en segundos. ¿Qué criterios y factores se debe tomar en cuenta para dicho cálculo?

2. Realizar los pasos del los casos I y II del ítem 4.2 del procedimiento general para realizar los siguientes casos:

a) FFT Compleja con entrada mayor que 128 y menor que 4096 datos.
b) FFT Real con entrada con el doble de datos que la FFT Compleja del paso anterior.

3. En base a los resultados obtenidos en el paso anterior realizar lo siguiente:
a) Comparar los resultados obtenidos. ¿Cuál es mayor?
b) Juzgar ¿explique por qué si se emplea cuantitativamente el mismo número de datos, los resultados difieren?
c) Graficar los datos obtenidos empleando la herramienta Window Plot de VisualDSP++.

EJERCICIO II

ESTRUCTURA OPTIMIZADA

1. De acuerdo a su criterio y sin emplear el programa de cálculo de la FFT responda las siguientes preguntas:

a) Considerando que la optimización de la estructura convencional de la FFT consiste en realizar cálculos paralelos en las etapas, ¿en qué porcentaje cree usted que mejorará el cálculo de la FFT?
b) Detallar un método que utilizaría para conseguir una optimización en la estructura convencional de la FFT.

2. Realizar los pasos del los ítems 4.2.1 y 4.2.2 del procedimiento general para realizar los siguientes casos:

a) FFT Compleja con entrada mayor que 4096 y menor que 32768 datos.
b) FFT Real con entrada con el doble de datos que la FFT Compleja del paso anterior.

3. En base a los conocimientos adquiridos de VisualDSP++, realice los siguientes pasos que le permitirá alcanzar una optimización de la estructura convencional de la FFT.

a) Detecte dentro del archivo fft_flp32,asm del proyecto las dos primeras etapas del cálculo de la FFT.
b) Cree un nuevo archivo .asm y ubique las dos etapas mencionadas en dicho archivo. El objetivo de la optimización está en ejecutar por separado las dos primeras etapas.
c) Enlace las dos etapas del nuevo archivo .asm con el resto del programa que ejecuta el cálculo.
d) Compile el programa.

4. Con la estructura optimizada realice los mismos pasos explicados en el procedimiento para las mismas FFT Compleja y Real empleadas en el punto 2.

5. En base a los resultados obtenidos en el punto 2 y punto 4 responda las siguientes preguntas:

a. Comparar los resultados obtenidos para cada tipo de FFT. ¿Cuál es mayor en cada una de ellas?
b. ¿Mejoró el desempeño de la FFT empleando la estructura optimizada? Explique ¿por qué?

c. Calcular cuantitativamente la mejora en rendimiento que tuvo la estructura optimizada con respecto a la estructura convencional.

EJERCICIO III

1. Tomar otro valor de número de datos de entrada para la FFT Compleja y su correspondiente valor para datos de entrada para la FFT Real y repetir los pasos 2, 4 y 5 del Ejercicio II.

ANEXO 6

DESARROLLO DE PRÁCTICA DE LABORATORIO No. 2

FILTROS FIR

PRÁCTICA No. 2

TEMA: Filtros FIR

1. INTRODUCCIÓN

Para la realización de esta práctica, en base a la tarjeta EZ-KIT Lite ADSP-TS201s, se diseñó previamente un código en lenguaje ensamblador para la implementación de un filtro digital tipo FIR. Este código implementado deberá ser usado durante toda la práctica para realizar los procedimientos de filtrado solicitados.

2. OBJETIVOS

- Familiarizar al alumno con la utilización de un filtro tipo FIR implementado con la tarjeta EZ-KIT Lite ADSP-TS201s.
- Realizar el filtraje de un conjunto de muestras mediante el filtro FIR implementado y comprobar su funcionamiento en tiempo real.
- Verificar la influencia que tiene el orden del filtro en la respuesta en frecuencia obtenida.
- Analizar los resultados obtenidos.

3. EQUIPOS

- Tarjeta EZ-KIT Lite ADSZ-TS201 Familia TigerSHARC
- Generador de señales (0 – 25 MHz)
- Osciloscopio
- 2 Cables de transmisión de datos (BNC hembra a Jack de 3.5 mm.)
- Una PC con sistema operativo Windows 2000 o Vista.
- Herramienta de programación Visual DSP++ 5.0
- Software MATLAB.

4. PROCEDIMIENTO GENERAL

4.1 CONEXIÓN DE HARDWARE

a) Conectar la tarjeta EZKIT-Lite a la PC. Como se indica en la Figura A.6.

b) Configurar el generador de ondas con los siguientes parámetros:

- Onda senoidal
- Onda con 100 mV_{PP}
- Frecuencia de 10 KHz

c) Conectar el extremo BNC del primer cable al generador de ondas y el otro extremo al canal de entrada de audio de la tarjeta (LINE IN) como si indica en la Figura A.6.

d) Conectar el extremo BNC del segundo cable al osciloscopio y el otro extremo al canal de salida de audio de la tarjeta (LINE OUT) como si indica en la Figura A.6.

e) Conectar la tarjeta EZKIT-Lite a la fuente de poder.

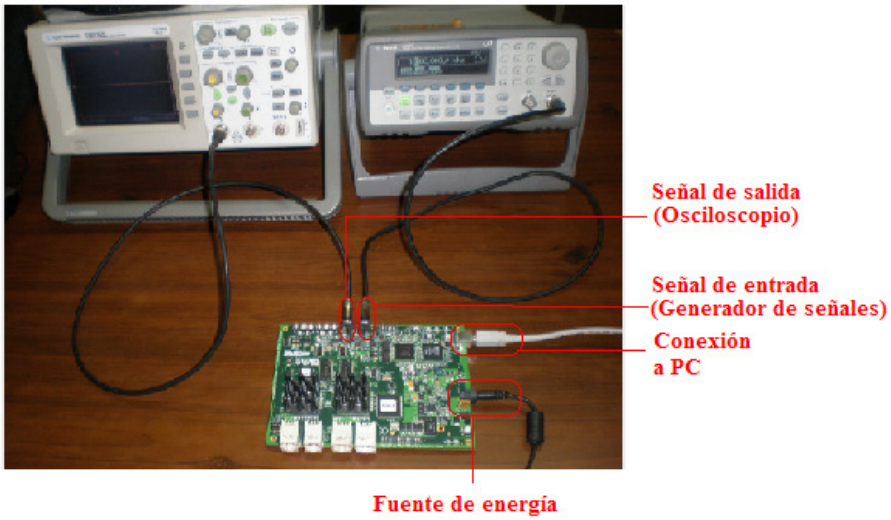

Figura A. 6. Conexión de la tarjeta con los equipos utilizados

4.2 SOFTWARE

Realizar los siguientes pasos para el diseño un filtro FIR pasa bajos:

4.2.1 OBTENCIÓN DE COEFICIENTES DEL FILTRO

a) Abrir el programa MATLAB.

b) Dar clic izquierdo en el botón Start. Seleccionar *ToolBoxes*, después la opción *Filter Design* y por ultimo dar clic en *Filter Design & Análisis Tool* (Figura A.7).

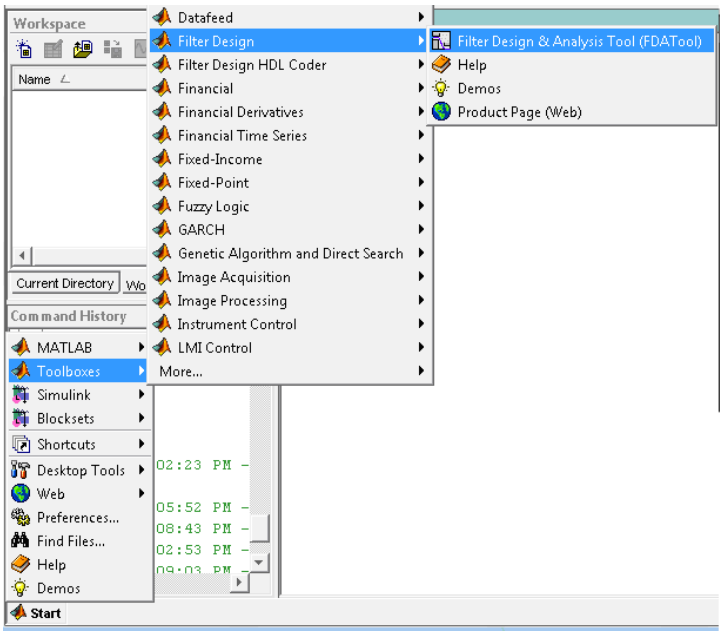

Figura A. 7. Pasos para abrir Filter Design *& Análisis Tool*.

c) Aparecerá una ventana en la cual se selecciona el tipo del filtro FIR y se escribe los valores de las especificaciones del filtro (Figura A.8).

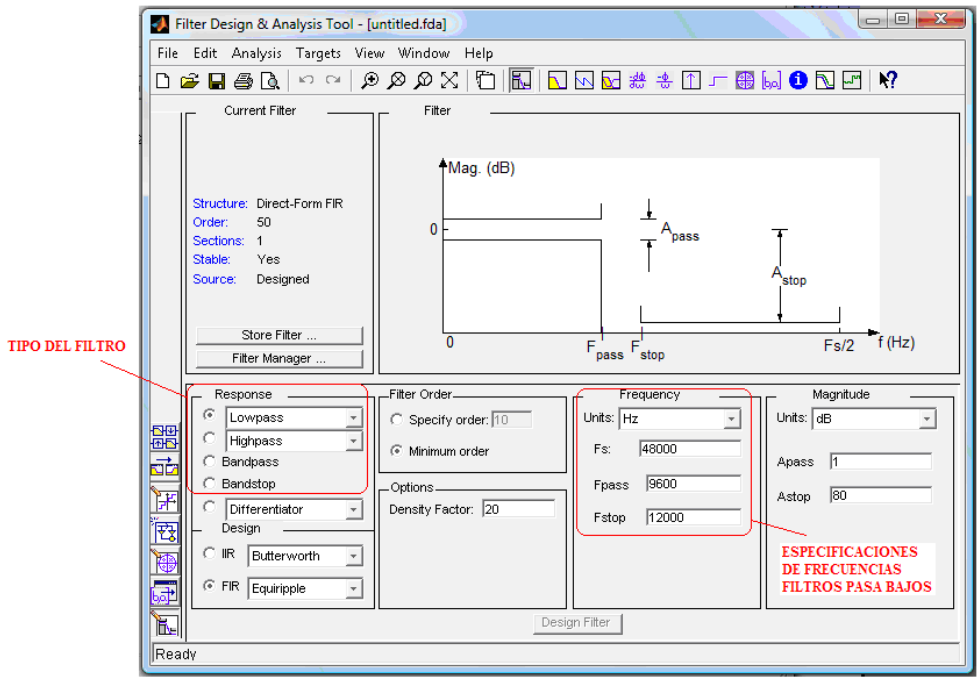

Figura A. 8. Ventana de Filter Design *& Análisis Tool* para el diseño de filtros FIR.

Para el diseño de cualquier tipo de filtro FIR se selecciona el tipo de filtro como se muestra en la Figura A.8.

Las especificaciones de frecuencias (Figura A.9) cambian según el tipo de Filtro FIR, la figura 7.8 muestra las especificaciones según el tipo de filtro ha ser diseñado.

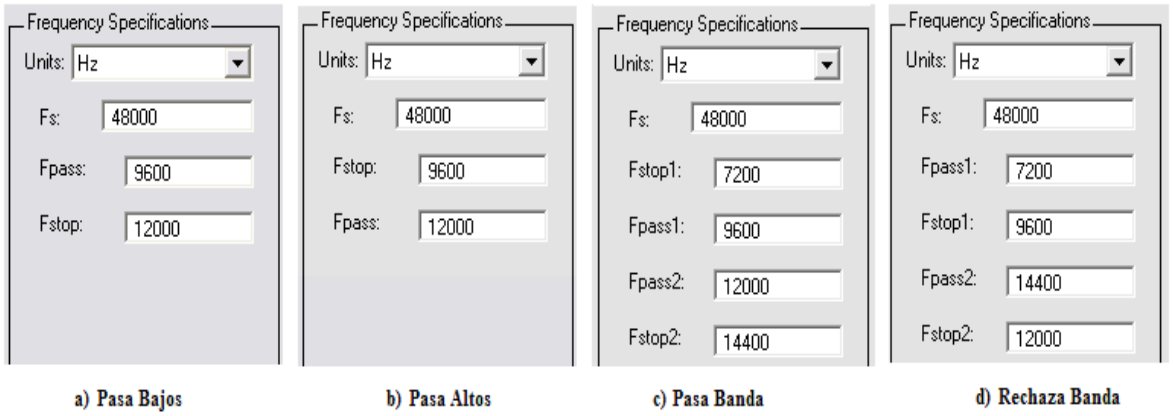

Figura A. 9. Especificaciones de frecuencia para los tipos de filtros FIR.

Para el diseño del filtro FIR para bajos, mantener fijos los siguientes parámetros:

- Response Type: Lowpass
- Design method: FIR (Equirpple)
- Frecuency:

  Units : Hz

  Fs (frecuencia de muestreo): 48000

En un filtro FIR Pasa Bajos, las frecuencias a cambiar para el diseño se muestran en la Figura A.9.

La frecuencia de corte (Fstop), la frecuencia de paso (Fpass) y el orden del filtro se configurarán de acuerdo a lo solicitado durante la práctica.

d) Una vez configurados todos los parámetros del filtro que se desea diseñar, dar clic en *Design Filter* (Figura A.10). MATLAB calculará la respuesta impulsiva del filtro que también corresponden a los coeficientes del filtro FIR deseado.

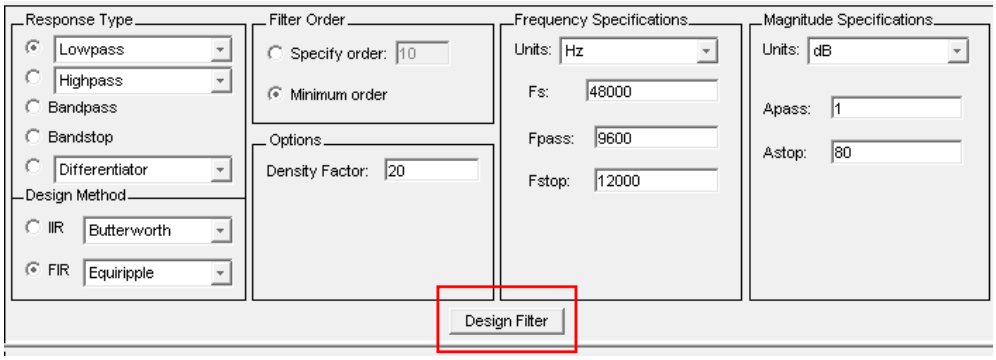

Figura A. 10. Botón para iniciar el diseño del filtro

e) Posteriormente, aún dentro de la ventana de *Filter Design* dar clic en el menú *File*, Seleccionar y dar clic en *Export*.

f) Aparecerá una ventana en la cual se puede exportar una variable con los valores de los coeficientes al *Workspace* de MATLAB, damos clic en *OK*.

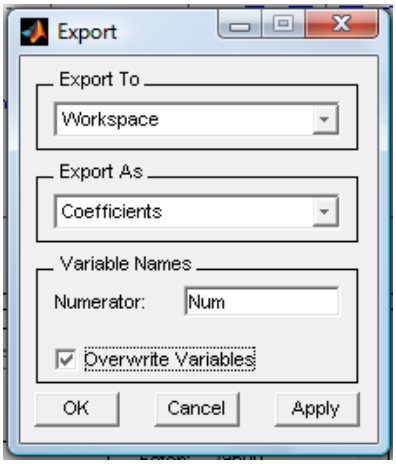

Figura A. 11. Cuadro para exportar variables.

g) Una vez exportados los coeficientes al *Workspace*, abrir el archivo coef_impFIR.m adjunto en la carpeta de Prácticas FIR. Este archivo sirve para crear un archivo *.dat* con los coeficientes del filtro diseñado que fueron exportados al *workspace*.

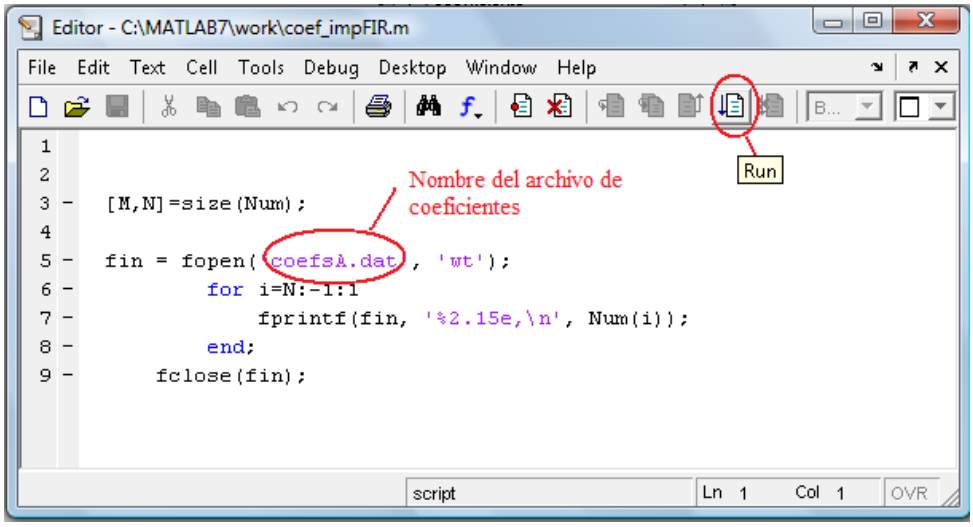

Figura A. 12. Archivo coef_impFIR.m para crear un archivo de coeficientes .dat

h) Asignar un nombre al archivo de coeficientes y ejecutar la aplicación con la pestaña que se indica en la figura A.11 para crear dicho archivo. Este archivo se creará en la carpeta *work* de MATLAB (C:\MATLAB7\work).

i) Obtenido el archivo de coeficientes del filtro FIR pasa bajos, copiar dicho archivo de la carpeta *work*. pegar el archivo en la carpeta que contiene el proyecto del diseño de filtros FIR, para que pueda ser compilado dentro del proyecto.

4.2.2 EJECUCIÓN DE LA APLICACIÓN DEL FILTRO FIR PASA BAJOS

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto *FIR.dpj* que se encuentra en la carpeta *FIR* como se realizó en el proceso del Capitulo IV ítem 4.2.2.

c) Dentro del proyecto, en el archivo *FIR.asm* se encuentra definida la variable *.var coefs[TAPS]* que es la que contiene el valor de los coeficientes (Figura A.12). Se debe cambiar el nombre del archivo asignado a dicha variable por el archivo que contiene los coeficientes del diseño actual.

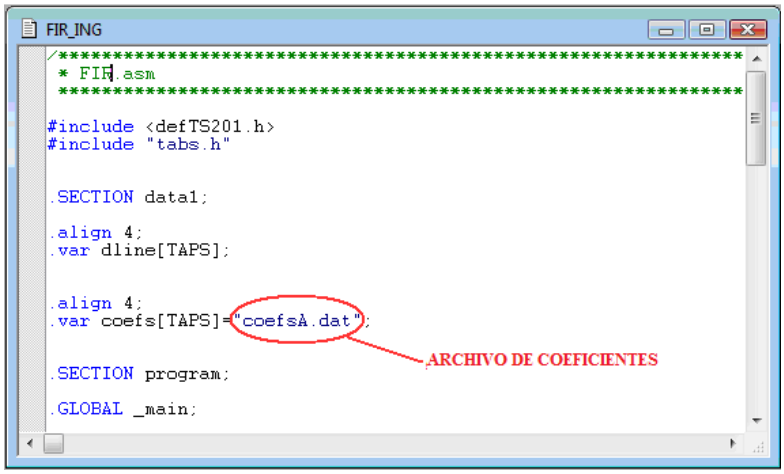

Figura A. 13. Archivo FIR.asm de Visual DSP++.

d) En el archivo *tabs.h,* proceder a cambiar el parámetro que guarda el orden del filtro (Figura A.13) de acuerdo al orden empleado en el diseño.

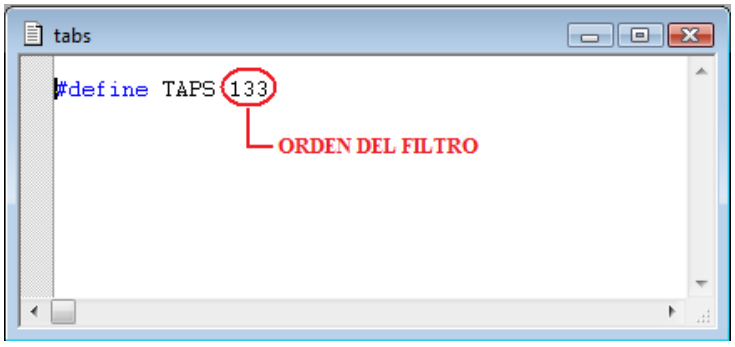

Figura A. 14. Archivo tabs.h que contiene declaración de variable.

e) Finalmente compilar y ejecutar el proyecto.

5. PARTE EXPERIMENTAL

PRÁCTICA No. 2A

FILTRO FIR PASA BAJOS

EJERCICIO I

1. Diseñar un filtro digital FIR pasa bajos con una frecuencia de paso de 13 KHz y una frecuencia de corte de 14 KHz.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el orden del filtro en 120.
3. Seleccionar una señal senoidal con frecuencia de 10000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro FIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 13 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Del comportamiento observado que puede decir de la frecuencia de muestro que está utilizando el programa de filtro FIR?
8. Incremente la frecuencia a 14 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros FIR realice lo siguiente:

Diseñar dos filtros FIR pasa bajos de las siguientes características:

- Frecuencia de paso de 13 KHz, una frecuencia de corte de 14 KHz. y el orden del filtro en 150.
- Frecuencia de paso de 13 KHz, una frecuencia de corte de 14 KHz. y el orden del filtro en 60.

Responder las siguientes preguntas:

- Comparar y explicar el desempeño de los filtros, en base a los resultados obtenidos, ¿en qué influye el orden del filtro dentro del diseño de un filtro FIR?
- De acuerdo a su criterio, ¿cuáles son los beneficios y cuáles los inconvenientes de usar un filtro de mayor orden?
- En base a que la frecuencia de muestreo de los filtros es de 48 KHz; según su criterio, ¿en qué rangos de frecuencias sería recomendable realizar un diseño de filtro?

PRÁCTICA No. 2B

FILTRO FIR PASA ALTOS

EJERCICIO I

1. Diseñar un filtro digital FIR pasa altos con una frecuencia de corte de 20 KHz y una frecuencia de paso de 21 KHz.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el orden del filtro en 120.
3. Seleccionar una señal senoidal con frecuencia de 10000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro FIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 20 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Incremente la frecuencia a 21 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros FIR realice lo siguiente:

Diseñar dos filtros FIR pasa altos de las siguientes características:

- Frecuencia de corte de 20 KHz, una frecuencia de paso de 21 KHz. y el orden del filtro en 150.
- Frecuencia de corte de 20 KHz, una frecuencia de paso de 21 KHz. y el orden del filtro en 60.

PRÁCTICA No. 2C

FILTRO FIR PASA BANDA

EJERCICIO I

1. Diseñar un filtro digital FIR pasa banda con las siguientes características:

- Frecuencia de corte 1 de 13 KHz.
- Frecuencia de paso 1 de 14 KHz.
- Frecuencia de paso 2 de 18 KHz.
- Frecuencia de corte 2 de 19 KHz.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el orden del filtro en 120.
3. Seleccionar una señal senoidal con frecuencia de 10000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro FIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 14 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Incremente la frecuencia a 19 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros FIR realice lo siguiente:

Diseñar dos filtros FIR pasa banda con las especificaciones de frecuencia del ítem 1 del ejercicio I, con la modificación de las siguientes características:

- Orden del filtro en 150.
- Orden del filtro en 60.

PRÁCTICA No. 2D

FILTRO FIR RECHAZA BANDA

EJERCICIO I

1. Diseñar un filtro digital FIR rechaza banda con las siguientes características:

- Frecuencia de paso 1 de 12 KHz.
- Frecuencia de corte 1 de 13 KHz.
- Frecuencia de corte 2 de 17 KHz.
- Frecuencia de paso 2 de 18 KHz.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el orden del filtro en 120.
3. Seleccionar una señal senoidal con frecuencia de 9000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro FIR, observar la salida del filtro que se muestra en el osciloscopio.

5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 12 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Incremente la frecuencia a 18 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros FIR realice lo siguiente:

Diseñar dos filtros FIR rechaza banda con las especificaciones de frecuencia del ítem 1 del ejercicio I, con la modificación de las siguientes características:

- Orden del filtro en 150.
- Orden del filtro en 60.

ANEXO 7

DESARROLLO DE PRÁCTICA DE LABORATORIO No. 3

FILTROS IIR

PRÁCTICA No. 3

TEMA: Filtros IIR

1. INTRODUCCIÓN

Para la realización de esta práctica, en base a la tarjeta EZ-KIT Lite ADSP-TS201s, se diseñó previamente un código en lenguaje ensamblador para la implementación de un filtro digital tipo IIR. Este código implementado deberá ser usado durante toda la práctica para realizar los procedimientos de filtrado solicitados.

2. OBJETIVO

- Familiarizar al alumno con la utilización de un filtro tipo IIR implementado con la tarjeta EZ-KIT Lite ADSP-TS201s.
- Filtrar un conjunto de muestras mediante el filtro IIR implementado y comprobar su funcionamiento en tiempo real.
- Mostrar la influencia que tiene el orden del filtro en la respuesta en frecuencia obtenida.
- Analizar los resultados obtenidos.

3. EQUIPOS

- Tarjeta EZ-KIT Lite ADSZ-TS201 Familia TigerSHARC
- Generador de señales
- Osciloscopio
- 2 Cables de transmisión de datos (BNC hembra a Jack de 3.5 mm.)
- Una PC con sistema operativo Windows 2000 o Vista.
- Herramienta de programación Visual DSP++ 5.0
- Software MATLAB.

4. PROCEDIMIENTO GENERAL

4.1 CONEXIÓN DE HARDWARE

a) Conectar la tarjeta EZKIT-Lite a la PC como ya se ha realizado anteriormente.

b) Configurar el generador de ondas con los siguientes parámetros:

- Onda senoidal
- Onda con 100 mV_{PP}
- Frecuencia de 10 KHz

c) Conectar el extremo BNC del primer cable al generador de ondas y el otro extremo al canal de entrada de audio de la tarjeta (LINE IN) como si indica en la figura:

d) Conectar el extremo BNC del segundo cable al osciloscopio y el otro extremo al canal de salida de audio de la tarjeta (LINE OUT) como si indica en la figura:

e) Conectar la tarjeta EZKIT-Lite a la fuente de poder.

4.2 SOFTWARE

Realizar los siguientes pasos para el diseño un filtro IIR pasa bajos:

4.2.1 OBTENCIÓN DE COEFICIENTES DEL FILTRO

a) Abrir el programa MATLAB.

b) Abrir el cuadro de *Filter Design & Análisis Tool*, como se realizó en las Prácticas de filtros FIR en el item 4.2.1.

c) En el cuadro *Filter Design & Análisis Tool* seleccionar el diseño de filtros IIR, en el cual se escribe los valores de las especificaciones del filtro (Figura A.14).

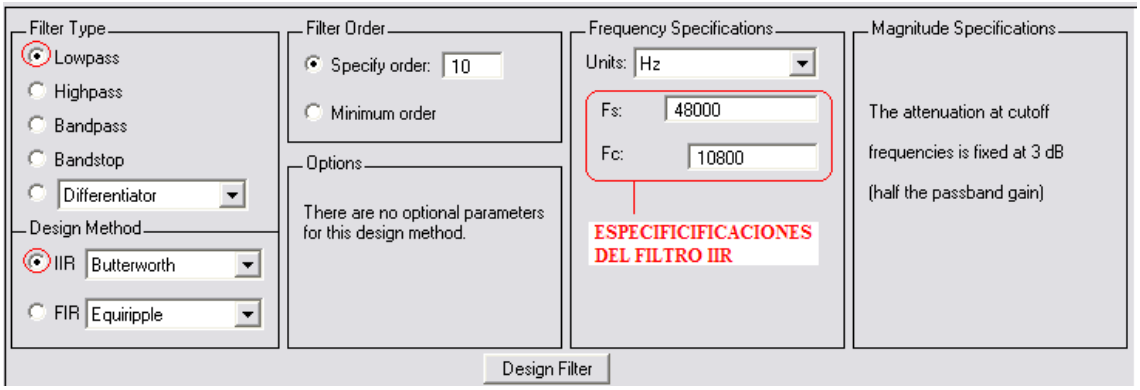

Figura A. 15. Ventana de Filter Design *& Análisis Tool* para el diseño de filtros IIR

Las especificaciones de frecuencias (Figura A.15) cambian según el tipo de Filtro IIR, la Figura A.16 muestra las especificaciones según el tipo de filtro ha ser diseñado.

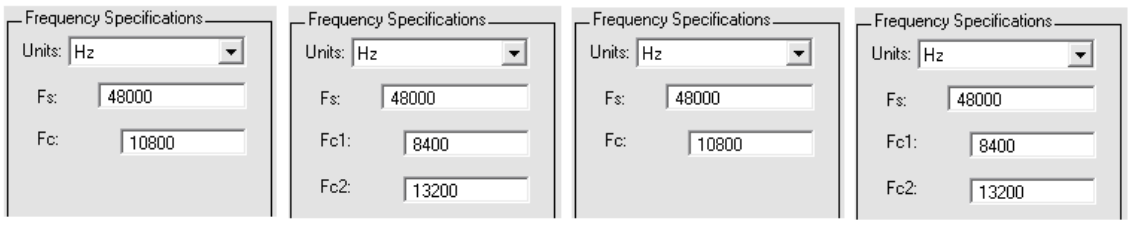

Figura A. 16. Especificaciones de frecuencia para los tipos de filtros IIR.

Para el diseño del filtro IIR pasa bajos mantener fijas los siguientes parámetros:

- Response Type: Lowpass
- Design Method: IIR (Butterworth)
- Frecuency:

  Units : Hz

  Fs (frecuencia de muestreo): 48000

La frecuencia de corte (Fc) y el orden del filtro se configuran de acuerdo a lo solicitado durante la práctica.

d) Una vez configurados todos los parámetros del filtro que se desea diseñar, dar clic en *Design Filter*. MATLAB calculará la respuesta impulsiva del filtro que también corresponden a los coeficientes del filtro IIR deseado.

e) Posteriormente, aún dentro de la ventana de *Filter Design* dar clic en el menú *File*, Seleccionar y dar clic en *Export* como se realizó en el ítem 4.2.1 de la obtención de coeficientes del filtro de la Práctica de filtros FIR.

f) Una vez exportados los coeficientes y las ganancias al *Workspace*, abrimos el archivo coef_impIIR.m adjunto en la carpeta de Prácticas IIR. Este archivo sirve para crear un archivo *.dat* con los coeficientes y las ganancias del filtro diseñado que fueron exportados al *Workspace*.

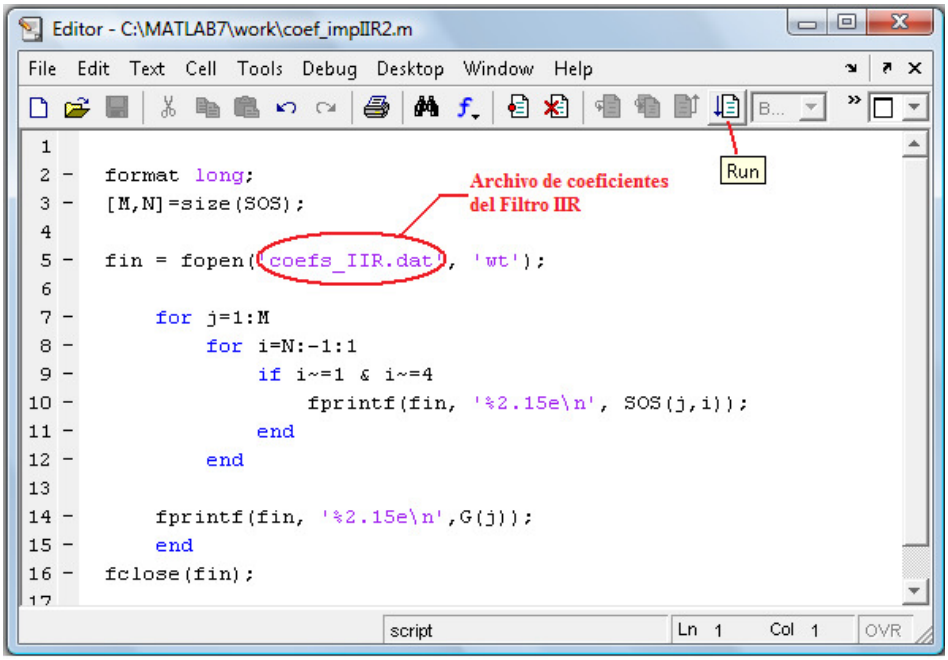

Figura A. 17. Archivo coef_impIIR.m para crear un archivo de coeficientes .dat

g) Asignar un nombre al archivo de coeficientes y ejecutar la aplicación con la pestaña que se indica en la Figura A.17 para crear dicho archivo. Este archivo se creará en la carpeta *work* de MATLAB (C:\MATLAB7\work).

h) Obtenido el archivo de coeficientes y ganancias del filtro IIR pasa bajos, copiar dicho archivo de la carpeta *work*. Una vez copiado, pegar el archivo en la carpeta que contiene el proyecto del diseño de filtros IIR, para que pueda ser compilado dentro del proyecto.

4.2.2 CORRIDA DE APLICACIÓN DEL FILTRO IIR PASA BAJOS

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto *IIR.dpj* que se encuentra en la carpeta *IIR* como se realizo en el proceso del Capitulo IV ítem 4.2.2

c) Dentro del proyecto, en el archivo *IIR.asm* se encuentra definida la variable *.var coefs[SECTIONS*5]* que contiene el valor de los coeficientes (Figura A.18). Se debe cambiar el nombre del archivo asignado a dicha variable el archivo que contiene los coeficientes del diseño actual.

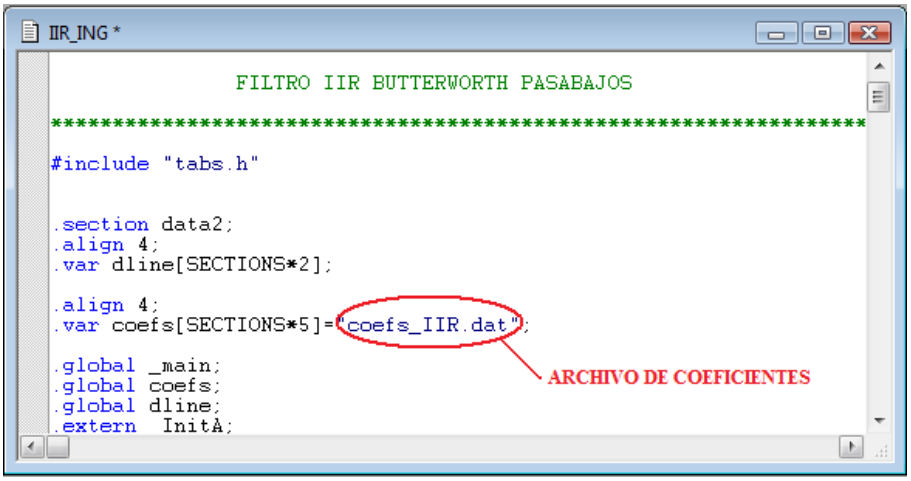

Figura A. 18. Archivo FIR.asm de Visual DSP++.

d) En el archivo *tabs.h,* proceder a cambiar la constante SECTIONS (Secciones del Filtro) de acuerdo al orden del filtro (Figura A.19). El número de secciones del filtro es igual a la mitad del valor del orden del filtro que se utilizó para el diseño.

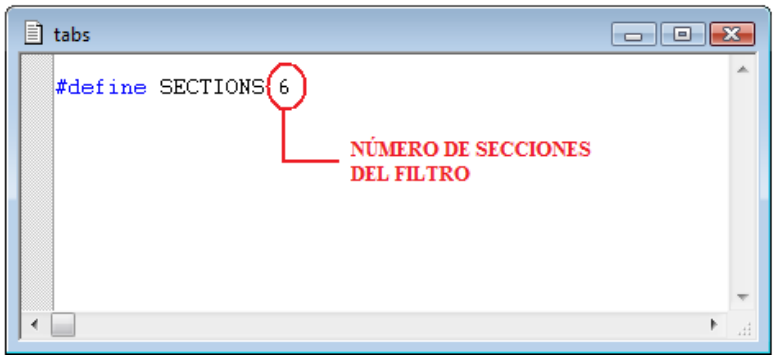

Figura A. 19. Archivo tabs.h que contiene declaración de variable.

e) Finalmente compilar y ejecutar el proyecto.

5. PARTE EXPERIMENTAL

PRÁCTICA No. 3A

FILTRO IIR PASA BAJOS

EJERCICIO I

1. Diseñar un filtro digital IIR pasa bajos con una frecuencia de corte de 14.5 KHz y una frecuencia muestreo de 48 KHz y orden del filtro en 120.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el número de secciones en 60.
3. Seleccionar una señal senoidal con frecuencia de 10000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro FIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 14.5 KHz y seguir con el incremento lentamente. Explique que sucede?

7. Del comportamiento observado que puede decir de la frecuencia de muestro que está utilizando el programa de filtro FIR?
8. Incremente la frecuencia a 16 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros IIR realice lo siguiente:

Diseñar dos filtros IIR pasa bajos de las siguientes características:

- Frecuencia de corte de 14.5 KHz, una frecuencia de muestreo de 48 KHz. y el orden del filtro en 150.
- Frecuencia de paso de 13 KHz, una frecuencia de corte de 14 KHz. y el orden del filtro en 60.

Responder las siguientes preguntas:

- Comparar y explicar el desempeño de los filtros, en base a los resultados obtenidos, ¿en qué influye el orden del filtro dentro del diseño de un filtro FIR?
- De acuerdo a su criterio, ¿cuáles son los beneficios y cuáles los inconvenientes de usar un filtro de mayor orden?
- En base a que la frecuencia de muestreo de los filtros es de 48 KHz; según su criterio, ¿en qué rangos de frecuencias sería recomendable realizar un diseño de filtro?

PRÁCTICA No. 3B

FILTRO IIR PASA ALTOS

EJERCICIO I

1. Diseñar un filtro digital IIR pasa altos con una frecuencia de corte de 18 KHz y una frecuencia de muestreo de 48 KHz y orden del filtro de 120.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el número de secciones en 60.
3. Seleccionar una señal senoidal con frecuencia de 13000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro IIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 18 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Incremente la frecuencia a 20 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros IIR realice lo siguiente:

Diseñar dos filtros IIR pasa altos de las siguientes características:

- Frecuencia de corte de 18 KHz, una frecuencia de muestreo de 48 KHz y el orden del filtro en 150.

- Frecuencia de corte de 18 KHz, una frecuencia de muestreo de 48 KHz y el orden del filtro en 60.

PRÁCTICA No. 3C

FILTRO IIR PASA BANDA

EJERCICIO I

1. Diseñar un filtro digital IIR pasa banda con las siguientes características:

- Frecuencia de corte 1 de 12.5 KHz.
- Frecuencia de corte 2 de 18.5 KHz.
- Frecuencia de muestreo de 48 KHz.
- Orden del filtro 120.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el número de secciones en 60.
3. Seleccionar una señal senoidal con frecuencia de 10000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro IIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 12.5 KHz y seguir con el incremento lentamente. Explique que sucede?

7. Incremente la frecuencia a 19 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros IIR realice lo siguiente:

Diseñar dos filtros IIR pasa banda con las especificaciones de frecuencia del ítem 1 del ejercicio I, con la modificación de las siguientes características:

- Orden del filtro en 150.
- Orden del filtro en 60.

PRÁCTICA No. 3D

FILTRO IIR RECHAZA BANDA

EJERCICIO I

1. Diseñar un filtro digital IIR rechaza banda con las siguientes características:

    - Frecuencia de corte 1 de 15 KHz.
    - Frecuencia de corte 2 de 18.5 KHz.
    - Frecuencia de muestro de 48 KHz.
    - Orden del filtro en 120.

Realizar el procedimiento descrito en el ítem 4.2.2 con las siguientes características para verificar su funcionamiento:

2. Fijar el número de secciones en 60.
3. Seleccionar una señal senoidal con frecuencia de 9000 Hz en el generador de señal y una tensión pico a pico de 100 mV.
4. Iniciar el programa del filtro IIR, observar la salida del filtro que se muestra en el osciloscopio.
5. Incrementar paulatinamente la frecuencia de la señal de entrada. ¿Explique que observa?
6. Incrementar la frecuencia hasta llegar a 14 KHz y seguir con el incremento lentamente. Explique que sucede?
7. Incremente la frecuencia a 16 KHz, capture la señal de salida y justifique porque se obtiene ese resultado?

EJERCICIO II

Utilizando el procedimiento general de los filtros IIR realice lo siguiente:

Diseñar dos filtros IIR rechaza banda con las especificaciones de frecuencia del ítem 1 del ejercicio I, con la modificación de las siguientes características:

- Orden del filtro en 150.
- Orden del filtro en 60.

ANEXO 8

DESARROLLO DE PRÁCTICA DE LABORATORIO No. 4

FILTROS ADAPTATIVOS

PRÁCTICA No 4

TEMA: Filtros Adaptativos

1. INTRODUCCIÓN

Para la realización de esta práctica, en base a la tarjeta EZ-KIT Lite ADSP-TS201, se diseñó previamente un código en lenguaje emsamblador para la implementación de filtros adaptativos LMS. Este código implementado deberá ser usado durante toda la práctica para realizar los procedimientos de filtrado solicitados.

2. OBJETIVO

- Familiarizar al alumno con la utilización de un filtro adaptativo implementado con la tarjeta EZ-KIT Lite ADSP-TS201.
- Realizar el filtraje de un conjunto de muestras mediante los filtros LMS y NLMS implementados y comprobar su funcionamiento.
- Identificar los parámetros que controlan la velocidad de convergencia del algoritmo LMS y NLMS.
- Realizar la comparación de la eficiencia computacional de los algoritmos de filtros adaptativos LMS vs. NLMS.
- Analizar los resultados obtenidos.

3. EQUIPOS

- Tarjeta EZ-KIT Lite ADSZ-TS201 Familia TigerSHARC
- Una PC con sistema operativo Windows 2000 o Vista, que tenga instalado los programas Visual DSP++ 5.0 y MATLAB

4. PROCEDIMIENTO GENERAL

4. 1 CONEXIÓN DE HARDWARE

a) Conectar la tarjeta EZKIT-Lite a la PC como ya se ha realizado anteriormente.
b) Conectar la tarjeta EZKIT-Lite a la fuente de poder.

4. 2 SOFTWARE

Para la implementación del filtro adaptativo es necesario crear dos señales, la primera señal es la señal deseada y la segunda es la señal de entrada del filtro adaptativo. La señal de entrada del filtro es una mezcla de la señal deseada con una señal de ruido.
Las señales generadas deben ser muestreadas a 48 Khz.

Realizar los siguientes pasos para diseñar un filtro adaptativo LMS:

a) Abrir el programa MATLAB.
b) Abrir el archivo *Gen_Senales.m* adjunto en la carpeta de Prácticas LMS. En este archivo se genera una escala de tiempo para los datos, desde t = 0 a t = 5 ms con pasos de 1/48000 s. Generar una señal x que contenga ondas seno 1000 Hz (Figura A.20). Adicionar a la señal generada x algún ruido aleatorio con una desviación estándar de 2 para producir una señal de entrada con ruido, ejecutar el programa.

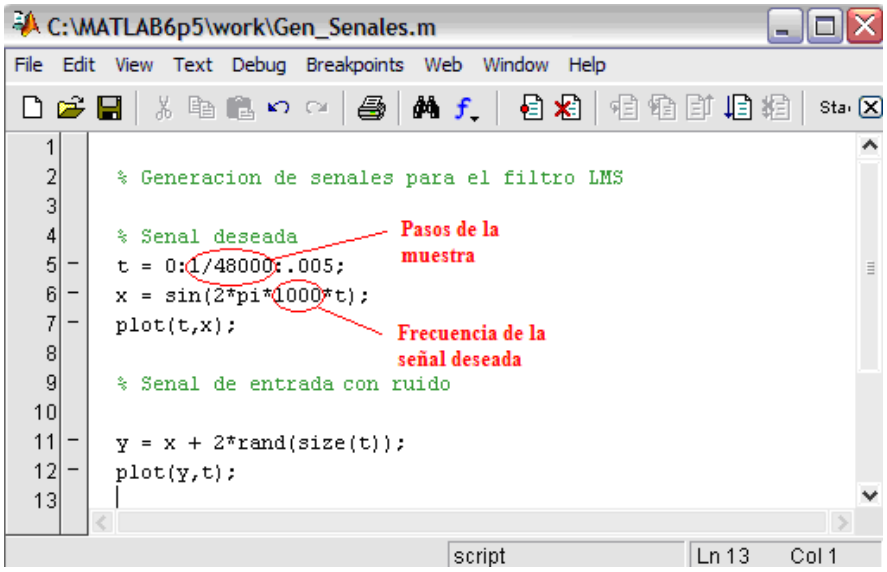

Figura A. 20. Archivo de Matlab Gen_Senales

c) Abrir el archivo *Senales_LMS.m* adjunto en la carpeta de Prácticas LMS. Ejecutar el programa para que genere un archivo .dat que contiene los valores de las señales generadas como se muestra en la Figura A.21.

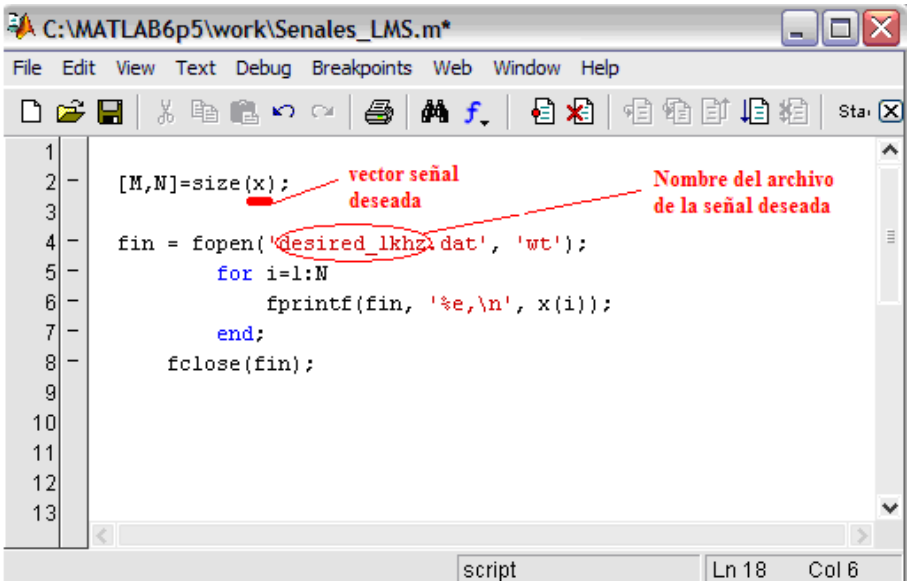

Figura A. 21. Archivo de Matlab Senales_LMS.m

d) Repetir el ***Paso c)*** del procedimiento de software, modificar el código de la Figura A.21 del vector de la señal de entrada "x" por el vector "y", cambiar el nombre del archivo *desired_khz.dat* por *input_1khz.dat*. Ejecutar el programa. Estos archivos se crearán en la carpeta *work* de MATLAB (C:\MATLAB6p5\work).

e) Generados los archivos de las señales para el filtro LMS, copiar dichos archivos de la carpeta *work*, pegar los archivos en la carpeta que contiene el proyecto del diseño de filtros LMS, para que pueda ser compilado dentro del proyecto.

4.3 MODIFICACIONES EN EL PROGRAMA IMPLEMENTADO PREVIO A SU COMPILACIÓN.

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto *LMS.dpj* que se encuentra en la carpeta *LMS* como se realizo en el proceso del Capitulo IV ítem 4.2.2.

c) Abrir el archivo *LMS.asm***,** se encuentran definidas las variables *TAPS* y *STEPSIZE* como se indica en la Figura A.22.

d) En el archivo *LMS.asm* cambiar el nombre del archivo que contiene los datos actuales de la señal de entrada y la señal deseada según el caso como indica la Figura A.22.

.var indata[TAPS]= **"input1khz.dat";**

.var desired[TAPS]= **"desired1khz.dat";**

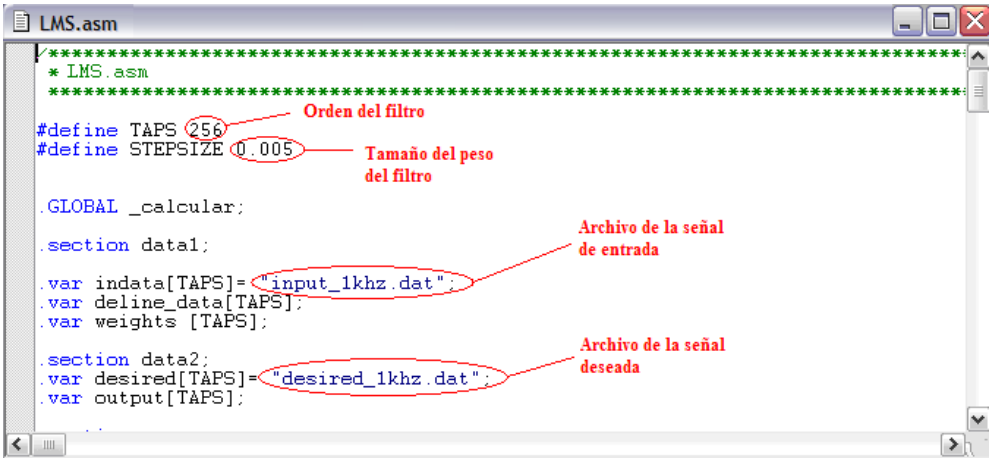

Figura A. 22. Parámetros modificables del Archivo LMS.asm

e) Modificar el parámetro *TAPS* que guarda el orden del filtro de acuerdo al orden empleado en el diseño (es igual al número de muestras).

f) Cambiar el valor del parámetro *STEPSIZE* el cuál especifica el tamaño del peso del filtro.

g) Proceder finalmente a compilar y ejecutar el proyecto.

4.4 VISUALIZACIÓN DE RESULTADOS

a) Detener la ejecución del programa.

b) En el menú *View*, seleccionar *Debug Windows*, *Plot* y dar clic en *New*. Aparecerá la ventana de configuración de Plot como se muestra en la Figura A.23.

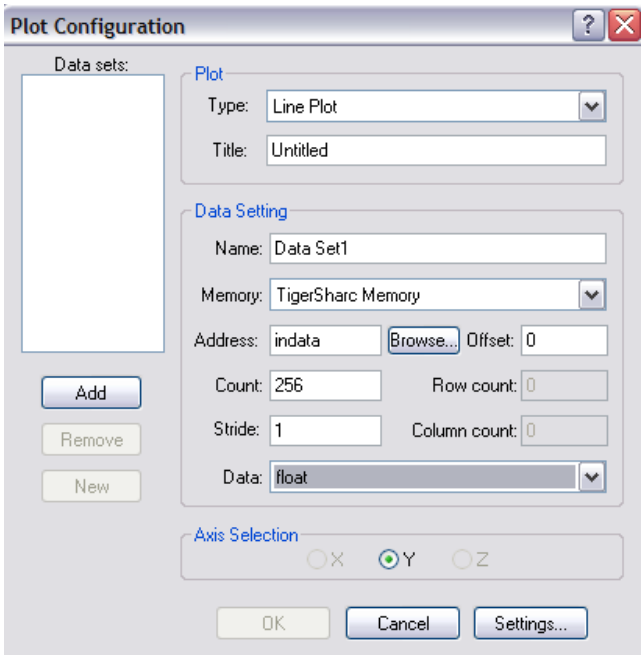

Figura A. 23. Ventana de configuración de Plot

c) La herramienta gráfica Plot debe ser configurada de la siguiente manera:

- En *Address* se debe buscar y colocar la variable *indata* de la señal de entrada.
- El valor de *Count* debe ser igual al número de muestras u orden del filtro.
- En *Data* seleccionar el tipo de dato *Float*.
- Dar clic en *Add*.

d) Repetir el paso c) de visualización de resultados, para las variables *desired* y *output*. Dar clic en OK. Se presenta el gráfico de las señales de entrada y salida del filtro como se muestra en la Figura A.24.

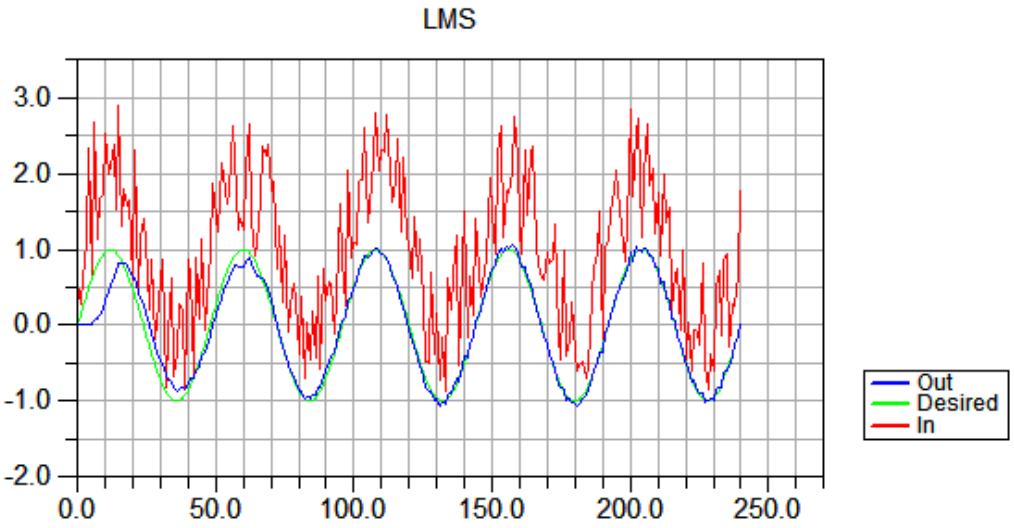

Figura A. 24. Gráfico con las señales de entrada y salida del filtro.

5. PARTE EXPERIMENTAL

PRÁCTICA No. 4A

FILTRO ADAPTATIVO LMS

EJERCICIO I

Utilizando el procedimiento general de filtros adaptativos realizar los siguientes pasos:

1. Diseñar un filtro digital LMS, con una señal deseada de tipo senoidal de frecuencia igual a 500 Hz y t = 5 ms.
2. Fijar el orden del filtro en 241.
3. Inicializar el programa del filtro LMS como el procedimiento 4.3, observar los resultados de la salida del filtro configurando la herramienta Plot como indica el ítem 4.4, capturar la salida.
4. Cambiar el valor del tamaño del peso del filtro por 0.007. Explique que sucede con la salida del filtro.
5. Cambiar el valor del tamaño del peso del filtro por 0.001. Comparar con los resultados obtenidos en el paso 3 del ejercicio 1.

Utilizando los pasos del procedimiento general de filtros adaptativos realizar el Ejercicio II.

EJERCICIO II

1. Diseñar un filtro LMS, con una señal deseada de tipo senoidal de f = 1500 Hz con t = 5 (s), tamaño del peso de 0.005. Capturar la salida de los resultados del filtro.
2. Colocar el valor del tamaño del peso del filtro en 0.008, después cambiarlo por 0.001. Analizar la comparación de los resultados de las salidas del filtro.

EJERCICIO III

1. Diseñar un filtro LMS, con una señal deseada de tipo senoidal de f = 3000 Hz con t = 5 (s), tamaño del filtro de 0.005. Capturar la salida de los resultados del filtro.

Responder las siguientes preguntas:

- Cuales son los efectos de trabajar con señales de mayor frecuencia.
- De los resultados obtenidos entre la señal de 500 Hz y 3000 Hz, el tiempo en que el filtro se demora en llegar a la convergencia de la solución de que factores depende?

PRÁCTICA No. 4B

FILTRO ADAPTATIVO NLMS

EJERCICIO I

Utilizando el procedimiento general de filtros adaptativos realizar los siguientes pasos:

1. Diseñar un filtro digital NLMS, con una señal deseada de tipo senoidal de frecuencia igual a 500 Hz y t = 5 ms.
2. Fijar el orden del filtro en 241.
3. Inicializar el programa del filtro NLMS como en el procedimiento 4.3, observar los resultados de la salida del filtro configurando herramienta Plot como indica el item 4.4, capturar la salida.
4. Cambiar el valor del tamaño del peso del filtro por 0.01. Explique que sucede con la salida del filtro?
5. Cambiar los valores de los parámetros GAMMA y ALPHA por 0.01. Realizar una comparación con los resultados obtenidos en el paso 3 del ejercicio I.
6. Cambie el valor de los parámetros GAMMA y ALPHA por 0.5. Realizar una comparación con los resultados obtenidos en el paso 3 del ejercicio I.

EJERCICIO II

Utilizando el procedimiento general de filtros adaptativos realizar el EJERCICIO II.

Diseñar dos filtros NLMS con las siguientes características:

- Señal deseada de tipo senoidal de f = 1500 Hz y t = 5 ms, TAPS = 241, STEPSIZE= 0.005, GAMMA y ALPHA de 0.5.
- Señal deseada de tipo senoidal de f = 3000 Hz y t = 5 ms. TAPS = 241, STEPSIZE= 0.005, GAMMA y ALPHA de 0.5.

CUESTIONARIO

Responder las siguientes preguntas:

- Describa las diferencias de los resultados obtenidos entre la señal de 1500 Hz y 3000 Hz.
- Analice las diferencias entre el algoritmo LMS y NLMS.
- Según lo expuesto cual algoritmo es más eficiente computacionalmente?
- Cuales son los parámetros que controlan el tiempo de convergencia en el filtro NLMS?

ANEXO 9

DESARROLLO DE PRÁCTICA DE LABORATORIO No. 5

ALGORTIMOS PARA SISTEMAS DISCRETOS

PRÁCTICA No. 5

TEMA: Algoritmos para sistemas discretos

1. INTRODUCCIÓN

En la presente práctica a través del sistema de Procesamiento Digital de Señales de la tarjeta EZ-KITLite ADSP-TS201s vamos a realizar dos tipos de algoritmos sobre las muestras de una señal analógica de entrada digitalizada. Los dos algoritmos a los que nos referiremos en particular son:

- Algoritmo de eco.
- Algoritmo de reverberación.

2. OBJETIVO

- Analizar los dos tipos de algoritmos, eco y reverberación, que se aplicarán sobre las muestras de una señal analógica de entrada digitalizada.
- Conocer la naturaleza del eco como algoritmo de sistema discreto.
- Conocer la naturaleza de la reverberación como algoritmo de sistema discreto.
- Determinar las propiedades que se presentan sobre las señales en ambos tipos de algoritmos.

3. EQUIPOS

- Tarjeta EZ-KIT Lite ADSZ-TS201 Familia TigerSHARC.
- PC con sistema operativo Windows 2000 o Vista
- Visual DSP++ 5.0.
- Micrófono
- Parlantes o audífonos

4. PROCEDIMIENTO GENERAL

4.1 CONEXIÓN DE HARDWARE

a) Conectar el micrófono a la entrada de audio (LINE IN) de la tarjeta EZKIT-Lite.
b) Conectar los parlantes o audífonos a la salida de audio (LINE OUT) de la tarjeta EZKIT-Lite.
c) Conectar la tarjeta EZKIT-Lite a la PC.
d) Conectar la tarjeta EZKIT-Lite a la fuente de poder.

4.2 SOFTWARE

4.2.1 PROGRAMA DE IMPLEMENTACIÓN DE ECO.

a) Abrir el programa VisualDSP++ 5.0.
b) Cargar el proyecto *DELAY.dpj* que se encuentra en la carpeta *ECO* como se realizó en el proceso del Capitulo IV ítem 4.2.2.
c) Proceder a compilar y ejecutar el proyecto.
d) Hablar a través del microfóno y escuchar la señal de salida a través de los parlantes o audífonos.

e) Anotar las características de la señal de salida.

f) Detener la ejecución del programa,

g) Abrir el archivo *Audio.asm*. En este archivo encontrará "comentada" la instrucción que contiene la definición de la macro *DELAY* (Figura A.25).

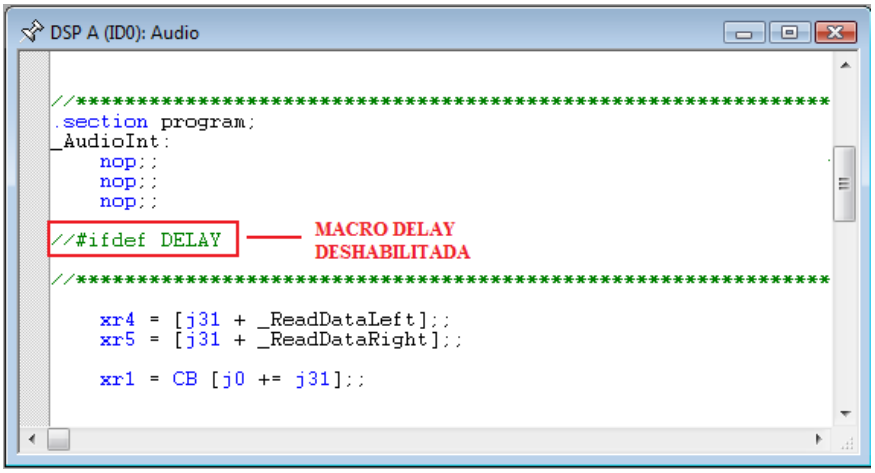

Figura A. 25. Macro DELAY deshabilitada

h) Habilitar la macro *DELAY* "descomentando" la línea de instrucción. Esto permitirá que el algoritmo que contiene el eco sea incluido (Figura A.26).

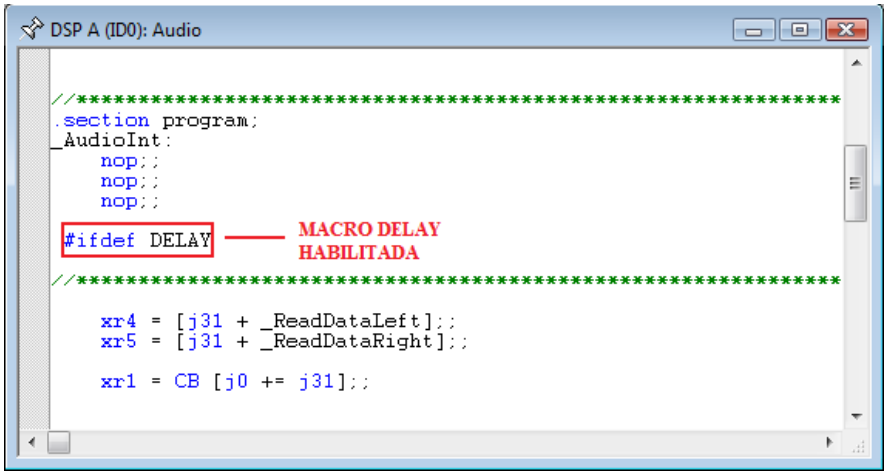

Figura A. 26. Macro DELAY habilitada

i) Guardar los cambios realizados.

j) Proceder a compilar y ejecutar el proyecto.

k) Hablar nuevamente a través del micrófono y escuchar la nueva señal de salida a través de los parlantes o audífonos.

l) Anotar las características de la actual señal de salida.

m) Detener la ejecución del programa.

4.2.2 PROGRAMA DE IMPLEMENTACIÓN DE REVERBERACIÓN.

a) Abrir el programa VisualDSP++ 5.0.

b) Cargar el proyecto *REVERB.dpj* que se encuentra en la carpeta *REVERBERACIÓN* como se realizó en el proceso del Capitulo IV ítem 4.2.2.

c) Abrir el archivo *Reverb.h*. En este archivo, establecer el nivel de volumen deseado (Figura A.27).

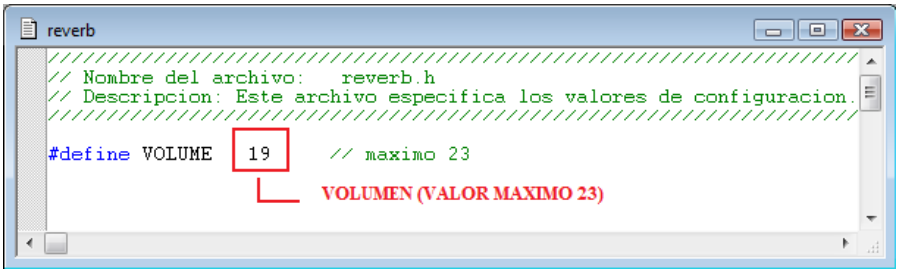

Figura A. 27. Configuración de nivel de volumen

d) Proceder a compilar proyecto.

e) Cargar los ejecutables en su respectivo procesador como se muestra en la Figura A.28.

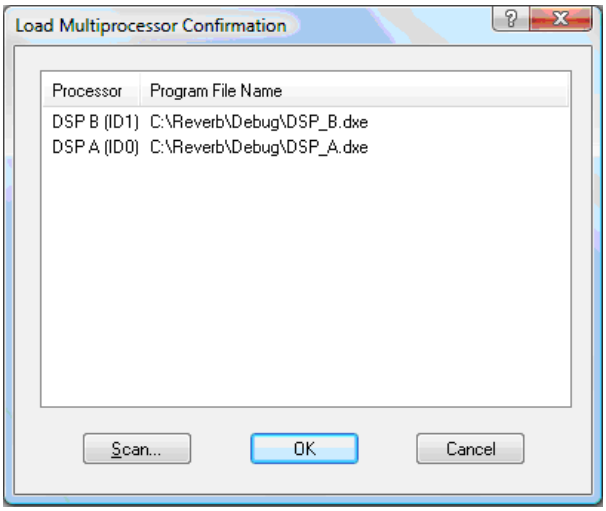

Figura A. 28. Ventana para cargar los archivos ejecutables

f) Esperar un tiempo hasta que Visual DSP termine de cargar el proyecto.

g) Una vez cargado el proyecto, ejecutarlo en forma de multiprocesamiento como se indica en la Figura A.29.

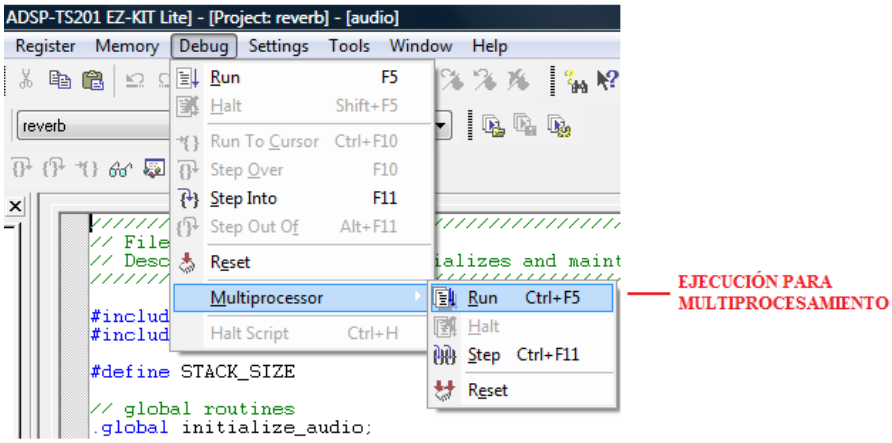

Figura A. 29. Ejecución para Multiprocesamiento

h) Hablar a través del micrófono y escuchar la señal de salida a través de los parlantes o audífonos.

i) Anotar las características de la señal de salida, repitiendo este proceso tantas veces como sea necesario para captar todas las características que posee la señal.

j) Oprimir el botón IRQ_A en la tarjeta EZ-KIT Lite. El LED4 está encendido cuando la reverberación está habilitada. El LED4 está apagado cuando la reverberación está deshabilitada.

k) Hablar nuevamente a través del micrófono y escuchar la nueva señal de salida a través de los parlantes o audífonos.

l) Anotar las características de la actual señal de salida.

m) Detener la ejecución del programa.

5. PARTE EXPERIMENTAL

PRÁCTICA No. 5A

ALGORITMO DE ECO PARA SISTEMAS DISCRETOS

CUESTIONARIO

En base al procedimiento general para generar eco como algoritmo de sistema discreto responder las siguientes preguntas:

1. ¿Qué características tiene la primera señal que se escucha?
2. ¿Qué características tiene la segunda señal que se escucha?
3. ¿Cuáles son las diferencias que encuentra entre cada una de las dos señales generadas en el procedimiento 4.2.1.?
4. Según su criterio ¿qué produce este algoritmo discreto de eco sobre la señal de entrada?.

EJERCICIO I

1. Abrir el archivo *Audio.asm* y buscar la parte del programa que se muestra en la Figura A.25.

2. Descomentar las líneas de código que se encuentran comentadas para incluirlas dentro del código.
3. Comente nuevamente en el archivo *Audio.asm* la línea de instrucción que contiene la definición de la macro *DELAY*.
4. Guarde los cambios realizados.
5. Compile y ejecute el programa.
6. Realice el mismo procedimiento explicado en el ítem 4.2.1.
7. Resuelve el cuestionario nuevamente en base a las nuevas señales obtenidas con los cambios realizados.

PRÁCTICA No. 5B

ALGORITMO DE REVERBERACIÓN PARA SISTEMAS DISCRETOS.

CUESTIONARIO

En base al procedimiento general para generar reverberación como algoritmo de sistema discreto responder las siguientes preguntas:

1. ¿Qué entiende por efecto de reverberación?
2. ¿Qué papel desempeñan los factores *twiddle* (Figura A.30) en el algoritmo que produce el efecto de reverberación?

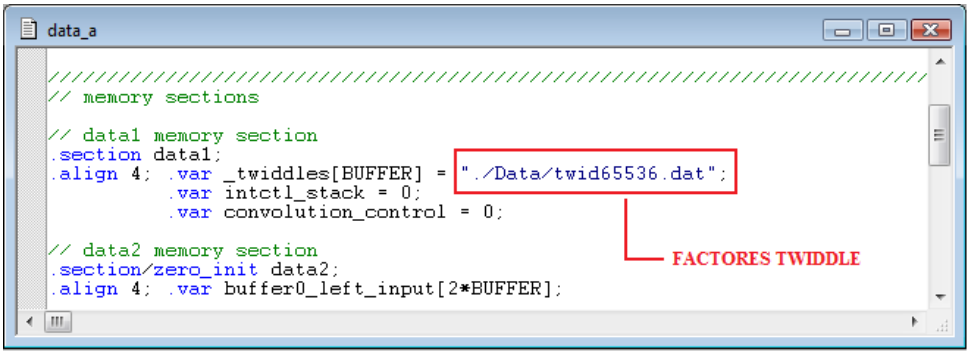

Figura A. 30. Archivo externo que incluye los factores *Twiddle*

3. ¿Por qué es necesario dentro del algoritmo de reverberación el empleo de una etapa para el cálculo de la FFT de la señal de entrada (Figura A.31)?

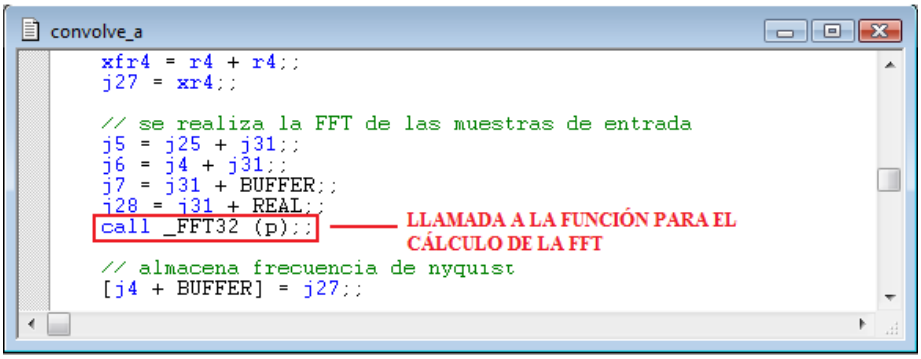

Figura A. 31. Llamada de la función para el cálculo de la FFT

4. ¿Qué características tiene la primera señal que se escucha siguiendo el procedimiento realizado en el ítem 4.2.2?
5. ¿Qué características tiene la segunda señal que se escucha siguiendo el procedimiento realizado en el ítem 4.2.2?
6. ¿Cuáles son las diferencias que encuentra entre cada una de las dos señales de salida que ha escuchado?
7. Según su criterio ¿qué produce este algoritmo discreto sobre la señal de entrada para poder generar una señal de salida que simule un efecto de reverberación de una sala de teatro?
8. En base a una breve revisión del programa, ¿qué parámetros considera usted que debería modificar para lograr un algoritmo de reverberación de otro tipo de sala?

REFERENCIAS BIBLIOGRÁFICAS

- Haykin, Simon, *Adaptive Filter Theory*, 3ra Ed, Prentice Hall, U.S.A, 1995.

- ADSP-TS201 TigerSHARC Processor Programming Reference, 1ra Ed, Analog Devices, U.S.A., Abril 2005.

- ADSP-TS201 TigerSHARC Processor Hardware Reference, 1ra Ed, Analog Devices, U.S.A., Diciembre 2004.

- ADSP-TS201S EZ-KIT Lite Evaluation System Manual, 1ra Ed, Analog Devices, U.S.A., Diciembre 2007.

- VisualDSP++ 5.0 User's Guide, 1ra Ed, Analog Devices, U.S.A., Agosto 2007.

- VisualDSP++ 5.0 Getting Started Guide, 1ra Ed, Analog Devices, U.S.A., Agosto 2007.

- VisualDSP++ 5.0 Assembler and Preprocessor Manual, 1ra Ed, Analog Devices, U.S.A., Agosto 2007.

- VisualDSP++ C/C++ Compiler and Library Manual for TigerSHARC Processors, 1ra Ed, Analog Devices, U.S.A., Agosto 2007.

- Signal Processing Applications Using the ADSP-2100 Family, Volumen 1, Analog Devices, U.S.A, 1991.

- Programs and Files DSP, www.dspguide.com, 19 de Noviembre del 2008.

Sangolquí, …. de abril del 2009

Elaborado por:

_____

Danilo Marcelo Carvajal Ramírez

_____

Paúl Fernando Páramo Vaca

Coordinador de la Carrera
que certifica

_____

Ing. Gonzalo Olmedo