UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

COMPRESIÓN DE DATOS UTILIZANDO TARJETA DSP

INFORME DE SUFICIENCIA

PARA OPTAR EL TÍTULO PROFESIONAL DE

INGENIERO ELECTRÓNICO

PRESENTADO POR:

PEDRO JAVIER RAMOS MATTA

PROMOCIÓN
1993 – II

LIMA – PERÚ
2002

A Dios por guiarme y darme fé

A mis padres Arturo y Zarela por su apoyo, cariño y comprensión

A mi hermana Roxana,Familia y Amigos por su estímulo para lograr mis objetivos

COMPRESIÓN DE DATOS UTILIZANDO TARJETA DSP

SUMARIO

En la actualidad debido al crecimiento en el volumen de la información se necesitan de técnicas que permitan mejorar la capacidad de almacenamiento de los datos, así como también mejorar la disponibilidad de acceso a dicha información a través de redes de datos, por tanto mejorar la velocidad de transmisión optimizando así el ancho de banda.

Por ésta razón el presente Informe considera la Compresión de Datos como Técnica para mejorar los servicios prestados en las redes informáticas.

Se utiliza el Algoritmo de Codificación Aritmética para Comprimir la Información implementado en hardware mediante una Tarjeta basada en el procesador digital de señales(DSP) TMS320F240 de Texas Instruments.

INDICE

PROLOGO

La compresión de datos ha adquirido gran auge en los últimos años, debido a la necesidad de mejorar las velocidades de transmisión sobretodo en Internet. El uso de procesadores digitales de señales(DSP) para el desarrollo de modernas técnicas de compresión brinda a los sistemas gran eficiencia. El sistema diseñado está basado en la técnica de Codificación Aritmética. El trabajo se realiza sobre la base de un procesador digital de señales(DSP) de la familia TMS320F2xx de Texas Instruments, específicamente el TMS320F240. Estos procesadores presentan una serie de características optimizadas para aplicaciones matemáticas lo cual permite al sistema una gran flexibilidad.

El trabajo incluye una explicación teórica de compresión, las características y funcionamiento del hardware y el software del sistema, así como las pruebas realizadas y los resultados obtenidos. Con éste fin se ha estructurado dicho trabajo en los siguientes capítulos:

El **Capítulo I** ofrece una descripción general del trabajo a realizar, la definición del problema a resolver, los objetivos propuestos con el mismo.

El **Capítulo II** brinda el marco teórico necesario para comprender el funcionamiento de un sistema de compresión, explicando los diferentes algoritmos de compresión.

El **Capítulo III** describe el hardware del sistema de compresión. Se ofrece además una breve explicación de la tarjeta evaluadora(kit) del DSP utilizada en el trabajo así como de las características, arquitectura y funcionamiento de los principales componentes del procesador TMS320F240.

El **Capítulo IV** explica el esquema general e implementación del software del sistema y la estructura del programa. Se expone las principales pruebas realizadas, tanto desde el punto de vista de simulación, como en la programación del algoritmo de compresión en el DSP, brindando los principales resultados experimentales alcanzados con las mismas.

Por último se muestran las **conclusiones** y **recomendaciones** del trabajo, la **bibliografía** utilizada y un conjunto de **anexos** que comprenden los diagramas del módulo del sistema, así como el listado completo de los programas elaborados en el trabajo.

CAPÍTULO I
ANTECEDENTES

1.1 Descripción general

En los últimos años se ha dado un aumento espectacular tanto de la capacidad de almacenamiento de las computadoras así como en la velocidad de procesamiento de datos. Sin embargo, el auge que últimamente han alcanzado las redes de computadoras y el Internet, hace que cada vez más usuarios pidan más servicios a la red sobre la que están conectados. Servicios que, como siempre, están por encima de las posibilidades reales. Cuando hablamos de posibilidades nos referimos principalmente a la velocidad de transferencia de datos y consumo de ancho de banda, siendo éste el principal **problema** al que se enfrentan todas las redes.

El cambio a mayores velocidades no es tarea fácil, básicamente por los siguientes motivos:

- Las grandes compañías de redes WAN son lentas en cuanto a cambios se refiere. Esto se debe a su vez al volumen de cambio requerido, teniendo en cuenta la cantidad de infraestructura que debe ser modificada(cableado, tecnologías, etc.).
- Otro problema es la falta de tecnología que acepte unas velocidades muy elevadas de transmisión.

En éste entorno, para conseguir mayores prestaciones de velocidad, se debe recurrir a técnicas que permitan superar de alguna manera las deficiencias físicas de

la red y de los equipos de conexión(que les permita, por ejemplo, ofrecer videoconferencia a través de *módems* de 14400 bps.).

Así la técnica más importante en éste sentido es la compresión de datos. La compresión de datos es beneficiosa en el sentido de que el proceso de compresión-transmisión-descompresión es más rápido que el proceso de transmisión sin compresión.

Pero no sólo para la transmisión se utiliza la compresión. También para el almacenamiento masivo de datos, sobretodo cuando la necesidad de almacenamiento crece por encima de las posibilidades del crecimiento de los discos duros o memoria.

La compresión de datos ha sido una herramienta matemática que ha permitido el desarrollo de muchas aplicaciones en Internet, sin ella el Web simplemente no podría existir.

1.2 Formulación del problema

Desarrollar un sistema de compresión de datos implementado con el algoritmo de codificación aritmética utilizando para ello una Tarjeta de desarrollo TMS320F24x DSP Starter Kit.

1.3 Objetivos

- Implementar un formato de compresión de datos universal que aplique el Algoritmo de Codificación Aritmética.
- Desarrollar un sistema de compresión de datos basado en una tarjeta de evaluación del DSP TMS320F240.

Con el cumplimiento de éstos objetivos se logra agilizar la transferencia de archivos a través de redes de datos, utilizando los mecanismos de compresión desarrollados. Así también se logra mejorar la capacidad de almacenamiento de datos.

CAPÍTULO II
COMPRESIÓN DE DATOS

2.1 Introducción

Es muy difícil clasificar los distintos tipos de compresión de datos que existen, debido a que, por un lado tenemos muchos algoritmos: LZ77, LZ78, LZW, Huffman, aritméticos, fractales, MPEG, JPEG, etc.

Además, hay muchas aplicaciones que utilizan la compresión con distintas expectativas: compresión de datos, vídeo y voz, compresión en tiempo real, etc. A su vez, cada uno requiere ciertas características como velocidad, reversibilidad(que el algoritmo pueda ser aplicado de forma reversible para obtener los datos originales), pérdida mínima de información(en el caso de los algoritmos con pérdida de información, etc.).

La compresión que se logra normalmente con los algoritmos más utilizados se describe en la Tabla 2.1.

Tabla 2.1 Los algoritmos de compresión más utilizados

Tipo de datos	Algoritmo o Formato	Tipo de compresión	Tasa de compresión
Archivos	Lempel-Ziv	Sin pérdida	2:1 – 5:1
Voz	ADPCM	Con pérdida	4:1 – 8:1
Audio	MPEG niveles I, II y III	Con pérdida	4:1 – 12:1
Imágenes continuas	JPEG	Con pérdida	10:1 – 100:1
Imágenes discretas	GIF	Sin pérdida	2:1 – 5:1
Video	MPEG – 1 y 2	Con pérdida	25:1 – 66:1
Videoconferencia	H.261	Con pérdida	24:1 – 95:1

Se ha elegido una clasificación muy general tomando como característica de división la reversibilidad del algoritmo. Así, los algoritmos de compresión se pueden dividir en dos tipos:

Compresores **lossless** o sin pérdidas, en el sentido que guarda absolutamente toda la información original(es reversible). Se utilizan para la compresión de datos, en los que no se puede dar pérdida de información, entre ellos se encuentran Huffman y Shannon-Fano, Aritméticos, Predictivos, Run Length(RLE) y de Lempel Ziv Welch(LZW).

Compresores **lossy** o con pérdidas, la compresión hace que se pierda información de la fuente original. Sin embargo, ésta pérdida es insignificante en comparación con la ganancia en compresión. Se utiliza sobre todo en imágenes y sonido, donde se puede "engañar" a los sentidos con una pérdida de calidad apenas percibida(pero ocasiona un **ratio** de compresión mucho mayor), entre ellos se encuentran GIF, JPEG, MPEG, MPEG 3(MP3).

En la Tabla 2.2 se clasifican los algoritmos existentes según los criterios mencionados.

Tabla 2.2 Clasificación de los algoritmos de compresion

<table>
<tr><td rowspan="5">Compresión de la entropía</td><td colspan="2">Run Length Coding</td></tr>
<tr><td rowspan="2">Codificación Estadística</td><td>Codificación Huffman</td></tr>
<tr><td>Codificación Aritmética</td></tr>
<tr><td>Compresión por diccionario</td><td>Lempel-Ziv</td></tr>
<tr><td rowspan="3"></td><td>Predicción</td><td>ADPCM</td></tr>
<tr><td rowspan="2">Transformación</td><td>FFT</td></tr>
<tr><td>DCT</td></tr>
</table>

Compresión de la fuente	Codificación por nivel	Codificación por sub-bandas
		Sub-muestreo
		Posición de los bits
		Cuantización vectorial
Compresión híbrida	JPEG	
	MPEG	
	H.261	

El nivel de compresión mejora mucho con la ayuda de chips o tarjetas dedicadas, pues la compresión por software es muy lenta(por ejemplo en MPEG-2 es imprescindible el uso de hardware especial).

En el presente trabajo sólo describiremos los algoritmos sin perdida.

2.2 Codificación Run Length(RLE)

En varios tipos de datos encontramos largas repeticiones consecutivas de símbolos, que pueden ser reemplazadas por tres símbolos: un marcador especial, un contador del número de símbolos repetidos y uno de éstos símbolos. Por ejemplo si consideramos el siguiente mensaje: H00000000000LLAAAAAAAAAAAAAAA

El mensaje puede ser considerablemente comprimido a: H*11OLL*15A, siendo * el caracter de control. Si éste caracter se encontrase en el mensaje el compresor lo colocará dos veces.

Las repeticiones de símbolos son comunes en el audio(por ejemplo, representando el silencio como grupos de ceros), en imágenes o video(con colores que varían poco dentro de una misma zona de la imagen).

2.3 Codificación Huffman

Es la más conocida compresión estadística sin pérdida de un mensaje, resultado del estudio de la teoría de la información y de las probabilidades.

La idea es observar que tan seguido se repiten secuencias de símbolos y asignarle códigos más cortos a las frecuentes y más largos a los que ocurren con menor probabilidad.

Se asignan códigos de largo variable a cada símbolo según su probabilidad de ocurrencia. Para determinar el código Huffman es útil construir un árbol binario. Las hojas representarían a los caracteres a comprimir. Cada nodo tendría un valor que corresponde a la probabilidad de la ocurrencia de los caracteres bajo su subárbol. El árbol se construye partiendo de las hojas hacia la raíz:

1. Los símbolos se ordenan según su frecuencia o probabilidad de ocurrencia.
2. Sucesivamente los dos símbolos o nodos con las menores probabilidades son reemplazados por un nodo compuesto padre que tendrá como probabilidad la suma de la de los nodos hijos. Cuando todos los subárboles han sido combinados tenemos un árbol de código Huffman.
3. El último paso es asignar códigos a los símbolos. Se asignan códigos partiendo de la raíz hacia abajo.

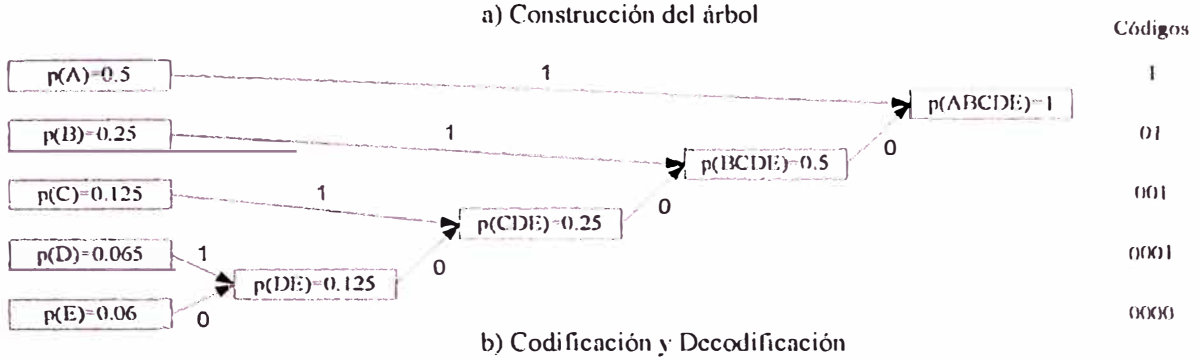
En la figura 2.1a encontramos un ejemplo de codificación Huffman para mensajes que sólo contienen lo símbolos A, B, C, D y E, y que tienen la probabilidad indicada.

Los códigos Huffman tienen la propiedad de que ninguno es prefijo de otro. De éste modo, existe un modo único de decodificar el mensaje.

La secuencia de caracteres es codificada símbolo por símbolo, mientras que la decodificación es realizada bit por bit. En la figura 2.1 observamos como se identifica únicamente cada caracter a partir del mensaje codificado.

Fig. 2.1 Ejemplo de Codificación Huffman

a) Construcción del árbol



b) Codificación y Decodificación

Mensaje Original: ABECEDE
Código Huffman: 10100000010000000010000
Decodificado: 1-01-0000-001-0000-0001-0000

La compresión Huffman es óptima, en el sentido de que utiliza la menor cantidad de bits para representar la información, sólo si las probabilidades son potencias negativas de 2(½, ¼, etc.). Esto no sucederá en el caso normal. El peor caso es cuando un símbolo es tan frecuente que su probabilidad se acerca a 1. Según la Teoría de la Información sólo sería necesaria una fracción de un bit para representar éste símbolo, sin embargo, los códigos Huffman no pueden ser menores a 1 bit. En éstas situaciones la compresión aritmética se comporta mejor.

El modelo descrito es estático. Se necesita una pasada para recolectar información acerca de las probabilidades de cada símbolo y de otra pasada para comprimir los datos. Una de las extensiones al código Huffman es hacerlo adaptivo. De éste modo, el árbol de códigos Huffman es actualizado cuando las probabilidades o frecuencia de los símbolos han cambiado. Sin embargo, el código adaptivo Huffman es mucho más complejo que las versiones adaptivas de otros algoritmos, como la compresión aritmética o Lempel-Ziv, por lo que es menos utilizado que éstas.

Otra limitación es el espacio utilizado en el árbol de códigos, el cual puede crecer mucho si el número de símbolos es elevado.

Una aplicación muy conocida de la compresión Huffman es el fax. Los cuales contienen algoritmos que comprimen sin perdida, imágenes binarias de fax. Estos algoritmos aplican compresión en una corrida(Run Length) y luego Huffman.

Un algoritmo más simple, similar a Huffman, es la codificación Shanon-Fano. Funciona así:

1. Se divide cada conjunto de símbolos(al principio hay sólo uno) en dos subconjuntos aproximadamente iguales, basado en la probabilidad de los caracteres de cada subconjunto. Al primer conjunto se le asigna un 0, al segundo un 1.
2. Se repite el paso anterior hasta que todos los subconjuntos tengan un sólo elemento(siempre agregando al código de cada subconjunto los ceros y unos que se le asignaron en el paso anterior).

La codificación de los símbolos se hará de acuerdo a los códigos obtenidos mediante los pasos anteriores. Este algoritmo opera de modo arriba-abajo(top-down), mientras que Huffman es abajo-arriba(botom-up). Este método algunas veces usa unos pocos bits más y fue usado en la primera versión del popular ZIP.

2.4 Codificación Aritmética

Esta técnica logra tasas de compresión muy cercanas al óptimo teórico. La idea es combinar todos los símbolos del mensaje, de modo que sean representados por sólo un número(de gran precisión).

Un ejemplo se muestra en la tabla 2.3, donde se comprimirá el mensaje "COMPRESORA". El proceso se inicia asignando probabilidades a cada símbolo del

mensaje, de modo que las suma de las probabilidades de todos los símbolos sea 1. Luego, se asigna un rango diferente a cada símbolo en el intervalo de 0 a 1. El ancho del rango corresponderá a la probabilidad del símbolo.

Luego, la codificación se realiza símbolo a símbolo, agregando precisión a un rango de números que llamaremos el intervalo del mensaje. Una vez finalizada la codificación, cualquier número dentro de éste intervalo podrá ser codificado únicamente como el mensaje original.

En el primer paso, Intervalo = [0.0 – 1.0]

Para codificar un nuevo símbolo en el paso i+1:

- Ancho(i+1) = [Intervalo(i+1)] – [Intervalo(i)]
- Intervalo(i+1) = Rango(i) x Ancho(i) + [Intervalo(i)]

Así el intervalo del mensaje irá reduciéndose con cada nuevo símbolo. Una vez terminada la codificación, el codificador podrá tomar cualquier número dentro del intervalo del mensaje, para enviarlo o almacenarlo. Este número representará únicamente al mensaje.

Tabla 2.3 Ejemplo de codificación aritmética estática
Mensaje = COMPRESORA
(a) Asignación de rangos a cada símbolo

Símbolo	Probabilidad observada	Rango del símbolo
C	0.1	0.0 – 0.1
O	0.2	0.1 – 0.3
M	0.1	0.3 – 0.4
P	0.1	0.4 – 0.5
R	0.2	0.5 – 0.7
E	0.1	0.7 – 0.8
S	0.1	0.8 – 0.9
A	0.1	0.9 – 1.0

(b) Codificación del intervalo

Símbolo	Rango (del símbolo)	Ancho (del intervalo anterior)	Intervalo (del mensaje)
			0.0 – 1.0
C	0.0 – 0.1	1.0	0.0 – 0.1
O	0.1 – 0.3	0.1	0.01 – 0.03
M	0.3 – 0.4	0.02	0.016 – 0.018
P	0.4 – 0.5	0.002	0.0168 – 0.017
R	0.5 – 0.7	0.0002	0.0169 – 0.01694
E	0.7 – 0.8	0.00004	0.016928 – 0.016932
S	0.8 – 0.9	0.000004	0.0169312 – 0.0169316
O	0.1 – 0.3	0.0000004	0.01693124 – 0.01693132
R	0.5 – 0.7	0.00000008	0.01693128 – 0.016931296
A	0.7 – 1.0	0.000000016	0.0169312944 – 0.016931296

Cuando el decodificador tome el número verá que está entre 0 y 0.1, que es justamente el rango del símbolo C y, por lo tanto, determinará que el primer símbolo del mensaje es C. Luego, resta el límite inferior del rango del símbolo al número y lo divide por el rango del símbolo para re-escalarlo. Se continua extrayendo símbolos hacia la derecha hasta un indicador especial de término, necesario pues puede que el proceso tenga un resto.

Un problema es cómo codificar mensajes largos, contando sólo con una cierta cantidad de bits para la precisión del número. Esto se soluciona codificando incrementalmente el mensaje. Una vez que la precisión no permite diferenciar entre los límites inferior y superior del intervalo del mensaje, éste número es enviado y se comienza de nuevo.

Para determinar las probabilidades iniciales de los símbolos es posible analizar sólo una fracción del mensaje. Mejor aún, una versión adaptiva del algoritmo iría ajustando las probabilidades con cada nuevo símbolo codificado. La versión adaptiva

tiene las ventajas(sobre Huffman) de no tener que analizar de antemano el mensaje, mantenerse óptimo frente a cambios en las probabilidades y no tener que enviar la tabla de probabilidades, puesto que el decodificador iría actualizando su tabla con cada nuevo símbolo tal como lo hizo el codificador al codificarlo.

Existe una variante binaria del algoritmo, en que se considera cada mensaje como binario(0 y 1) y se codifican bit por bit. Esto permite simplificar las estadísticas y la aproximación de las probabilidades.

La gran desventaja de éste algoritmo es su velocidad. La cantidad de operaciones matemáticas necesarias para codificar cada símbolo demoran el proceso y exigen poder computacional.

2.5 Codificación Lempel Ziv Welch(LZW)

La idea es cambiar secuencias de símbolos frecuentes por índices de un diccionario de tales secuencias. Obviamente, el índice deberá ocupar menos bits que la secuencia de símbolos.

Este tipo de algoritmos funciona muy bien con texto, donde las mismas palabras se repiten con frecuencia. Esta idea, por ejemplo, es la que se utiliza al emplear los números 1 al 12 en vez de nombrar los meses, ahorrando espacio.

A fines de los Años 70 Abraham Lempel y Jacobo Ziv publicaron dos algoritmos adaptivos LZ77(Ziv77) y LZ78(Ziv78), que luego serían muy utilizados.

Mientras que los Códigos de Huffman proporcionaron buenos resultados, éstos se limitaron a codificar un carácter a la vez. Lempel y Ziv propusieron un esquema de diccionario(adaptativo) para codificar cadenas de datos. Este algoritmo tomó ventaja de la secuencia de caracteres que ocurren frecuentemente como la palabra "the" del idioma inglés.

En 1984 Terry Welch publicó para la *IEEE* el algoritmo LZW(Lempel Ziv Welch), una variante del algoritmo LZ77, que llegó a ser el algoritmo de compresión de datos para las computadoras personales.

En la codificación LZW se busca reemplazar cadenas de caracteres con códigos simples que son almacenados en un diccionario o tabla de cadenas. El algoritmo no analiza para nada el texto de entrada, únicamente añade cada nueva cadena a una tabla de cadenas de caracteres.

El código que genera el algoritmo puede ser de cualquier longitud arbitraria, pero debe tener más bits que un carácter simple. Los primeros 256 códigos(cuando se usan caracteres de 8 bits) se asignan por omisión al conjunto estándar de caracteres. Los restantes son asignados a cadenas a medida que el algoritmo realiza su trabajo.

La porción de pseudo código siguiente ilustra el algoritmo en su forma más simple.

```
STRING = get(caracter de entrada)
WHILE "existan caracteres de entrada" DO
        CHARACTER = get(caracter de entrada)
        IF STRING+CHARACTER "está en la tabla" then
                STRING = STRING+ CHARACTER
        ELSE
                "Enviar como salida el código de STRING"
                add STRING+CHARACTER " a la tabla
                STRING = CHARACTER
        END of IF
```

END of WHILE

"Enviar como salida el código de STRING"

En la descompresión LZW se crea la misma tabla de cadenas y se actualiza por cada caracter en el flujo de entrada excepto para el primero. Después de que el caracter ha sido expandido a su cadena correspondiente vía la tabla de cadenas, el caracter final de la cadena es agregado a la cadena previa. Esta nueva cadena se agrega a la tabla en la misma localidad que en la tabla de cadenas del compresor.

2.6 Consideraciones

Los algoritmos que se han presentado tienen virtudes y desventajas frente a los otros. Dado que, ninguno es mejor en todos los aspectos a los otros, entonces ¿qué algoritmo de compresión debe uno de utilizar?

Como siempre, todo depende del ámbito de aplicación. La compresión Run-Length es la menos poderosa, pero nadie lo hará mejor si los datos a comprimir tienen grandes cantidades de caracteres repetidos, como podría ser cierto tipo de imágenes binarias como el fax.

La literatura menciona evaluaciones de rendimiento sobre el rango de compresión y velocidades de compresión y descompresión. La tabla 2.4 otorga puntajes en éstas áreas a los algoritmos mencionados en las secciones anteriores. Otros aspectos como utilización de memoria han sido poco estudiados.

Tabla 2.4 Rendimiento de los algoritmos para datos simbólicos

(Bajo=1,Alto=5)	Run Length Coding	Huffman	Aritmética	LZ77/LZ78
Tasa de compresión	1	3	5	4
Velocidad de compresión	5	2	1	4
Velocidad de descompresión	5	3	1	4

El algoritmo Run-Length sólo remueve un tipo particular de redundancia, pero es muy rápido. Huffman es mucho más poderoso, pero sólo es óptimo en condiciones muy especiales. Para la compresión necesita dos pasadas por el mensaje y la descompresión es lenta pues se realiza bit a bit. La codificación Aritmética se acerca al óptimo teórico, pero requiere de gran número de operaciones matemáticas, lo que hace el algoritmo más lento. Los algoritmos Lempel-Ziv-Welch son casi tan poderosos como la codificación aritmética y bien implementados procesan muy rápido.

Es probable que resulten avances en ésta área en el futuro. Puede esperarse una mayor velocidad de procesamiento(uno a dos ordenes de magnitud), no tanto por mejoras a los algoritmos mismos, sino por el uso de hardware dedicado. Hoy en día, el más rápido circuito comprime datos simbólicos usando una versión rápida de LZ77 a una tasa de 40 Mbytes por segundo. Con las continuas mejoras a los circuitos VLSI se espera se desarrollen codificadores más rápidos y complejos.

CAPÍTULO III
HARDWARE DEL SISTEMA DE COMPRESIÓN

3.1 Introducción

La forma de resolver un proyecto propuesto se centraliza en mayor porcentaje de peso en el diseño, fabricación y pruebas de la tarjeta de Procesamiento Digital de Señales, para la adquisición de señales y compresión de datos, por medio de un DSP con respuesta en Tiempo Real.

En la actualidad las empresas líderes en adquisición, y procesamiento de señales han desarrollado los Procesadores de Señales Digitales(DSP), cuyas características son ventajosas con relación a procesadores convencionales en éste tipo de aplicaciones. Apoyándose en un DSP se han construido Kits de desarrollo y módulos de evaluación de alto rendimiento en diferentes versiones y diferentes configuraciones. La forma de emplearse y los programas de aplicaciones también tienen bastante variedad y han permitido el auge de un creciente campo de aplicaciones basadas en DSP.

Texas Instruments es pionero y líder en el desarrollo y fabricación de Kits y Módulos de Evaluación para aplicaciones en Tiempo Real y tiene disponible actualmente varios modelos para aplicaciones específicas, de los cuales sólo se hará referencia al utilizado en el Proyecto.

3.2 Hardware del Sistema

Comprende las siguientes componentes:

- Computadora Personal, la computadora constituye el elemento de interfaz entre el usuario y el sistema de compresión. La tarjeta de evaluación(kit) de DSP se conecta a la PC a través del puerto serial. En ella se corren los programas de ensamble, enlace y depuración("debugger"), del procesador TMS320F240.
- Tarjeta de evaluación(kit) del DSP TMS320F240 y componentes accesorios.

3.3 Tarjeta de Evaluación del DSP TMS320F240 Starter Kit

La familia de procesadores de Texas Instrumentas TMS320F2xx, proporciona un conjunto de dispositivos y herramientas de software para la confección y puesta en marcha de aplicaciones relacionadas con éstos procesadores. Este conjunto de herramientas está compuesto por:

1. Tarjeta de DSP("F24x Evaluation board").
2. Adaptador de puerto serial, éste dispositivo sirve de interfaz entre la PC y la tarjeta de evaluación.
3. Fuente de alimentación: Entrada: 100-250v, Salida: 5v, 3.3A.
4. Software: Incluye ensamblador, enlazador("linker"), y el emulador("F24x EVM C Source Debugger"). Además se brindan algunas utilerías básicas y bibliotecas de programas en ensamblador del TMS320F240.

El Anexo A muestra información acerca de la instalación y descripción de la tarjeta de evaluación. A continuación se explican algunos de los componentes fundamentales de ésta tarjeta:

1. Procesador digital de señales(DSP), TMS320F240, de punto fijo(U6).

2. 128 Kwords de memoria RAM en la tarjeta(U3 y U4).
3. Conversor digital/analógico(D/A) de 12 bits, con cuatro canales de conversión(U9).
4. Dos arreglos lógicos de compuertas(GAL) de tipo 16V8 para la lógica de control y decodificación de la tarjeta(U7 y U14).
5. Oscilador de reloj de 10Mhz(U16).
6. Puerto serial compatible con RS232(P6).
7. Puerto para el adaptador XDS51010PP(P5).
8. Banco de 8 conmutadores("DIP switches" – SW2).
9. Banco de 8 LEDS(SW1).
10. Cuatro conectores de expansión de 34 pines, agrupados funcionalmente, los cuales brindan las señales más importantes del DSP y la tarjeta de evaluación. Estos conectores son:
    - Conector de Entrada/Salida: permite el acceso a todas las señales del módulo de manejo de eventos("event manager") del DSP, así como a las señales de los puertos de comunicación serial(SCI y SPI).
    - Conector analógico: brinda las señales de los módulos de conversión A/D y D/A.
    - Conector de direcciones/datos: a través del conector se accede a las señales de los buses de direcciones y datos del procesador con el objetivo de conectar periféricos externos o expandir memoria.
    - Conector de control: brinda las señales de control más importantes del procesador.

La distribución de pines de cada conector puede ser consultada en el manual TMS320F24x DSP Controllers. Evaluation Module.

3.4 El Procesador Digital de Señales(DSP) TMS320F240

A continuación se exponen algunas de las características más importantes del procesador TMS320F240.

3.4.1 Características generales

a.) Procesadores de 16 bits de punto fijo.

b.) Ciclo de instrucción de 50ηs(20 MIPS).

c.) Memoria:

- 544 words de RAM interna de acceso dual(DRAM).
- 16 kwords de ROM tipo flash(EEPROM).
- 224 kwords de memoria total(64kwords de memoria de programa, 64kwords de memoria de datos, 64kwords de espacio de entrada/salida y 32kwords de memoria global).

d.) Código compatible con los procesadores de la familia TMS320C5x.

e.) Seis fuentes de interrupción, cada una con múltiples subniveles.

f.) 12 canales para modulación de ancho de pulso(PWM).

g.) Tres temporizadores("timers") de 16 bits, con seis modos de trabajo cada uno.

h.) Módulo de conversión analógico/digital de 10 bits con 6 μs de tiempo de conversión y 16 entradas analógicas, divididas en dos bloques de ocho entradas cada uno, lo que permite la lectura de dos señales simultáneamente.

i.) 28 líneas de entrada/salida multiplexadas.

j.) Dos puertos para comunicación serial(SCI y SPI).

3.4.2 Módulo Manejador de Eventos ("Event Manager")

Este módulo proporciona una amplia gama de funciones que son de gran utilidad para aplicaciones de control de motores.

1. Tres temporizadores de 16 bits de propósito general("GP timers 1, 2 y 3").
2. Tres unidades de comparación completa("Full compare 1, 2 y 3").
3. Tres unidades de comparación simple("Simple compare 1, 2 y 3").
4. Circuitos para generación de señales PWM que incluyen:
   - Circuito de generación de PWM usando el método de vectores espaciales.
   - Unidad de generación de banda muerta("dead band").
   - Lógica de salida que permite controlar el nivel de activación de las señales PWM que se generan.
5. Cuatro unidades de captura.
6. Circuito para detección de pulsos en cuadratura(QEP).
7. Interrupciones asociadas a la generación de cada evento del módulo.

3.4.3 Temporizadores("Timers")

Cada temporizador puede ser programado en seis modos de trabajo diferentes(modo 0 al modo 5), dependiendo de las características de la señal que se necesite usar como base de tiempo. La programación de los temporizadores requiere de la iniciación de los siguientes registros:

1. GPTCON: se programan los niveles de activación de las salidas de cada temporizador, el inicio de conversión del A/D por un evento del temporizador, etc.
2. TxCON(x=1,2,3): registro de control de cada temporizador, se programa el modo de trabajo, la fuente de reloj del temporizador, el valor del pre-escalador,

entre otras funciones.

3. TxPER(x=1,2,3): registro de período de la señal, se programa la duración del período(cantidad de pulsos de reloj) de la señal que se genera.
4. TxCNT(x=1,2,3): registro contador de pulsos de reloj de cada temporizador. La estructura completa de cada registro se muestra en el manual TMS320F24x DSP Controllers. Vol 2.

3.4.4 Unidades de Comparación Completa("Full Compare")

Este procesador incluye tres unidades de comparación completa(1,2,3), que permiten **comparar** el valor del contador de un temporizador, con otro valor previamente cargado en un registro de comparación. Cuando ambos valores coinciden se modifica el valor de las salidas de la unidad. Cada unidad tiene dos salidas complementadas, que pueden ser programadas para generar PWM.

La programación de las unidades de comparación completa incluye la iniciación de los siguientes registros:

1. COMCON: permite programar la condición de recarga de los comparadores, el modo de trabajo de cada salida (comparación/PWM), entre otras funciones.
2. ACTR: se programa el nivel de activación de cada una de las salidas de cada unidad.
3. CMPRx: registro donde se carga el valor a comparar para cada una de las unidades.

La estructura completa de cada registro se muestra en el manual TMS320F24x DSP Controllers. Vol 2.

La familia de procesadores TMS320F2xx, proporciona hasta doce salidas PWM, uniendo las seis salidas de las unidades de comparación completa, las tres de las

unidades de comparación simple y las salidas individuales de cada temporizador. Entre las posibilidades que brinda la generación de PWM se encuentran:

- Resolución máxima: 16 bits.
- Mínima amplitud del pulso de PWM: Un ciclo de reloj(50ηs).
- Posibilidad de cambiar la frecuencia de portadora durante el conteo. Esto es posible por la existencia de registros de períodos(TxPER) dobles, que permiten escribir un nuevo valor sin afectar el conteo actual.
- Posibilidad de cambiar la amplitud de pulso durante el conteo. Esto es posible por la existencia de registros de comparación(CMPRx) también dobles.
- Posibilidad de programar banda muerta("dead band"), usando el registro DBTCON en un rango de 0 a 102μs para un reloj de 50ηs. La banda muerta es un intervalo de tiempo necesario para la conmutación de los transistores en un inversor.
- Posibilidad de programación de PWM en tres modos de trabajo: asimétrico, simétrico y por generación de vectores espaciales(SV PWM).

3.4.5 Unidades de Captura

Las unidades de captura permiten leer el valor del contador de un temporizador cuando una determinada señal llega a la entrada de la unidad. Este procesador contiene cuatro unidades de captura con las siguientes características:

1. Cada unidad puede elegir el temporizador 2 ó 3 como base de conteo.
2. Cada unidad presenta una estructura FIFO("First In First Out") de dos niveles de 16 bits que permite almacenar hasta dos conteos.
3. Cada unidad presenta un registro de control(CAPCONx), donde se programa,

entre otras, la habilitación de la unidad, la selección del temporizador que utilizará y el frente de activación de la señal de captura(subida, bajada o ámbos frentes).

4. Las unidades de captura 1 y 2 pueden ser programadas en un modo llamado **cuadratura de pulsos**(QEP) en la cual se detecta el tiempo transcurrido entre dos pulsos desfasados 90°.

3.4.6 Interrupciones del Módulo de Manejo de Eventos

Este módulo genera una gran cantidad de eventos que pueden ser tratados por interrupción y que se encuentran agrupados en tres bloques llamados A, B y C. Entre las fuentes de interrupción más importantes se encuentran:

1. **Interrupción por comparación** en las unidades de comparación completa: se produce cuando el valor del registro de comparación es igual al valor del contador del temporizador.
2. **Interrupciones de los temporizadores,** se producen por:
   - Período: cuando el valor del contador es igual al del registro de período(TxPER).
   - "Overflow": el contador alcanza su máximo valor.
   - "Underflow": el contador alcanza el valor 0 contando de forma descendente.
3. **Interrupciones de la unidad de captura,** el trabajo con interrupciones involucra la programación de los siguientes registros:

   a.) IMR: registro de máscaras de interrupción, el cual habilita o no las seis fuentes de interrupciones principales del procesador.

b.) IFR: registro de banderas de interrupción, el cual permite conocer el estado de las fuentes de interrupción. Escribiendo un "1" en el bit correspondiente se limpia esa solicitud de interrupción.

c.) EVIMRA, EVIMRB, EVIMRC: registros de máscaras de interrupción de cada una de las fuentes de interrupción del módulo de manejo de eventos.

d.) EVIFRA, EVIFRB, EVIFRC: registros de banderas de interrupción de cada una de las fuentes de interrupción del módulo de manejo de eventos.

3.5 Módulo de Conversión Analógico Digital(A/D)

El procesador TMS320F240 incluye un módulo de conversión analógico/digital interno. Un esquema general de este módulo se muestra en el anexo A, el cual presenta las siguientes especificaciones:

1. 16 entradas analógicas agrupadas en dos módulos multiplexores de 8 bits cada uno, lo cual permite la lectura de dos canales simultáneamente.
2. Resolución de los conversores: 10 bits.
3. Tiempo de conversión: 6.6μs.
4. Estructura FIFO de dos niveles para cada conversor, lo cual permite el almacenamiento de dos lecturas sin pérdida de información.
5. Los voltajes de referencia máximo($Vref_{hi}$) y mínimo ($Vref_{lo}$), pueden ser establecidos en cualquier valor entre 0 y +5v.
6. Posibilidad de realizar conversión simple o continua.
7. El inicio de conversión puede ser suministrado por software, eventos internos del módulo A/D o eventos externos producidos por el módulo manejador de eventos("event manager").

8. Circuito pre-escalador programable. La elección de un valor en el pre-escalador debe satisfacer la siguiente fórmula:

SYSCLK * prescaler*6 > 6μs, donde SYSCLK es 2* CLK = 100ηs.

La programación del módulo de conversión A/D se realiza a través de los siguientes registros:

1. ADCTRL1: se habilita el conversor, el inicio de conversión, la interrupción del A/D y el canal a leer.
2. ADCTRL2: se programa el valor del pre-escalador, el inicio de conversión por un evento externo, entre otras funciones.

3.6 Módulo de Conversión Digital Analógico(D/A)

La tarjeta de evaluación del DSP TMS320F240 incorpora un conversor digital analógico de cuatro canales con una resolución de 12 bits cada uno. Cada canal tiene un registro asociado en el cual se carga el valor digital a convertir. Estos registros están mapeados en la página 0 de memoria y ocupan las siguientes direcciones:

Registro	Dirección	Descripción
DAC0	0000h	Registro de entrada Canal 0
DAC1	0001h	Registro de entrada Canal 1
DAC2	0002h	Registro de entrada Canal 2
DAC3	0003h	Registro de entrada Canal 3
DAC UPDATE	0004h	Registro de actualización

El registro DAC UPDATE permite iniciar un ciclo de conversión escribiendo cualquier valor en el mismo.

El módulo de conversión D/A requiere la generación de estados de espera para una operación correcta. El procesador debe ser programado para generar estados de espera por software, para lo cual son necesarios los siguientes requerimientos:

1. El registro WSGR debe ser programado para habilitar un estado de espera en el espacio de entrada/salida.
2. La señal CPUCLK de 20 Mhz debe ser enviada por el pin CLKOUT del DSP. Esta señal es usada por la GAL(U14) de la tarjeta de evaluación para generar los estados de espera requeridos por el módulo de conversión D/A.

La resolución de los cuatro canales del conversor D/A es de 12 bits por lo que se pueden enviar valores entre 0 y 0fffH equivalentes a niveles de voltajes analógicos a la salida entre 0 y 5v.

Este capítulo abordó los aspectos relacionados con las especificaciones y el funcionamiento de los módulos que intervienen en la Tarjeta DSP.

CAPÍTULO IV
PROGRAMACIÓN DE LA APLICACIÓN

4.1 Elección del Algoritmo

En el Capítulo II se describió la Compresión de Datos y se consideró a la Codificación Aritmética como uno de los algoritmos más óptimos(Tabla 2.4), pero su principal desventaja era que se requería de un gran procesamiento matemático, es precisamente por dicha razón por la cual se seleccionó el Algoritmo debido a que su implementación se realizará con una tarjeta digital de señales con un procesador basado en DSP, el cual ha sido diseñado para éste tipo de trabajos.

También se consideró en la evaluación el desempeño de los algoritmos Huffman, Run Length, y Lempel Ziv Welch, cuyos resultados se muestran en la Tabla 4.1a, para un conjunto de archivos con diferentes formatos.

Se observa, que los algoritmos con mejor desempeño son Huffman y Aritmético, pero se descartó el Huffman(almacena en memoria árboles binarios para codificar) considerando las limitaciones de memoria de la Tarjeta DSP.

Tabla 4.1a Desempeño de los Algoritmos de Compresión

Algoritmo	**Tipo de Archivo**				
	Texto (TXT)	**Documento (DOC)**	**Imagen (JPG)**	**Aplicación (EXE)**	**Tablas (XLS)**
Huffman	–	√	√	√	–
Aritmético	√	√	√	√	√
Run Lenght	√	√	**X**	**X**	√
LZW	–	√	√	**X**	**X**

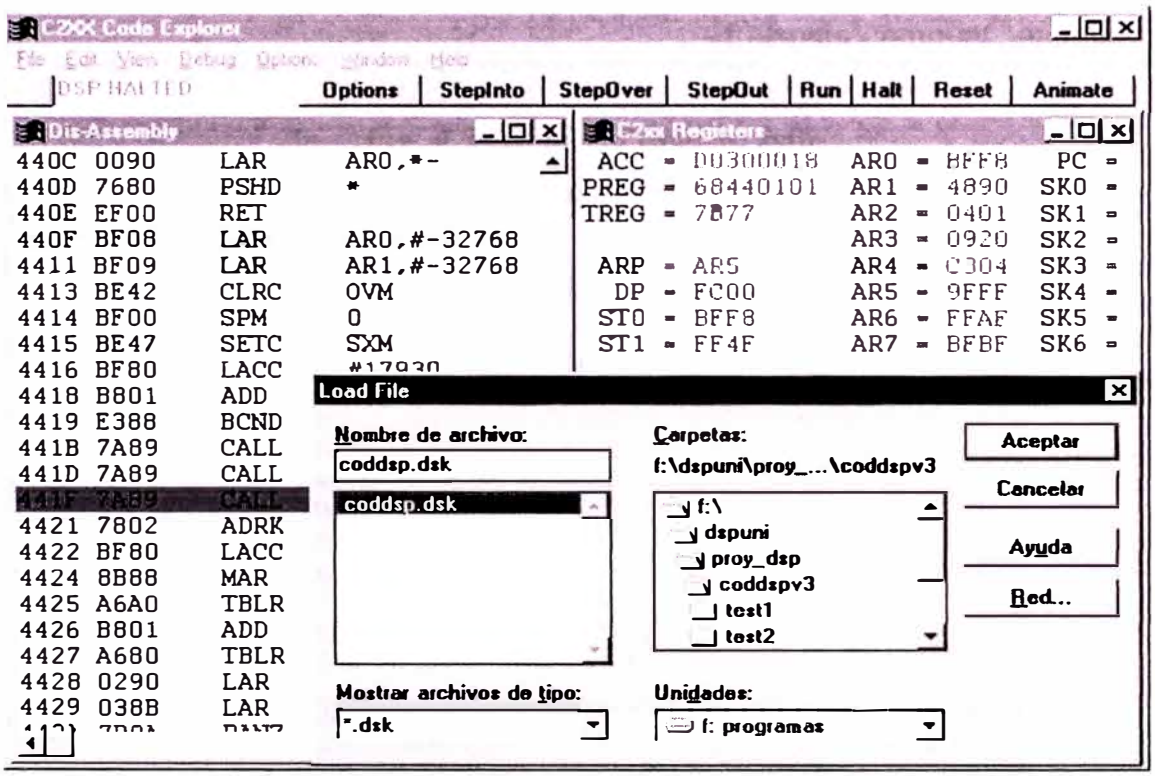
Tabla 4.1b Desempeño del Algoritmos de Compresión

Símbolo	Descripción Compresión / Descompresión
√	valores correctos para todas las pruebas
–	valores de respuesta para algunos archivos
X	valores incorrectos

4.2 Tarjeta de Evaluación del DSP TMS320F240

La Tarjeta de desarrollo TMS320F240 de National Instruments trabaja con el Code Explorer, el cual es un Compilador en Lenguaje Ensamblador que permite realizar tareas de carga, ejecución y depuración de programas en formato DSK.

Fig. 4.1 Compilador Code Explorer



4.2.1 Secuencia de desarrollo de un Proyecto

Cada procesador tiene una configuración de memoria, de lenguaje y de periféricos. La mayoría de compiladores trabajan incluso con varias familias de procesadores, por dicha razón hay que tener en cuenta los drivers y las opciones que permiten la configuración del DSP requerido. Para el caso del TMS320F240, éste no

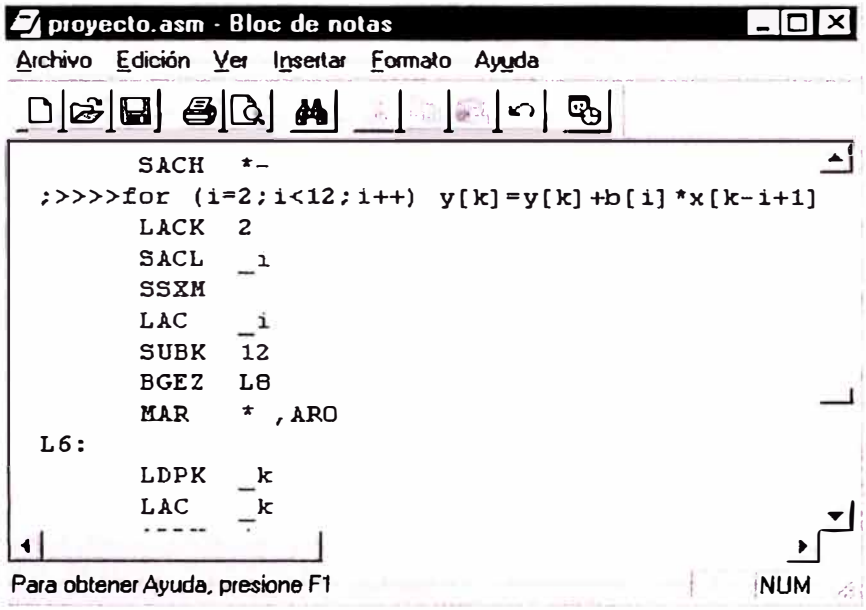
tiene drivers para el Code Explorer en Lenguaje C, lo que se utiliza es un compilador que trabaja en ambiente DOS, sin embargo todas la simulaciones y los programas trabajan de manera adecuada. Y a continuación correr una serie de programas que compilan y construyen el lenguaje ensamblador para el TMS320F240.

Compilación

Mediante un comando de DOS se ejecuta la compilación, donde se crea un archivo ensamblador **Proyecto.asm**, el que está preparado para el F240 pero no está construído en cuanto a las posiciones de memoria donde se va a ubicar el programa.

```
C:\>F:\dspuni\compilaC\bin\dspcl -al -s -k -v2xx Proyecto.c
                                          -if:\dspuni\compilaC\include
```

Fig. 4.2 Archivo Proyecto.asm



Construcción

Para que el sistema de construcción pueda ubicar las variables y programas se tiene un archivo de configuración de memoria **Proyecto.cmd**, el cual indica al constructor(linkeador) que genere un archivo **Proyecto.map** para visualizar las posiciones de memoria de las variables:

Fig. 4.3 Archivo Proyecto.cmd

```
Proyecto.obj
-o  Proyecto.out
-m Proyecto.map
MEMORY
{
PAGE 0 : LOC: origin = 04000h, length = 04000h
PAGE 1 : DATA: origin = 07550h, length = 01000h
         VARS:  origin = 0560h, length = 01000h
}
SECTIONS
{
.text      > DATA PAGE 1
.cinit     > PROG PAGE 0
.vectors   > VECT PAGE 0
.bss       > VARS PAGE 1
.const     > DATA PAGE 1
.sysmem > DATA PAGE 1
.stack     > PROG PAGE 2
.data      > CHECK PAGE 11
.vars      > VARS PAGE
}
```

Para iniciar la ejecución se deben seguir la siguiente secuencia de comandos:

```
F:\dspuni\compilaC\bin\dspa -l -k -v2xx Proyecto.asm
F:\dspuni\compilaC\bin\dsplnk Proyecto.cmd
F:\dspuni\coff-dsk\coff-dsk Proyecto.out  Proyecto.dsk
```

El Archivo de mapa **Proyecto.map** nos ayuda a visualizar las posiciones de las variables y el inicio del programa:

Fig. 4.4 Archivo Proyecto.map

```
OUTPUT FILE NAME:   <Proyecto.out>
ENTRY POINT SYMBOL: "_c_int0"  address: 000041a9
0000901c _a                    000044d2 F$$GT
00004360 _atexit               000044d5 F$$FTOI
00009004 _b                    00004516 F$$NEG
000041a9 _c_int0               0000451e cinit
00004332 _exit                 0000451e etext
00004284 _f$$error             00008000 .data
00009001 _i                    00008000 edata
00004000 _main                 00009000 .bss
00009002 _pi                   00009001 _i
00004382 _sin                  00009002 _pi
00009804 _x                    00009004 _b
00009034 _y                    0000901c _a
```

Al final se genera un archivo **Proyecto.dsk**(extensión DSK) que puede ser cargado en la Tarjeta DSP mediante el Programa Monitor Code Explorer.

4.3 Formato de Archivo de Datos(GO DSP) en Tarjeta DSP

Los Archivos GO DSP son archivos de texto con una línea de cabecera y luego los datos se almacenan de una muestra por línea. Los datos pueden ser cualquiera de los siguientes formatos: Hexadecimal, Entero, Largo o Flotante.

La cabecera para archivos de datos GO DSP tienen el siguiente formato:

NúmeroMágico Formato DireccióndeInicio NúmerodePágina Longitud

donde :

NúmeroMágico, se establece a 1651.

Formato, es un valor numérico de 1 a 4 indicando el formato de las muestras en el archivo. El número corresponde a las opciones listadas anteriormente. Es decir, un Formato de 1 indica la notación de Hexadecimal mientras un Formato de 4 indica los números de punto Flotante.

Direcciónde Inicio, es la dirección de inicio del bloque grabado.

NúmerodePágina, es el número de página del bloque tomado.

Longitud, es el número de muestras en el bloque.

Ejemplo de un archivo GO DSP:

```
1651 1 800 1 3
0x0000
0x0000
0x0000
```

4.4 Acceso de Archivos en Tarjeta DSP(FileIO)

El Code Explorer permite trasmitir datos hacia(o de) un archivo destino en PC. Esta es una gran manera de simular el código de programa que utiliza los valores

muestreados. Notar que ésta característica de Entrada/Salida de archivo que no se considera suficiente para satisfacer los requerimientos de acceso a tiempo-real.

El File Input/Output(FileIO) ofrece el concepto de Punto de Prueba. Un punto de prueba permite al usuario el extraer/inyectar una prueba, o tomar una instantánea de situaciones de memoria en un punto que definió el usuario(es decir punta de prueba). Un punto de prueba puede ubicarse en cualquier punto del algoritmo(similar a la manera que un punto de ruptura o breakpoint es fijo). Cuando la ejecución del programa alcanza el punto de prueba, el objeto conectado(si es un archivo, gráfico o ventana de memoria) se actualiza. Una vez el objeto conectado se actualiza, la ejecución del programa continúa. Utilizando éste concepto, si nosotros ubicamos un Punto de Prueba específico en el código, y entonces conectamos un archivo a éste punto de prueba, podemos llegar a implementar las funcionalidades de un Archivo de Entrada/Salida.

Un archivo puede asociarse con una señal de entrada o una señal de salida. Un punto de prueba específico, puede leer o escribir datos de un archivo especificado.

NOTA: Cuando se selecciona un nombre de archivo para salida, el archivo permanece abierto para escribir muestras hasta que se quita de la lista de Archivos de Salida. Por tanto, no se puede ver éste archivo mientras el archivo esta agregado en la lista de Archivos de Salida.

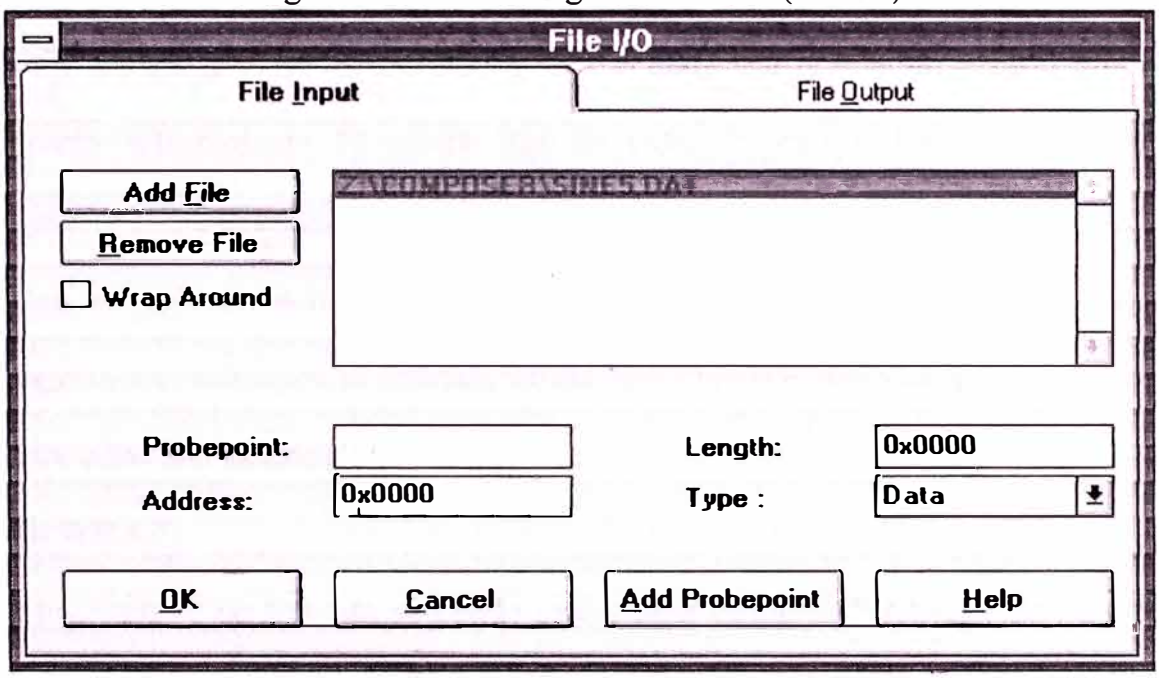
Transmisión de Datos a/de un Archivo:

1. Ubicar un Punto de Prueba: antes de especificar la información sobre el archivo, es mejor seleccionar el punto de prueba y dejarlo no conectado. El punto de prueba le dice a Code Explorer CUANDO quiere empezar a fluir los datos de/a un archivo. Es decir, una vez la ejecución del código alcanza éste punto, Code

Explorer actualizará(o leerá de) el archivo que se conecta al punto de prueba. Una vez que termine, empezará de nuevo. Para establecer un punto de prueba, simplemente ubicar el cursor al punto requerido, clic derecho y del menú emergente seleccionar Punto de Prueba.

2. Seleccionar del Menú FILE el comando FILE I/O: una vez seleccionado, se solicitará la información específica para el FILE I/O. La primera pestaña de la caja de diálogo contiene la información con respecto a Archivos de Entrada y la segunda pestaña es para Archivos de Salida.

Fig. 4.5 Cuadro de Ingreso de Datos(FileIO)



3. Escoger el archivo específico: seleccionando el botón ADD FILE para la pestaña de entrada o salida de archivos. Notar que más de un archivo puede ser configurado para entrada o salida. El archivo de datos puede ser un archivo objeto COFF o un archivo con el formato GO DSP.
4. Los campos de la lista corresponden a la información específica de cada archivo. Y son asociados con el archivo que se selecciona. Por consiguiente, al

seleccionar un archivo pulsando el botón en su nombre se necesita conectar el archivo a un punto de prueba particular. Cuando el archivo se crea por primera vez, no se conecta a un punto de prueba. Esto se indica como "No Connected". Al conectar un archivo a un punto de prueba aparecerá su condición como "Connected".

5. Para cada archivo seleccionado, ingresar una **dirección de inicio, longitud** y **tipo de valor**. El campo **dirección de inicio** corresponde a dónde se quiere transferir los datos de el archivo de entrada o a el archivo de salida. El campo **longitud** indica cuántas muestras serán transferidas al destino seleccionado(archivo de entrada o salida) cada vez que se alcanza el punto de prueba seleccionado. El campo **tipo de valor** indica que los datos pueden ser transferidos a la sección DATA o PROGRAM seleccionando el tipo apropiado.
6. Una vez seleccionado el botón de OK, Code Explorer verificará los parámetros ingresados realizando la Entrada/Salida de archivos cuando los puntos de prueba asociados son alcanzados.

Modo WRAP

Para un Archivo de Entrada se puede seleccionar el modo "WRAP". Este modo se utiliza para reiniciar la lectura de un archivo cuando se alcance el final, para luego acceder al inicio del archivo nuevamente. Esta característica es útil cuando se necesita generar una señal periódica de un archivo(por ejemplo onda seno). Si el modo "WRAP" no se selecciona y se alcanza fin de archivo, se mostrará un mensaje que indicará que se alcanzó el final del archivo y el programa DSP se detendrá.

4.5 Desarrollo del Software

El software de Compresión y Descompresión se desarrolló en base al Algoritmo de Codificación Aritmética implementado utilizando el Lenguaje C, para posteriormente ser compilado y generado en un formato compatible con la Tarjeta DSP.

Se trabajó para el Proyecto con los siguientes compiladores:

a.) Borland C: es un compilador de lenguaje C que permite crear programas para una computadora personal(PC). Se utilizó para crear el proyecto de compresión de datos y simular el funcionamiento de transferencia de datos de la tarjeta DSP.

b.) Code Composer: es un compilador de lenguaje C que permite realizar programas y emular su funcionamiento en un procesador DSP. Se utilizó para depurar el funcionamiento del proyecto.

c.) Code Explorer: el monitor de la Tarjeta DSP trabaja con el lenguaje ensamblador, por tanto, todo programa realizado en lenguaje C debe convertirse a un formato compatible con la tarjeta.

4.6 Conversión de Archivos a Formato GO DSP

Considerando que la información será ingresada a la Tarjeta DSP, los archivos de datos deben convertirse previamente al formato GO DSP en hexadecimal.

El Programa CODDAT.C se desarrollo en Lenguaje C para trabajar en la PC y permite obtener la longitud de un archivo y convertir a su equivalente en formato hexadecimal, la figura 4.6a muestra un extracto de dicho programa(ver Anexo D Programa Completo).

Fig. 4.6a Programa CODDAT.C

```
void main (int argc, char *argv[])
{
  fprintf(fich_cfg,"1651 1 0 1 0\n");// Genera Archivo de Configuración
```

```
  curpos = ftell(fich_ent); fseek(fich_ent, 0L, SEEK_END);
  length = ftell(fich_ent);  fseek(fich_ent, curpos, SEEK_SET);
  nMSB = length/256; nLSB = length % 256;
  fprintf(fich_cfg,"0x%.4X\n", nMSB ); fprintf(fich_cfg,"0x%.4X", nLSB );
  // Genera Archivo de Salida en Formato Hexadecimal
  fprintf(fich_sal,"1651 1 0 1 0");
  while( Salir )
    {
     fscanf(fich_ent,"%c",&Car);
     if ( feof(fich_ent) ) { Salir = 0; break; }
     fprintf(fich_sal,"\n0x%.4X",Car);
    }
  Cerrar_Ficheros();
}
```

El Programa DECDAT.C permite obtener de un archivo en formato hexadecimal a su equivalente en formato original, la figura 4.6b muestra un extracto.

Fig. 4.6b Programa DECDAT.C

```
void main (int argc, char *argv[])
{
  // Obtiene Longitud de Archivo de Configuración
  fscanf(fich_cfg, "%s%s%s%s%s",
                    &sNum1,&sNum2,&sNum3,&sNum4,&sNum5);
  fscanf(fich_cfg, "%x%x", &cnMSB, &cnLSB);
  nMSB = (int)cnMSB; nLSB = (int)cnLSB%256;
  L = nMSB*256 + nLSB;

  // Forma Archivo Original
  fscanf(fich_ent, "%s%s%s%s%s",
                     &sNum1,&sNum2,&sNum3,&sNum4,&sNum5);
  while( Cont < L )
    {
     fscanf(fich_ent, "%x", &Num);
     fprintf(fich_sal,"%c", Num);
     Cont++;
    }
```

4.7 Evaluación de tiempos en la Tarjeta DSP

Se diseñó un programa llamado Prueba Básica en Vacío para lectura y escritura de archivos utilizando la opción FileIO, el cual traslada información de un archivo de

entrada hacia un archivo de salida. La figura 4.7 muestra el código fuente del programa.

Se puede observar que cuando el programa alcanza la dirección referida a la Sentencia "Control = 0", la opción FileIO lee de un archivo(previamente especificado en la PC) un bloque de datos de 256bytes y lo almacena en la memoria de la tarjeta DSP a un arreglo llamado BufferIn. De igual manera sucede cuando se alcanza la sentencia "Control = 1" se escribe 256bytes del BufferOut(en la memoria de la tarjeta DSP) hacia un archivo en disco.

La Tabla 4.2 describe los tiempos de respuesta del programa de la figura 4.7, para diferentes valores de tamaño de bloques de lectura y escritura.

Fig. 4.7 Programa de Prueba Básico en Vacío

```
#define kRegInSize   256  // ← Número de Muestras de lectura FileIO
#define kRegOutSize 256 // ← Número de Muestras de escritura FileIO

// Variables de Intercambio de Información de Archivos
char  BufferIn[kRegInSize], BufferOut[kRegOutSize];

void main( )
{
  int     NumIn = kRegInSize, NumOut = kRegOutSize;
  int     NumBytes = 0, i = 0, Salir = 1, Control;
  /* Lee Archivo */
  while( Salir ) /* Lazo Principal : 1 vuelta */
    {
        /* Punto de Prueba FileIO In */
         Control = 0; // ← Dirección de Memoria de Entrada
        /* Proceso Y = X , Transferencia de Datos de Entrada a Salida*/
        for ( i = 0; i<NumIn; i++ )
            BufferOut[i] = BufferIn[i];
        NumBytes = NumBytes + kRegInSize;
        if ( NumBytes >= NumTotalIn ) Salir = 0;
        /* Punto de Prueba FileIO Out */
         Control = 1; // ← Dirección de Memoria de Salida
    }
}
```

Tabla 4.2 Tiempos de Respuesta del Algoritmo Básico en Vacío para un archivo de 6kbytes y acceso DMA por contador

Acceso	**Buffer FileIO (bytes)**	**Tiempo (seg)**	**Bytes/seg**	**Tiempo 1Vuelta (seg)**
Archivo	**2048**	**32**	192	10.67
Archivo	**1024**	**30**	204.8	5
Archivo	**512**	**30**	204.8	2.5
Archivo	**256**	**32**	192	1.33
Archivo	**1**	**150**	40.96	0.0244
Memoria 10000v	**1024**	**40**	250k	0.004

Considerando que la opción FileIO se realiza mediante comunicación serial, se observa que a mayor tamaño del bloque de datos más tiempo toma el proceso, por tanto de la Tabla se deduce que el valor de 512 bytes para el tamaño del bloque de datos es el óptimo.

Luego se modificó el programa de la figura 4.7 para repetir 10000 veces el Lazo Principal simulando el acceso directo a memoria(es decir sin comunicación serial) y se observó que la velocidad de respuesta mejora en 1250.

4.8 Programa Orientado a la PC

El programa de Compresión(CODARI.C) y Descompresión(DECARI.C) se desarrolló utilizando el Compilador BorlandC++ v3.1 para DOS. En la figura 4.8 se muestra un extracto del programa donde la información es leída de archivos almacenados en disco y se procesa uno a uno(Programa Completo en Anexo D).

Fig. 4.8 Programa de Compresión CODARI.C

```
void main (int argc, char *argv[])
{
 unsigned char ch;
 Comprobar_Parametros (argc); Abrir_Ficheros (argv[1], argv[2]);
```

```
Inicializar_Modelo ();
  while ((fscanf (fich_ent, "%c", &ch)) != EOF)
  {
    Codificar_Simbolo (ch);
    Actualizar_Modelo (ch);
  }
  Codificar_Simbolo (Simbolo_EOF);
  Escribir_Bits_Pendientes ();
  Escribir_Ultimo_Byte ();
  Cerrar_Ficheros ();
}
```

4.9 Programa Orientado a la Tarjeta DSP

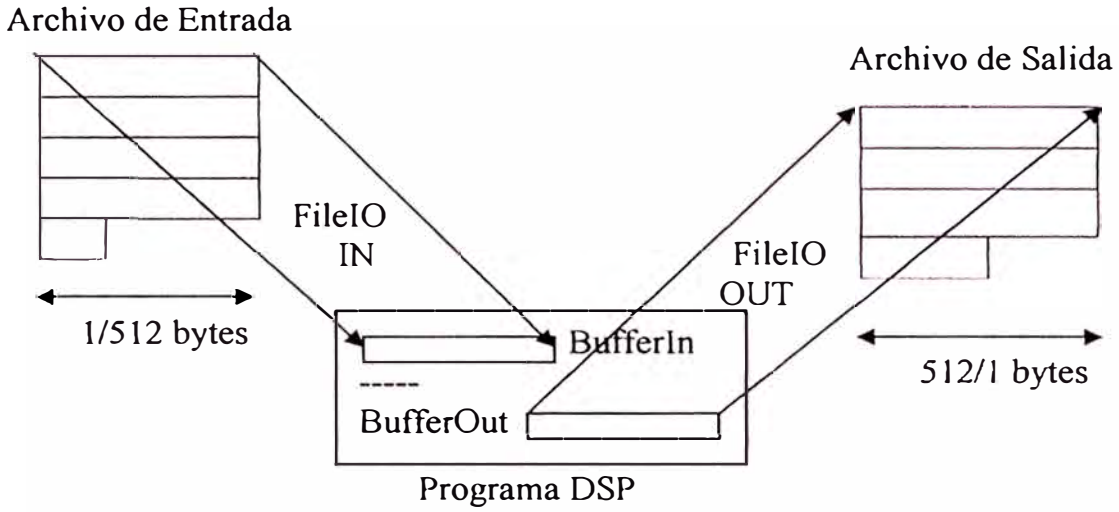
Debido a que el Programa Fuente se desarrolló en BorlandC se tuvieron que hacer algunas modificaciones para su posterior implementación en una Tarjeta DSP:

- La lectura y escritura se realizó mediante bloques de datos de 1 y 512bytes, para almacenar posteriormente en un Buffer de Entrada y uno de Salida, y utilizar la opción FileIO del Code Explorer de intercambio de archivos.
- Ingresar al programa la longitud del archivo de datos previo a la ejecución, para no leer indefinidamente dicho archivo debido a que se activó la opción WRAP del FileIO.
- Trabajar con programas adicionales de conversión de formatos, el Code Explorer trabaja con archivos en formato hexadecimal.
- Trabajar con programas que trunquen datos, la opción FileIO del Code Explorer escribe bloques de datos de un tamaño definido y considerando la longitud de la información final no múltiplo del tamaño del bloque de escritura se van a generar datos adicionales en el archivo de respuesta los cuales necesitan ser eliminados.

Debido a que la Tarjeta DSP no realiza procesamiento de archivos de la PC, se utilizó la opción de depuración FileIO para transferencia de archivos. Por ésta razón

en el Programa Original, sólo cambia el modo de acceso a los datos, para adaptarse a la Tarjeta DSP. En la figura 4.9 se muestra la forma de acceso del Programa DSP hacia los archivos en PC utilizando la opción FileIO.

Fig. 4.9 Esquema de Trabajo



Los Programas de Compresión y Descompresión son muy similares y trabajan con las siguientes variables y funciones principales:

- **BufferIn[1 ó 512]**: es un arreglo de tipo unsigned char que almacena la información que va a ser procesada.
- **BufferOut[512 ó 1]**: es una arreglo de tipo unsigned char que almacena la información procesada por el DSP.
- **SizeFileOri[2]**: es del tipo unsigned char y representa la longitud del archivo sin comprimir(Longitud = SizeFileOri[0]*256 + SizeFileOri[1]).
- **SizeFileCod[2]**: es del tipo unsigned char y representa la longitud del archivo procesado(Longitud = SizeFileCod[0]*256 + SizeFileCod[1]).

Las siguientes funciones representan los **puntos de prueba** para utilizar con la opción FileIO:

- **FileIO_In()**: el programa al llegar a solicitar ésta función lee 1 ó 512 datos de un archivo en disco y los almacena en BufferIn.
- **FileIO_Out()**: el programa al llegar a solicitar la función, escribe los 1 ó 512 datos almacenados en BufferOut hacia un archivo en disco.
- **FileIO_Oricfg()**: al solicitar ésta función lee 2 datos de un archivo en disco y los almacena en SizeFileOri.
- **FileIO_Codcfg()**: al ser requerido escribe la información de SizeFileCod hacia un archivo en disco.

De manera similar al Programa de Prueba Básico en Vació descrito en la figura 4.7, el funcionamiento es como sigue:

1. El Programa se inicializa con la longitud del archivo original mediante la función FileIO_Oricfg().
2. Se realiza el procesamiento de la información, teniendo como condición de finalización la longitud del archivo previamente ingresado.
3. Mientras el programa se encuentra en procesamiento se lee con la función FileIO_In() un nuevo bloque de datos, y luego la información procesada ubicada en BufferOut, es escrita a disco mediante la función FileIO_Out().
4. Finalmente la longitud del archivo procesado se escribe a disco con la función FileIO_Codcfg().

En la figura 4.10 se muestra un extracto del Programa Codificador en Vacío, es decir, considerando como procesamiento sólo transferencia de información de un archivo de entrada a la salida.

Fig. 4.10 Programa de Codificación en Vacío

```
/*************** Rutinas de Archivos ***************/
void FileIO_In( )       /* Lectura de kRegInSize Bloque de Datos de  */
{ Control = 1; }        /* Archivo Fuente  para almacenar en BufferIn */
void FileIO_Out( )      /* Escritura de kRegOutSize Bloque de Datos de */
{  Control = 2; }       /* BufferOut hacia Archivo Comprimido          */
void FileIO_OriCfg( )  /* Lectura de Datos de Configuración */
{  Control = 0; }       /* Longitud de Archivo Original       */
void FileIO_CodCfg( )  /* Escritura de Datos de Configuración    */
{   Control = 3; }      /* Longitud de Archivo Comprimido       */
/************************ Programa Principal ***************************/
int main( )
{
  unsigned char chcod;
  unsigned int  Cont = 0, Salir = 1, chcodi;
  unsigned int  TamFile = 0, TamFileIn = 0, TamFileOut = 0;
  Inicializa_Buffers();
  FileIO_OriCfg();/* Lectura de Longitud de Archivo Original */
  TamFileIn = SizeFileOri[0]*256 + SizeFileOri[1];
  while(Salir)
   {
    Indice_BufferIn = 0; Cont = 0;
    FileIO_In();/* Lectura de Bloque de Datos del Archivo Original */
    while( (Cont < NumIn) && (TamFile < TamFileIn) )
        {
         chcod = BufferIn[Indice_BufferIn++]; chcodi = chcod*1;
         BufferOut[Indice_BufferOut++] = chcod;
          if ( Indice_BufferOut >= NumOut )
           {
            FileIO_Out();/*Escribe Bloque de Datos al Archivo Comprimido*/
            Indice_BufferOut = 0;  NumBytesOut = NumBytesOut + NumOut;
           }
         TamFile = NumBytesIn + Indice_BufferIn;
         Cont = Indice_BufferIn;
        }
    NumBytesIn = NumBytesIn + NumIn;
    if ( NumBytesIn >= TamFileIn ) Salir = 0;
   }
  TamFileOut = NumBytesOut + Indice_BufferOut;
  FileIO_Out();/* Escritura de Bloque de Datos al Archivo Comprimido */
  SizeFileCod[0] = TamFileOut/256; SizeFileCod[1] = TamFileOut%256;
  FileIO_CodCfg(); /* Escritura de Longitud de Archivo Comprimido */
  return(0);
}
```

La Tabla 4.3 muestra los tiempos de respuesta aproximados del Programa de Codificación en Vacío en la Tarjeta DSP con bloques de datos de I/O de 512 bytes.

Tabla 4.3 Tiempos de Respuesta Programa de Codificación en Vacío

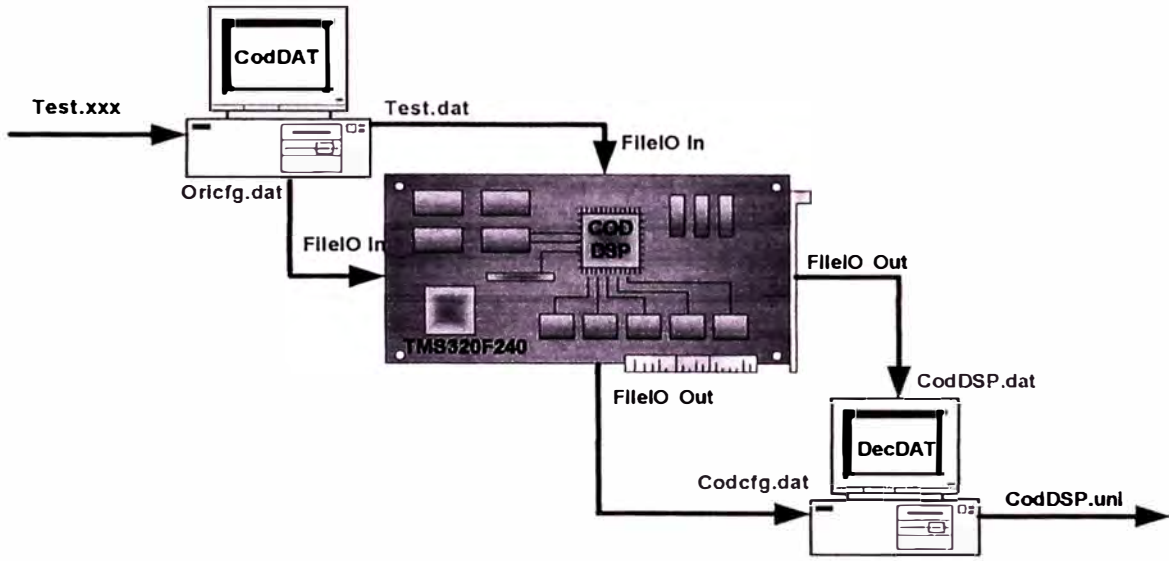
	Bytes	Tiempo
Texto	21	3 seg
Documento	1181	10.52 seg
Texto	2252	16.2 seg
Imagen	11764	1 min 16 seg
Excel	17408	1 min 45 seg
Aplicación	18944	1 min 56 seg

4.9.1 Programa de Compresión

El Programa permite comprimir cualquier tipo de archivo hasta un máximo de 65535 bytes. Ver Código Fuente Completo(CODDSP.C) en Anexo B .

En la figura 4.11 se muestra un diagrama de bloques para ejecutar la aplicación en la Tarjeta DSP.

Fig. 4.11 Esquema de Compresión en Tarjeta DSP

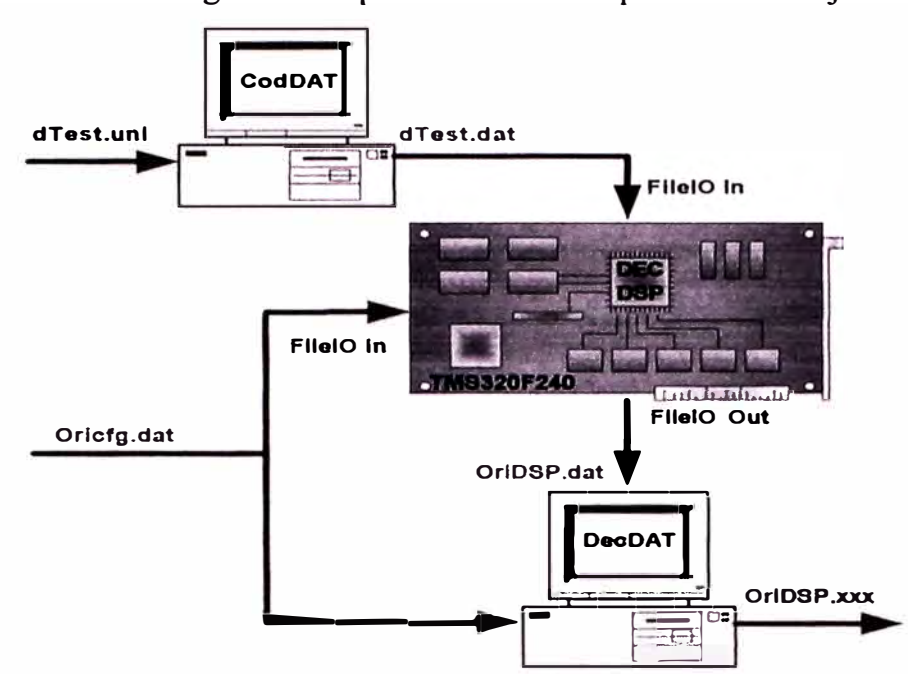


4.9.2 Programa de Descompresión

El Programa permite descomprimir los datos generados del programa anterior(compresión). Ver Código Fuente Completo(DECDSP.C) en Anexo C.

La figura 4.12 muestra un diagrama de la secuencia para implementar en la Tarjeta DSP.

Fig. 4.12 Esquema de Descompresión en Tarjeta DSP



A continuación se describen los diagramas de bloques:

1. **CODDAT.C**: programa que se ejecuta en la PC, convierte un archivo original(Test.xxx) a un nuevo archivo(Test.dat) en formato GO DSP, y también obtiene su longitud(Oricfg.dat) correspondiente.
2. **CODDSP.C** : el programa se ejecuta en la tarjeta DSP. Permite comprimir un archivo. Se debe de ingresar mediante FileIO el archivo a procesar(Test.dat), así como su longitud(Oricfg.dat), obteniendo como resultado el archivo comprimido(CodDSP.dat) y su longitud correspondiente(Codcfg.dat).
3. **DECDSP.C**: el programa se ejecuta en la Tarjeta DSP, permite descomprimir un archivo. Se debe de ingresar un archivo comprimido(dTest.dat) y la longitud del archivo original(Oricfg.dat) sin comprimir, obteniendo como resultado el archivo original(OriDSP.dat).

4. **DECDAT.C** : el programa que se ejecuta en la PC acepta como datos la longitud y un archivo en formato GO DSP(CodDSP.dat, OriDSP.dat) para convertirlo posteriormente a su formato original(CodDSP.uni, OriDSP.xxx) eliminando el excedente de bytes generados por la opción FileIO en el archivo.

4.10 Generación de Formato para Tarjeta DSP

Como se describió anteriormente la Tarjeta DSP trabaja con archivos de formato DSK, por tanto un archivo debe ser compilado y procesado para ese fin. A continuación se muestra la secuencia de comandos necesarios para generar una aplicación DSK a partir de un proyecto en lenguaje C.

```
F:\dspuni\compilaC\bin\dspcl -al -s -k -v2xx CodDSP.c -if:\dspuni\compilaC\include
F:\dspuni\compilaC\bin\dspa -l -k -v2xx CodDSP.asm
F:\dspuni\compilaC\bin\dsplnk CodDSP.cmd
copy CodDSP.out _Compila.out
F:\dspuni\coff-dsk\coff-dsk < _Compila.scr
copy _Compila.dsk CodDSP.dsk
```

4.11 Puesta en marcha del Proyecto

Teniendo el formato DSK se busca en el archivo de mapa de memoria(.MAP) las direcciones de memoria del punto de inicio de programa(Registro PC), así como de los datos requeridos para posteriormente configurar la opción FileIO de monitoreo en la Tarjeta DSP. Las variables y funciones necesarias son:

PC	**Compresión de Datos**			
0x43f2				
Entrada	**Test.dat**		**Oricfg.dat**	
	BufferIn	Longitud	SizeFileOri	Longitud
	0x9217	0x0200	0x900a	0x0002
	FileIO_In()	0x434f	FileIO_Oricfg()	0x4357
Salida	**CodDSP.dat**		**Codcfg.dat**	
	BufferOut	Longitud	SizeFileCod	Longitud
	0x9002	0x0001	0x900c	0x0002
	FileIO_Out()	0x4353	FileIO_Codcfg()	0x435b

PC	Descompresión de Datos			
0x4426				
Entrada	dTest.dat		Oricfg.dat	
	BufferIn	Longitud	SizeFileOri	Longitud
	0x9001	0x0001	0x900b	0x0002
	FileIO_In()	0x438c	FileIO_Oricfg()	0x4394
Salida	OriDSP.dat			
	BufferOut	Longitud		
	0x921a	0x0200		
	FileIO_Out()	0x4390		

En la figura 4.13a se observa el Programa DSP en funcionamiento y en la figura 4.13b la asociación de FileIO con las variables y archivos.

Fig. 4.13a Programa en Tarjeta DSP

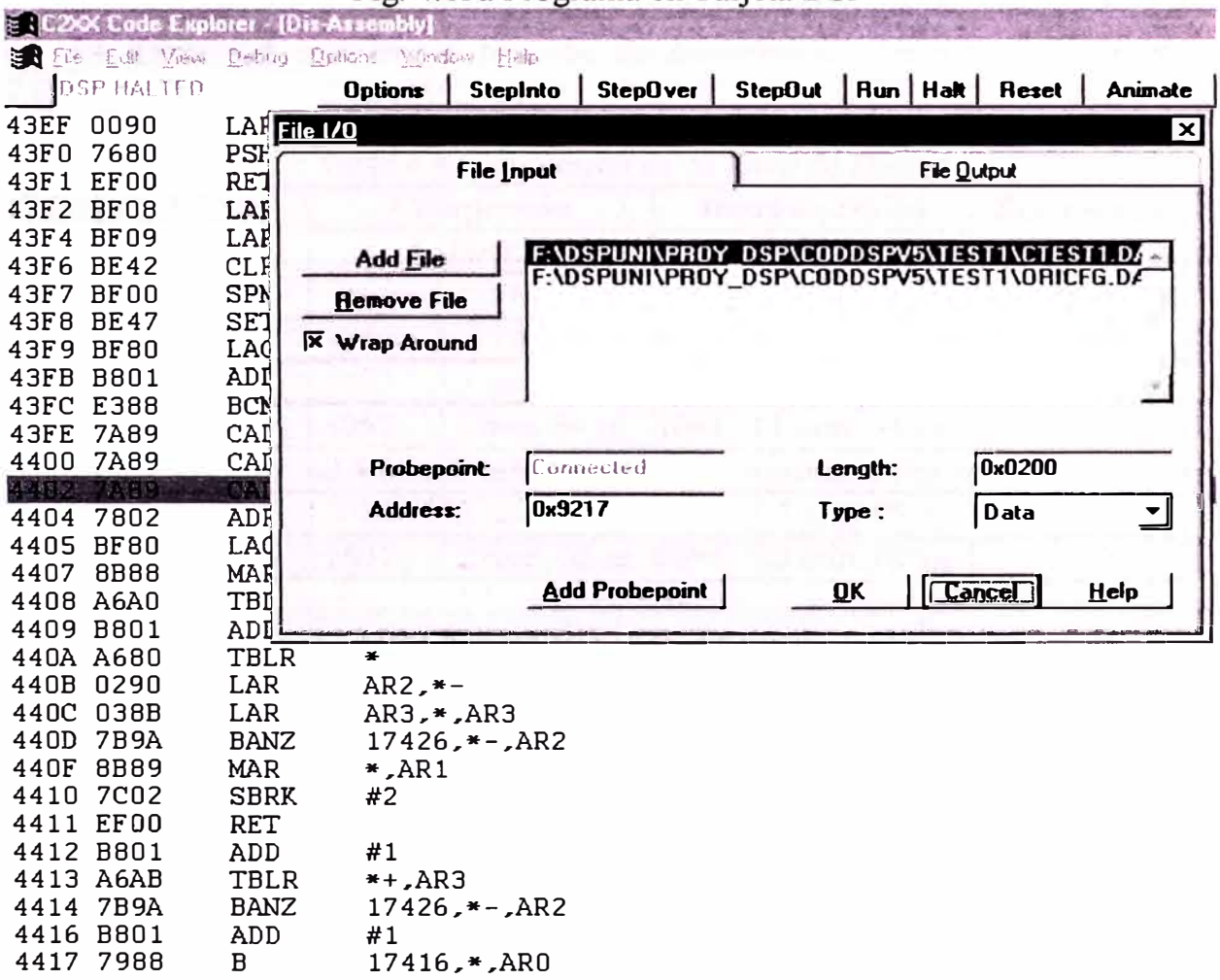
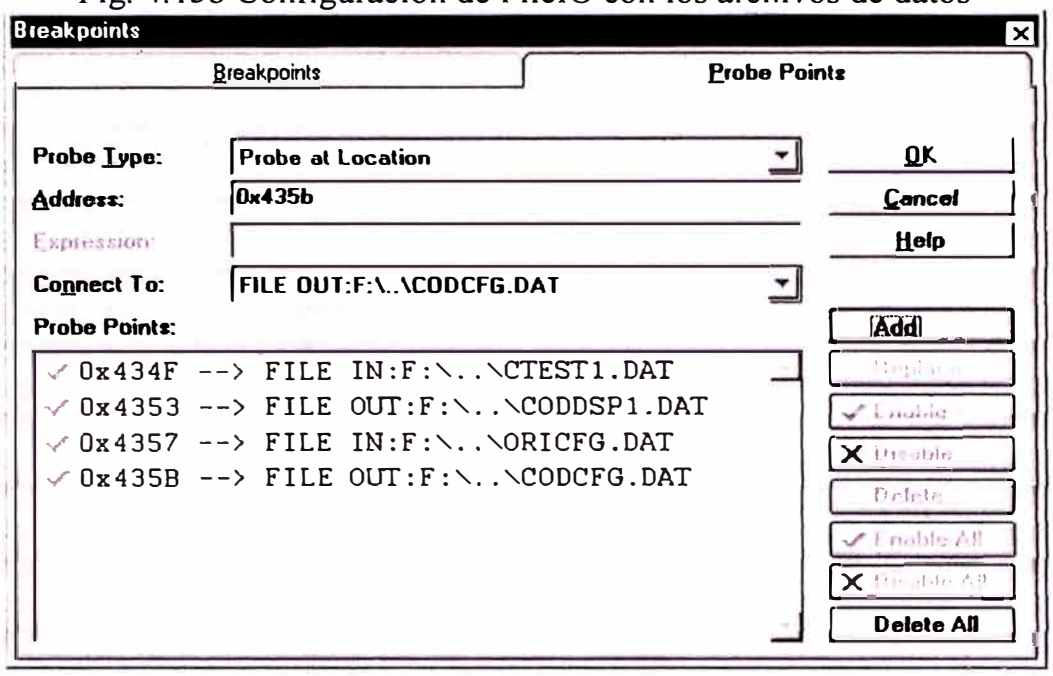


Fig. 4.13b Configuración de FileIO con los archivos de datos



La Tabla 4.4 muestra la relación de compresión y tiempos del programa ejecutado en la Tarjeta DSP.

Tabla 4.4 Compresión de Archivo de Datos

	Test	Compresión CODDSP		Descompresión DECDSP		Relación de Compresión %
		Bytes	Tiempo	Bytes	Tiempo	
1	TXT	21	4 sg.	19	4 sg	9.5
2	DOC	1181	1 min. 24 sg.	758	1 min. 24 sg.	35.8
3	JPG	1061	1 min. 54 sg.	1043	1 min. 54 sg.	1.7
4	BMP	2702	1 min. 39 sg.	855	1 min. 39 sg.	68.35
5	EXE	3584	3 min. 07 sg.	1675	3 min. 07 sg.	53.26
6	WAV	1837	2 min. 02 sg.	1094	2 min. 02 sg.	3.42

Este capítulo describió los detalles de programación en el DSP de cada uno de los módulos que integran el sistema de compresión, describiendo la programación de la etapa de entrada, el bloque de compresión y la etapa de salida.

CONCLUSIONES

El proyecto desarrollado ha permitido utilizar una infraestructura tecnológica(Tarjeta de Evaluación DSP) necesaria para el desarrollo de técnicas de compresión de datos. Como resultado del mismo se pueden resumir las siguientes conclusiones:

1. Se desarrolló un trabajo relacionado con compresión de datos usando procesadores digitales de señales(tarjeta de evaluación del DSP TMS320F240), lo cual abre un importante campo de investigación en ésta temática.
2. Se programó y comprobó una compresión de datos usando el método de codificación aritmética, validando de forma práctica los resultados teóricos alcanzados hasta ese momento.
3. Se realizó la comparación entre los resultados obtenidos en simulación, usando el Compilador de Lenguaje C Borland y el Code Composer, con los resultados obtenidos en la realización práctica en el DSP, lográndose resultados satisfactorios.
4. Desde el punto de vista de investigación, la experiencia adquirida en la puesta en marcha de éste sistema, permite contar con una valiosa herramienta para trabajos futuros basados en esta técnica, tanto desde el punto de vista de hardware, software, así como en técnicas de simulación.

5. Desde el punto de vista académico, el trabajo permitió adquirir conocimientos y habilidades en el manejo de herramientas de programación relacionadas con los DSPs de Texas Instruments, el uso de algoritmos de compresión, así como las herramientas de simulación mencionadas.
6. En el proceso de desarrollo no se trabajó con el Code Composer, en cambio se diseñó en Borland C y luego de adaptar el código se implementó en el Code Explorer de la Tarjeta DSP.
7. El Tamaño del Buffer de Lectura/Escritura para utilizar con la opción FileIO de la Tarjeta DSP se seleccionó a 512 bytes, debido a que no hay mucha variación en los tiempos de ejecución entre 256 y 2048 bytes, teniendo así mejor velocidad y promedio de bytes leídos.
8. Al ejecutar el Algoritmo en Vacío en la Tarjeta DSP con un Buffer de Entrada/Salida de 1024 bytes, y accediendo a la información directo de memoria (es decir sin comunicación serial) se comprueba que la velocidad del procesamiento aumenta en 1250 veces, para un Ciclo de Lectura/Escritura de Programa con la opción FileIO.
9. El considerar valores mayores para el Buffer de Entrada/Salida con la opción FileIO de la Tarjeta DSP, haría más lento al programa debido a que se realiza mediante comunicación serial.
10. El presente trabajo pretende comprobar que se puede implementar técnicas de compresión en hardware.
11. El Algoritmo desarrollado es del Tipo General.

12. La aplicación del Proyecto sería formar parte de un sistema en el cual se comprima bloques o tramas de datos para su posterior envío y transmisión(por ejemplo en tarjetas de red).
13. El Proyecto comprueba que se puede mejorar la capacidad de almacenamiento, utilizando dispositivos de hardware(DSPs).
14. Se puede contar con un sistema de compresión de información a bajo costo.
15. El tiempo de respuesta del programa es lento para archivos superiores a 5 kbytes, debido que la carga de información de los archivos de la PC hacia el DSP y viceversa se realiza mediante comunicación serial a través de la opción FileIO de la Tarjeta DSP.
16. El tamaño máximo del archivo a procesar mediante la Tarjeta DSP es de 65535 bytes, debido que la variable que representa la Longitud del Archivo(TamFileIn) es del tipo unsigned int.
17. El Programa de la Tarjeta DSP trabaja con archivos de configuración, en los cuales se representa la longitud del archivo.

RECOMENDACIONES

Como se ha explicado anteriormente, el trabajo constituye una primera aproximación al desarrollo de técnicas de compresión de datos. Por tal motivo, no se ha podido abarcar toda la amplia gama de aplicaciones que éstas técnicas ofrecen en la actualidad. La continuidad y profundización de éste trabajo en el futuro constituye un interés de los profesores, investigadores y estudiantes vinculados al tema, por lo que se proponen las siguientes recomendaciones para darle un seguimiento al mismo:

1. Desde el punto de vista de **hardware**, se mejoraría considerablemente el tiempo de respuesta del programa si la transferencia de archivos desde la PC al DSP se realizara mediante el Puerto Paralelo con el conector JTAG, o con Acceso Directo a Memoria
2. Debido a que el Programa se desarrolló íntegramente en Lenguaje C con el Compilador Borland, el cual está estructura para trabajar en una PC, y luego se adaptó para su desempeño en la Tarjeta DSP, se podría optimizar los tiempos de respuesta del programa si se convierte algunas secciones del programa fuente en C a su equivalente en Lenguaje Ensamblador, teniendo así un programa mixto de mejor desempeño.
3. En cuanto a la **implementación de algoritmos**, agregar al sistema diseñado algunas posibilidades no contempladas en ésta primera versión como por ejemplo, conversión de bits directo con código ensamblador.

4. Desarrollo de una **interfaz de usuario** que permita al operador del sistema mayor flexibilidad en la entrada de parámetros del control y visualización de resultados alcanzados, sin necesidad de conocer a profundidad el ambiente de desarrollo de los procesadores de Texas Instruments.

ANEXO A

MÓDULO DE DESARROLLO TMS320F240 DSP STARTER KIT

SPECTRUM DIGITAL
INCORPORATED

F24X DSK

Setup and Tutorial

1999

DSP Development Systems

F24X DSK
Setup and Tutorial

504706-0001 Rev. A
July 1999

SPECTRUM DIGITAL, INC.
10853 Rockley Road Houston, TX. 77099
Tel: 281.561.6952 Fax: 281.561.6037
sales@spectrumdigital.com www.spectrumdigital.com

About This Manual

This document describes how to install the hardware and software that comes with F24X DSK. In addition it provides a brief tutorial on how to use the Code Explorer debugger.

Notational Conventions

This document uses the following conventions.

The following will sometimes be referred to as the DSK: "TMS320F24X DSP Starter Kit", "F240 DSP Starter Kit", "F243 DSP Starter Kit", "F240 DSK, "F243 DSK"

The "Spectrum Digital Symbolic Assembler for the F24x DSP" will sometimes be referred to as the SD24XASM

Program listings, program examples, and interactive displays are shown is a special italic typeface. Here is a sample program listing.

equations
!rd = !strobe&rw;

Information About Cautions

This book may contain cautions.
This is an example of a caution statement.
A caution statement describes a situation that could potentially damage your software, or hardware, or other equipment. The information in a caution is provided for your protection. Please read each caution carefully.

Related Documents

Spectrum Digital F240 DSK Technical Reference
Spectrum Digital F243 DSK Technical Reference
Spectrum Digital Symbolic Assembler for the F24X DSP Technical Reference
Texas Instruments TMS320F240 Users Guide
Texas Instruments TMS320F243 Users Guide
Texas Instruments TMS320C2XX Fixed Point Assembly Language Users Guide
Texas Instruments TMS320C2XX Fixed Point C Language Users Guide
Texas Instruments TMS320C2XX Fixed Point C Source Debugger Users Guide

Chapter 1

Setup of the F24X DSK

This chapter provides you with a description of how to setup the F24X DSK and install the debugger and assembler.

Topic **Page**

1.0 Overview of the F24X DSK Development System

The F24X DSK Development System is a complete development system that allows engineers, programmers, students, and evaluators examine certain characteristics of the TMS320F24X Digital Signal Processor(DSP). In using this system you can:

- Develop and debug software algorithms
- Use hardware features of the DSP
- Prototype custom logic

This document is intended to help you set up the DSK Development System, install the assembler and debugger, and provide a small tutorial.

For more information on the specific components of the F24X DSK Development System refer to the following documents:

- Spectrum Digital F240 DSK Technical Reference
- Spectrum Digital F243 DSK Technical Reference
- Spectrum Digital Symbolic Assembler for the F24X DSP Technical Reference

1.1 What's In The Box

The following items should be in the DSK F24X Development System box:

__ F240 DSK or F243 DSK printed circuit card with TMS320F240 or TMS320F243 DSP

__ Power Supply for F24X DSK

__ 9 Pin serial port cable, F-M

__ Diskettes containing: Documentation (manuals in .pdf format), SD24XASM Assembler, and Code Explorer Debugger, Samples programs

__ Setup instruction sheet

__ Warranty card

If you did not receive all of these items contact the firm from whom your purchased the F24X DSK Development System.

1.2 What You'll Need

The following checklists detail items that you will need to use the F24X DSK Development System.

Hardware checklist

__ host	An IBM PC/AT or compatible PC or laptop with a hard-disk system, and a 1.44M floppy-disk drive
__ memory	Minimum of 32MB, additional memory may increase performance
__ display	color, VGA
__ serial port	One serial port
__ target	An F240 or F243 DSK printed circuit board with and power supply **(in the box)**
__ pointing device	A Microsoft-compatible mouse
__ miscellaneous materials	Blank, formatted disks

Software checklist

__ operating system	Microsoft Windows 95/98
__ debugger	Code Explorer **(in the box, on a diskette)**
__ assembler	Spectrum Digital SD24XASM symbolic Assembler **(in the box, on a diskette)**
__ editor	A system text editor like **"Edit"**, a word processing package will **not** work

1.3 Installing the SD24XASM Assembler and Code Explorer

This section contains the instructions to install the SD24XASM Symbolic Assembler and the Code Explorer debugger. To install this software perform the following steps:

1. Insert the supplied diskette labeled "**DSK24X System Disk, Disk 1**" into drive "A" of your PC.

2. From the Windows desktop execute the program "*a:Setup.exe*". During the install procedure you will be prompted to insert additional diskettes, e.g. "**DSK24X System Disk, Disk 2** " into drive A.

   The install shield will create the following directories:

   "c:\specdig\dsk24x"
   "c:\specdig\dsk24x\codeexplorer24x"
   "c:\specdig\dsk24x\sdasm24x"
   "c:\specdig\dsk24x\manuals"

   The install procedure will create a program folder for Code Explorer that can be run from the Windows desktop by running the mouse through "*start*", "*Programs*", "*Spectrum Digital DSK243*" and then clicking on "*Ce_c2xxw*".

1.4 Connecting the F24X DSK to Your Computer

This section contains the instructions for connecting the F24X DSK to your computer.

1. Shut down your computer and turn the power off to it.

2. Attach one end (female) of the 9 conductor serial cable to an available serial port on your PC (COM1, COM2, COM3, or COM4).

3. Attach the other end (male) of the 9 conductor serial cable to connector P4 on the F24X DSK.

4. Plug in the barrel connector on the DSK power supply to connector P3 on the DSK. The P3 connector is next to the 9 pin serial connector.

5. Plug in the DSK power supply to a 110/220 VAC outlet. The green LED (DS1) on the F24X should come on. This indicates the DSK has power.

6. Turn your computer back on. After the system comes up you should be able to enter Code Explorer from the Windows desktop by clicking on *"Start"*, running the mouse through "*Programs*" and "*Spectrum Digital DSK24X*" and then clicking on "*Ce_c2xxw*".

Figure 1-1 illustrates the system configuration with the DSK connected to your PC or laptop.

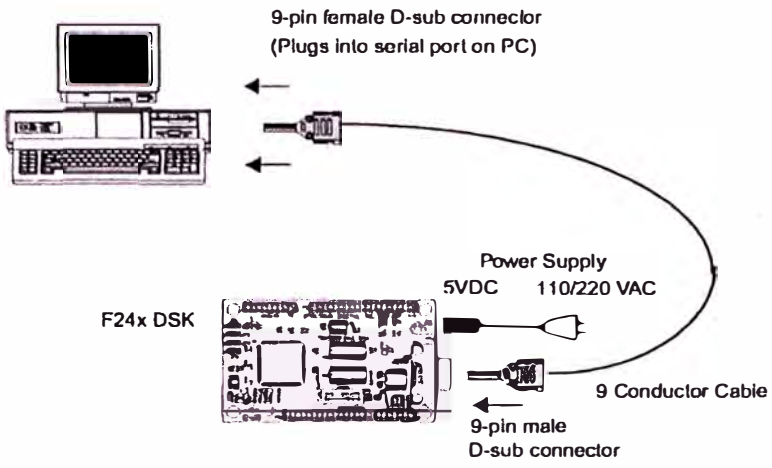


Figure 1-1 Connecting The DSK To Your Computer

Chapter 2

Code Explorer Tutorial

This chapter provides examples on how to invoke the Code Explorer Debugger, select a serial port, load a program, set break points, run the program in animation mode, and view memory and registers.

Topic **Page**

2.0 Starting The Debugger

To start the Code Explorer debugger from the Windows desktop click on "Start", run the mouse through "*Programs*" and "*Spectrum Digital DSK24X*", and then click on "*Ce_c2xxw*". The Code Explorer will come up. It should look like the following figure.

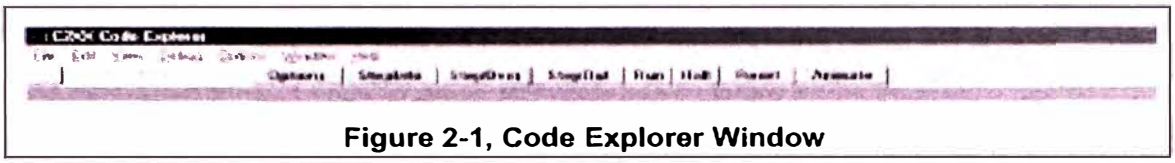


Figure 2-1, Code Explorer Window

Code Explorer has on line documentation which means any time you require information about a feature of Code Explorer it can be found by using the "HELP" pulldown.

2.1 Selecting the Serial Port

When Code Explorer first comes up it will present the prompt shown in the figure below. This allows you to select the serial port (COM1, COM2, COM3, or COM4) that you are going to use and it's I/O address. After you select the COM port enter its address, then click "OK". The I/O address for each COM port should be available from your BIOS or control Panel.

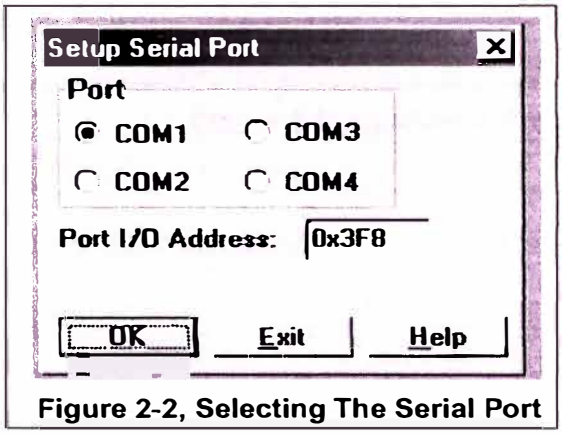


Figure 2-2, Selecting The Serial Port

2.2 Loading A Program

To load a program you must first make sure it has been assembled by the SD24xASM. The output of the SD24XASM is a file ending in "*dsk*". For instance a correct file name would be "*simple.dsk*". The figure below will load a file named "*simple.dsk*".

To load a file move the mouse over the "File" pulldown. The 1st selection is "Load Program". If you select "Load program" the following panel will appear. Traverse the directory structure until you locate your file, click on its name, then click the "OK" button.

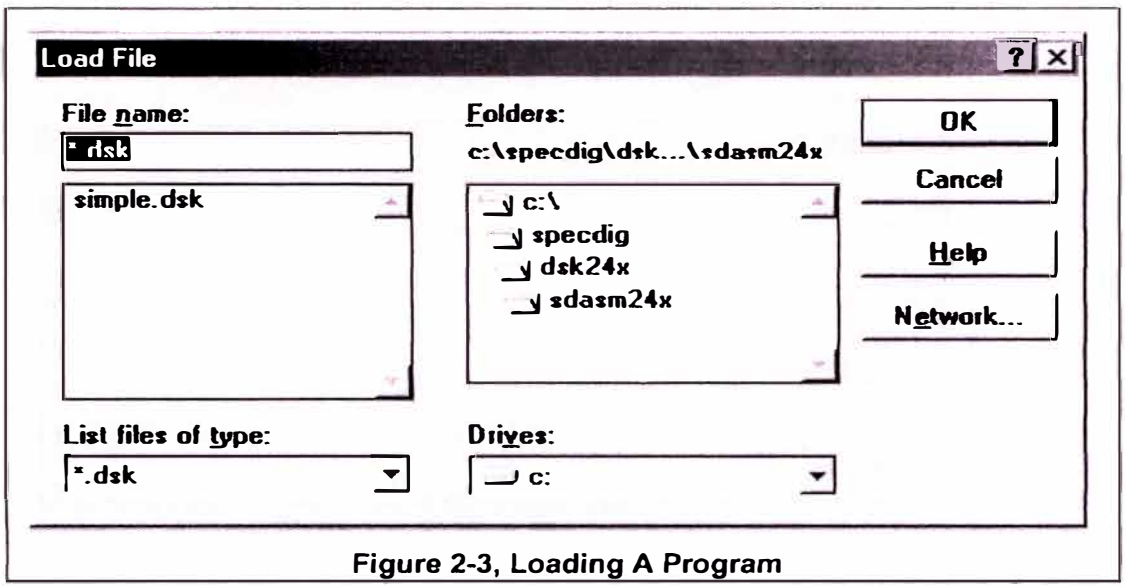


Figure 2-3, Loading A Program

2.3 Setting Breakpoints

To set a breakpoint run the mouse over the "Debug" pulldown, click on the first option "Set Breakpoints". The following panel will appear.

To enter a breakpoint puts its address on the second line and then click the "Add" button. When you have selected all of your breakpoints then press the "OK" button.

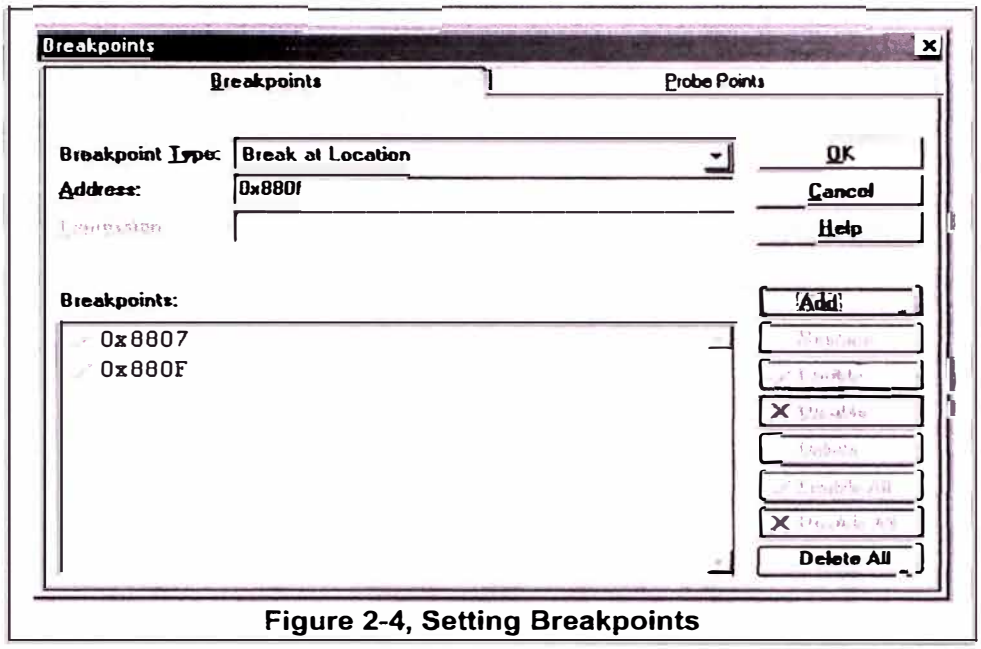


Figure 2-4, Setting Breakpoints

Figure 2-5 shows the breakpoints set at locations 0x8807 and 0x880F. These lines in the program will be highlighted in purple. The location of the program counter will be highlighted in yellow.

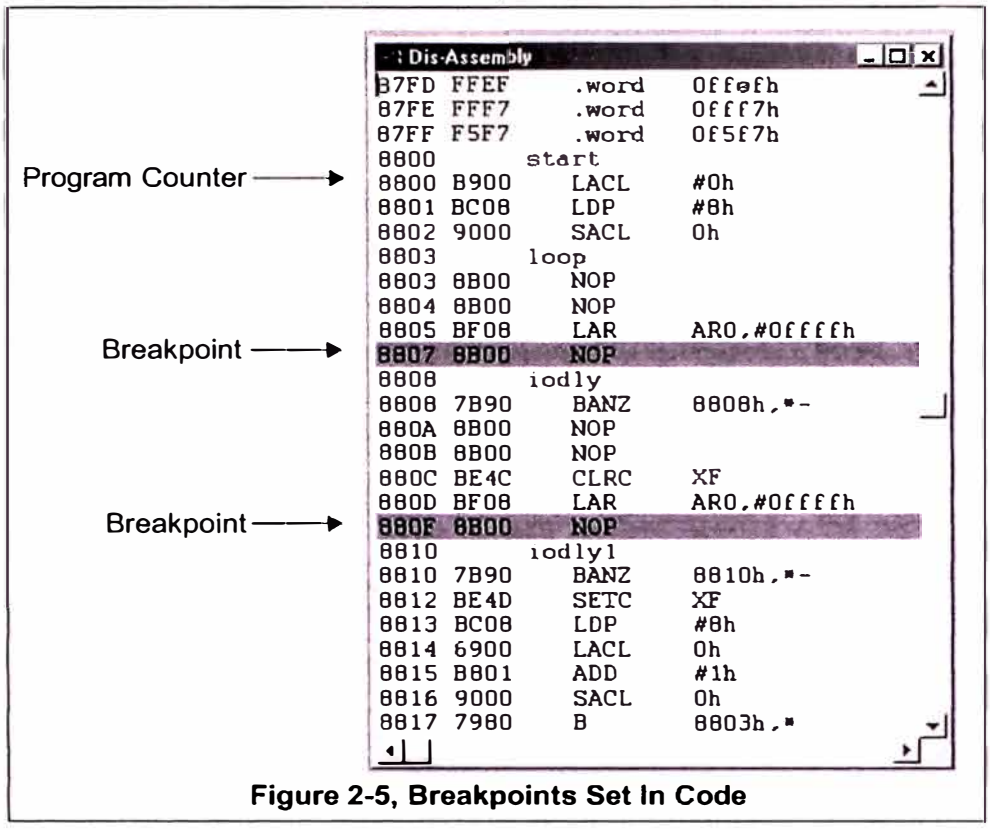


Figure 2-5, Breakpoints Set In Code

2.3.1 Running In Animation Mode

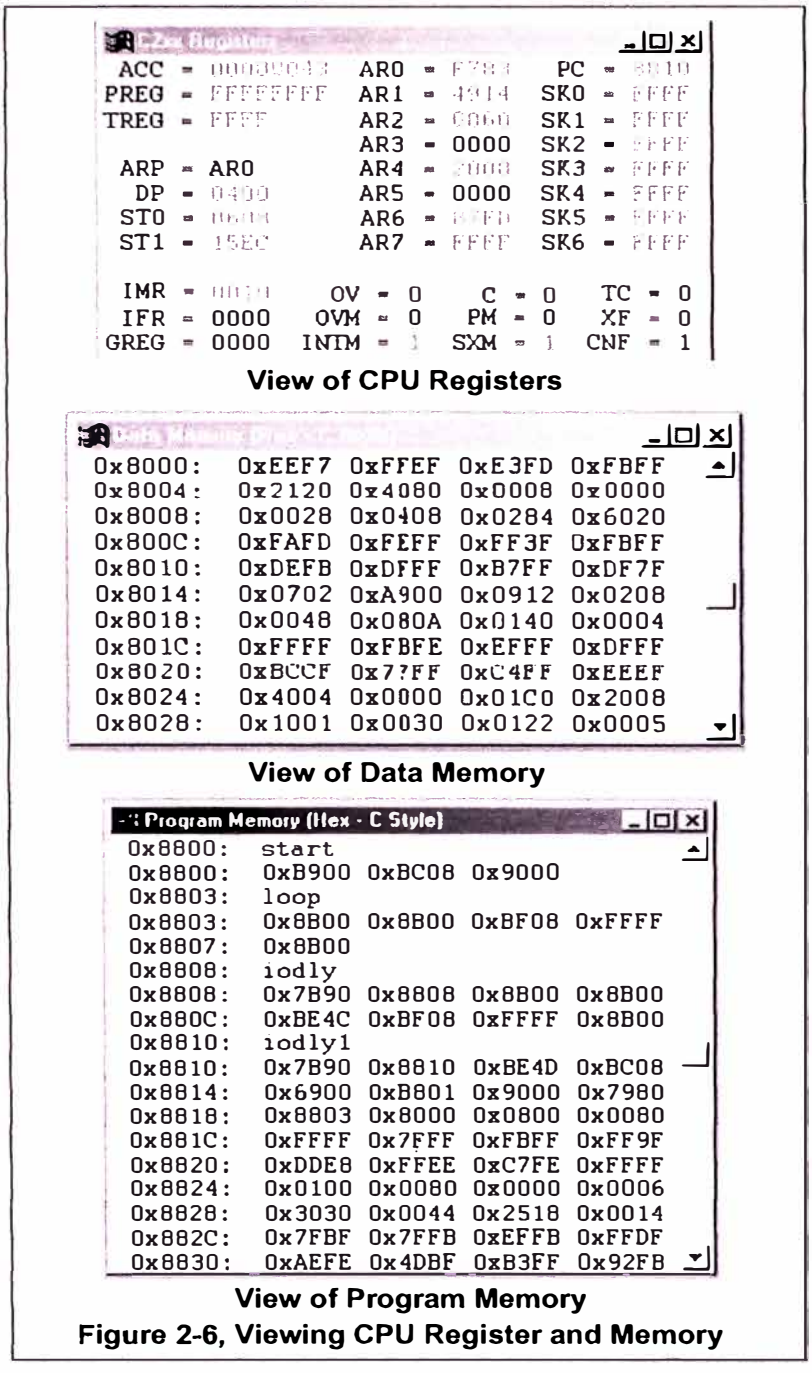
After breakpoints have been set the user can run the program in animation mode which basically runs from breakpoint to breakpoint. To run the animation click on the "Animation" button on the top of the screen. This is the right most selection.

The pause time at each breakpoint during animation can be selected in the "Options" area.

To run from breakpount to breakpoint manually the user would just click the "RUN" button at the top of the screen.

2.4 Viewing CPU Registers and Memory

To inspect the CPU registers, Data memory, program memory, or the I/O space move the cursor over "View" and when the pull down appears select an item. This section shows what some of these selections result in.



View of CPU Registers

View of Data Memory

View of Program Memory

Figure 2-6, Viewing CPU Register and Memory

Appendix A
F24X DSK Troubleshooting

This appendix contains the possible solutions for problems that you may encounter when setting up the DSK

A.1 Hardware Problems

This section addresses some of the possible hardware problems that can be encountered when installing the DSK

Problem: Green LED on DSK does not come on.

Possible solutions: Plug power supply into DSK
Plug DSK power supply into outlet
Turn on power to outlet
Replace DSK

Problem: Debugger on PC cannot communicate with DSK.

Possible solutions: Plug power supply into DSK
Plug DSK power supply into outlet
Turn on power to outlet
Plug serial cable into DSK
Plug serial cable into PC/laptop
Select correct serial port and address
Replace serial cable
Replace DSK

A.2 Software Problems

This section addresses some of the possible software problems that can be encountered when installing the DSK

Problem: Cannot find Code Explorer debugger to run

Possible solutions: Install Code Explorer from diskettes
Execute "*Ce_c2xxw.exe*" from
"*c:\specdig\dsk24x\codeexplorer24x*"

Problem: Cannot find sample program to run

Possible solutions: Install Code Explorer and SD24XASM from diskettes
Find "simple.dsk" in"*c:\specdig\dsk24x\sdasm24x*"

SPECTRUM DIGITAL
INCORPORATED

F240 DSK

Technical Reference

1999

DSP Development Systems

F240 DSK
Technical Reference

503485-0001 Rev. A
June 1999

SPECTRUM DIGITAL, INC.
10853 Rockley Road Houston, TX. 77099
Tel: 281/561-6952 Fax: 281/561-6037
sales@spectrumdigital.com www.spectrumdigital.com

About This Manual

This document describes the board level operations of the F240 DSK which is based on the Texas Instruments TMS320F240 Digital Signal Processor.

The F240 DSK is a stand alone module that allows engineers and software developers to evaluate certain characteristics of the TMS320F240 DSP to determine if the processor meets the designers application requirements. Evaluators can create software to execute onboard or expand the system in a variety of ways.

Notational Conventions

This document uses the following conventions.

The "TMS320F240 DSP Starter Kit" or "F240 DSP Starter Kit" or "F240 DSK" will sometimes be referred to as the DSK.

Program listings, program examples, and interactive displays are shown is a special italic typeface. Here is a sample program listing.

```
equations
!rd = !strobe&rw;
```

Information About Cautions

This book may contain cautions.
This is an example of a caution statement.
A caution statement describes a situation that could potentially damage your software, or hardware, or other equipment. The information in a caution is provided for your protection. Please read each caution carefully.

Related Documents

Texas Instruments TMS320F240 Users Guide
Texas Instruments TMS320C2XX Fixed Point Assembly Language Users Guide
Texas Instruments TMS320C2XX Fixed Point C Language Users Guide
Texas Instruments TMS320C2XX Fixed Point C Source Debugger Users Guide

Chapter 1

Introduction to the F240 DSK

This chapter provides you with a description of the Digital Signal Processor Starter Kit (DSK) for the TMS320F240 (F240 DSK) along with the key features and a block diagram of the circuit board.

Topic **Page**

1.0 Overview of the F240 DSK

The F240 DSK is a stand-alone card that lets evaluators examine certain characteristics of the TMS320F240 digital signal processor(DSP) to determine if this DSP meets their application requirements. Furthermore, the module is an excellent platform to develop and run software for the TMS320F240 processor.

The F240 DSK is shipped with a TMS320F240. The F240 DSK allows full speed verification of F240 code. With 32K words of on board program/data RAM the DSK can solve a variety of problems as shipped. Three expansion connectors are provided for any necessary evaluation circuitry not provided on the as shipped configuration.

To simplify code development and shorten debugging time a symbolic assembler and Windowed debugger are provided. In addition an on board JTAG connector provides interface to emulators which operate with other debuggers that provide assembly language and 'C' high level language debug.

1.1 Key Features of the F240 DSK

The F240 DSK has the following features:

- TMS320F240 Digital Signal Processor
- RS-232 Communications interface to host PC for debug and communications
- 32K words on board program/data RAM
- 16K words on chip Flash memory
- On board 10 Mhz crystal
- 3 Expansion Connectors (analog, I/O, expansion)
- On board IEEE 1149.1 JTAG Connection for Optional Emulation
- 5 volt only operation with supplied AC adapter
- 9 pin serial cable included
- GO DSP Code Explorer Debugger included
- Symbolic assembler included
- Compatible with GO DSP Code Composer
- Compatible with TI 'C' Compiler/assembler/linker

1.2 Functional Overview of the F240 DSK

Figure 1-1 shows a block diagram of the basic configuration for the F240 DSK. The major interfaces of the DSK include the external program and data ram, JTAG interface, UART, and expansion interface.

The DSK interfaces to 32K Words of on board static memory. This memory is divided between the program and data space An external I/O interface supports 65,000 parallel I/O ports.

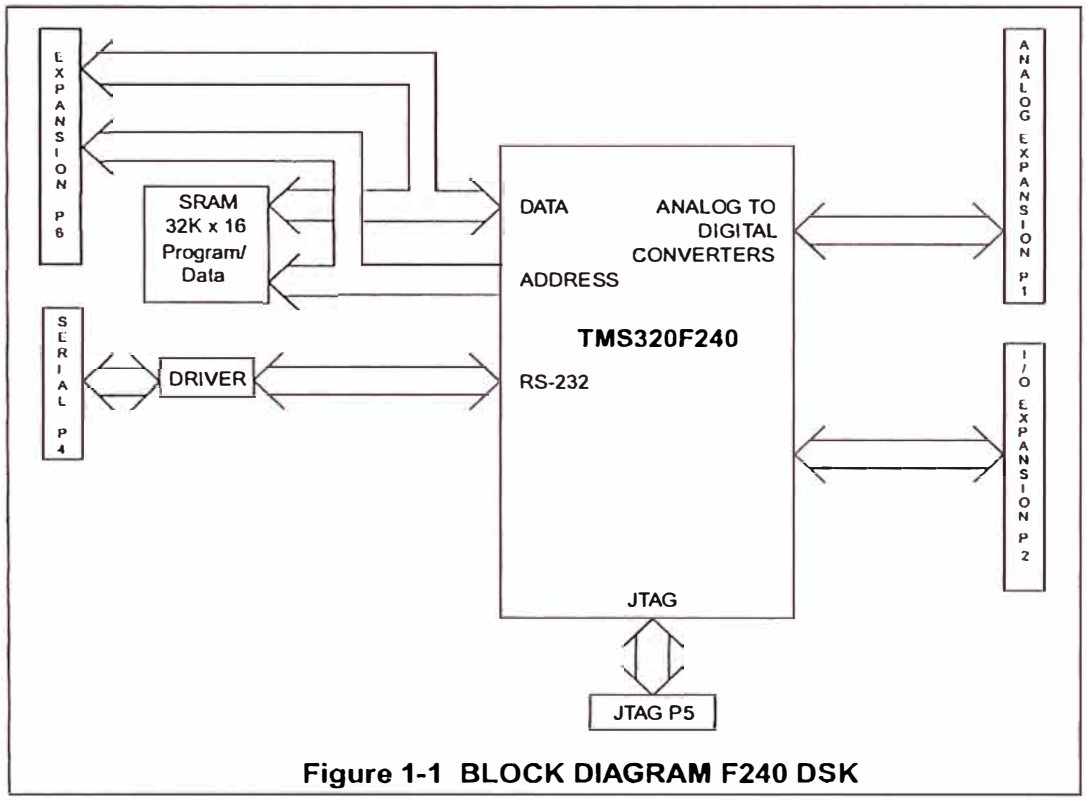


Figure 1-1 BLOCK DIAGRAM F240 DSK

Chapter 2

Operation of the F240 DSK

This chapter describes the operation of the F240 DSK along with the key interfaces and an outline of the circuit board.

Topic

2.0 The F240 DSK Operation

This chapter describes the F240 DSK, its key components, and how they operate. It also provides information on the DSK's various interfaces. The F240 DSK consists of six major blocks of logic.

- External program and data memory
- Analog Interface
- I/O Interface
- RS-232 interface
- Expansion interface
- JTAG Interface

2.1 The F240 DSK Board

The F240 DSK is a 4.0 x 2.5 inch multi-layered printed circuit board which is powered by an external 5 Volt only power supply. Figure 2-1 shows the layout of the F240 DSK

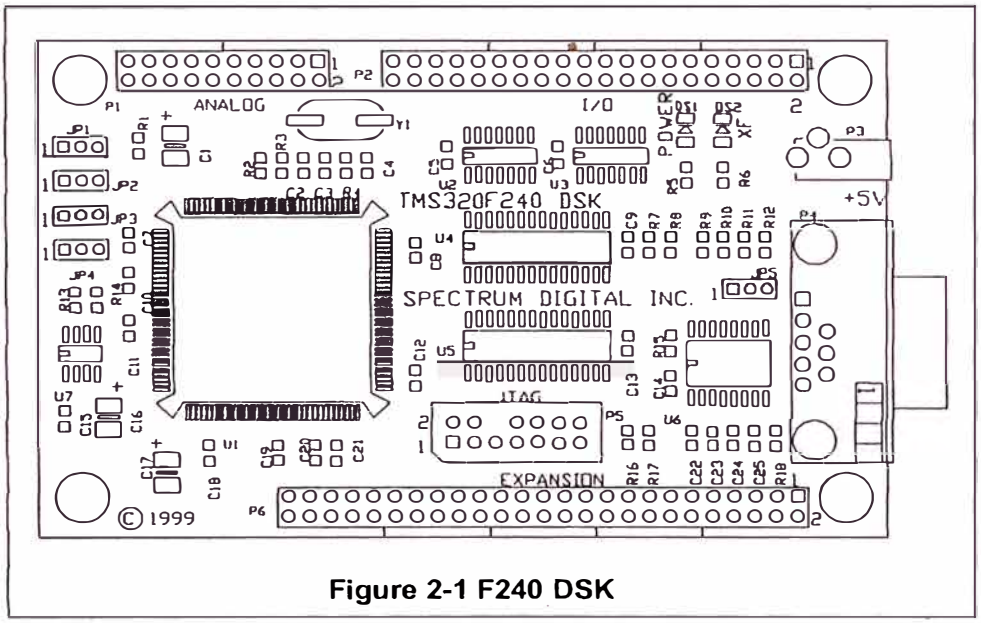


Figure 2-1 F240 DSK

2.1.1 Power Connector

The F240 DSK is powered by a 5 Volt only power supply which is included with the unit. The power supply has a current rating of 1 amp. The unit requires 150 milliamps. The power is supplied via connector P3. If expansion boards are connected to the DSK a higher amperage power supply may be necessary. Section 2.4.2 provides more information on connector P3.

2.2 F240 DSK Memory Interface

The DSK includes the following memory: 16K words on Chip Flash memory, 32K words on board memory split between program and data space. In the as shipped configuration the on chip Flash memory contains the debug target agent that interfaces to the host personal computer. The processor on the DSK is configured for the microcomputer mode.

The DSK is designed such that the user can develop software and load it into on board RAM for debug.

2.2.1 Program Memory

The DSK uses program memory in two different areas; the on chip flash, and the on board RAM. The on chip Flash memory contains the interrupt vectors and target debug agent.

This target debug agent is a small program monitor that interfaces to the debugger program on the PC through the RS-232 interface. The target agent resides low in the on chip Flash and is executed upon reset or power up. The on board RAM resides off chip and is used by application programs.

The 16K words of off chip RAM in program space (0x8000-0xBFFF) has images duplicated at addresses starting at 0x4000 and 0xC000.

The figure below shows the program memory configuration on the F240 DSK.

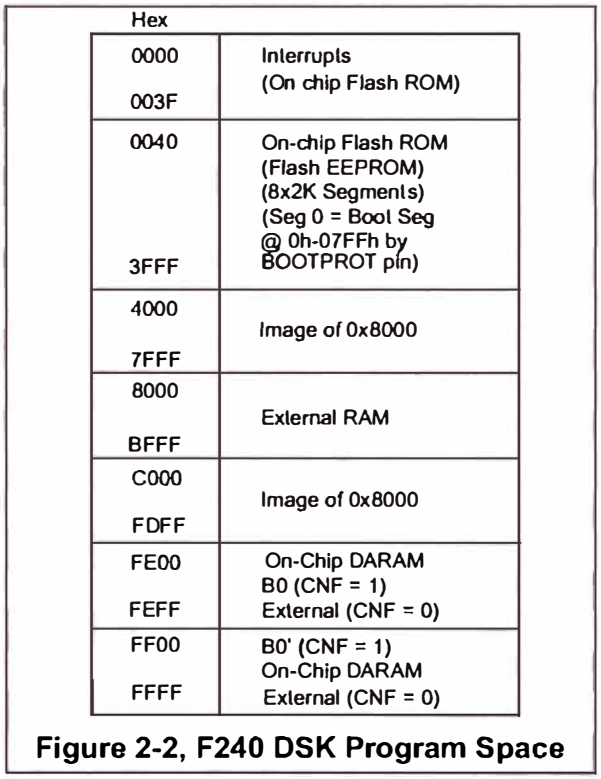


Figure 2-2, F240 DSK Program Space

2.2.1.1 Interrupts

Because the interrupt vectors are in Flash EPROM the user does not have access to them. In addition the target debug agent uses the power up vector and the on chip UART interrupt vector. All other vectors are redirected to a base address of 0x8000 in program space plus the vector offset. The debug agent does this remapping by placing a "branch" instruction to 0x8000 + the offset for each interrupt. The agent also places a "return" instruction at each user vector location on power up to help prevent false/unwanted interrupts from hampering debug sessions.

Furthermore, the debug agent must leave interrupts enabled all the time to allow for the serial communications interface to communicate with the host PC. As a result the user should use a trap for enabling and disabling interrupts since only breakpoints can be used to debug "around" this code.

This allows the user's program to make use of the interrupts on the F240 DSK.

CAUTION

The user should **not** disable interrupts for extended periods of time. When interrupts are disabled the user cannot cause a break from the debugger on the PC..

2.2.2 Data Memory

The data memory configuration on the F240 DSK is shown in the figure below. The on chip data memory is partially used by the debug target agent for its dynamic variables. The remainder of the on chip data memory and the off chip data RAM can be used by user applications. The location of the on chip memory mapped peripheral registers are also shown because these reside in the data space. An image of data space memory 0x8000-0xBFFF is duplicated starting at data space address 0xC000. Off board memory 0xC000-0xFFFF is accessible with a GREG value of 0x00C0,.

Hex	
0000 005F	Memory-Mapped Register and Reserved
0060 007F	On-Chip DARAM B2
0080 00FF	Reserved
0100 01FF	On-Chip DARAM B0 (CNF = 0) Reserved (CNF = 1)
0200 02FF	On-Chip DARAM B0' (CNF = 0) Reserved (CNF = 1)
0300 03FF	On-Chip DARAM B1
0400 04FF	On-Chip DARAM B1'
0500 07FF	Reserved
0800 6FFF	Illegal
7000 73FF	Peripheral Memory-Mapped Registers (System, ADC, SCI, SPI, I/O, Interrupts)
7400 743F	Peripheral Memory-Mapped Registers (Event Manager)
7440 77FF	Reserved
7800 7FFF	Illegal
8000 BFFF	External RAM
C000 FFFF	Image of 0x8000 or Off Board (GREG=0x00C0)

Figure 2-3, F240 DSK Data Space

2.2.3 I/O Space

The entire I/O map for the F240 DSK is available to the user for development.

2.3 Onchip UART

The F240 DSK has an on chip UART. This UART is used by the target debug agent in to communicate to the debugger on the host PC. Therefore the user application should not program any of the registers that are used by the UART.

2.4 F240 DSK Connectors

The F240 DSK has six connectors. Pin 1 of each connector is identified by a square solder pad. The function of each connector is shown in the table below:

Table 1: F240 DSK Connectors

Connector	Function
P1	Analog Interface
P2	I/O Interface
P3	Power Connector
P4	RS-232 Interface
P5	JTAG Interface
P6	Expansion Connector

The diagram below shows the position of each connector.

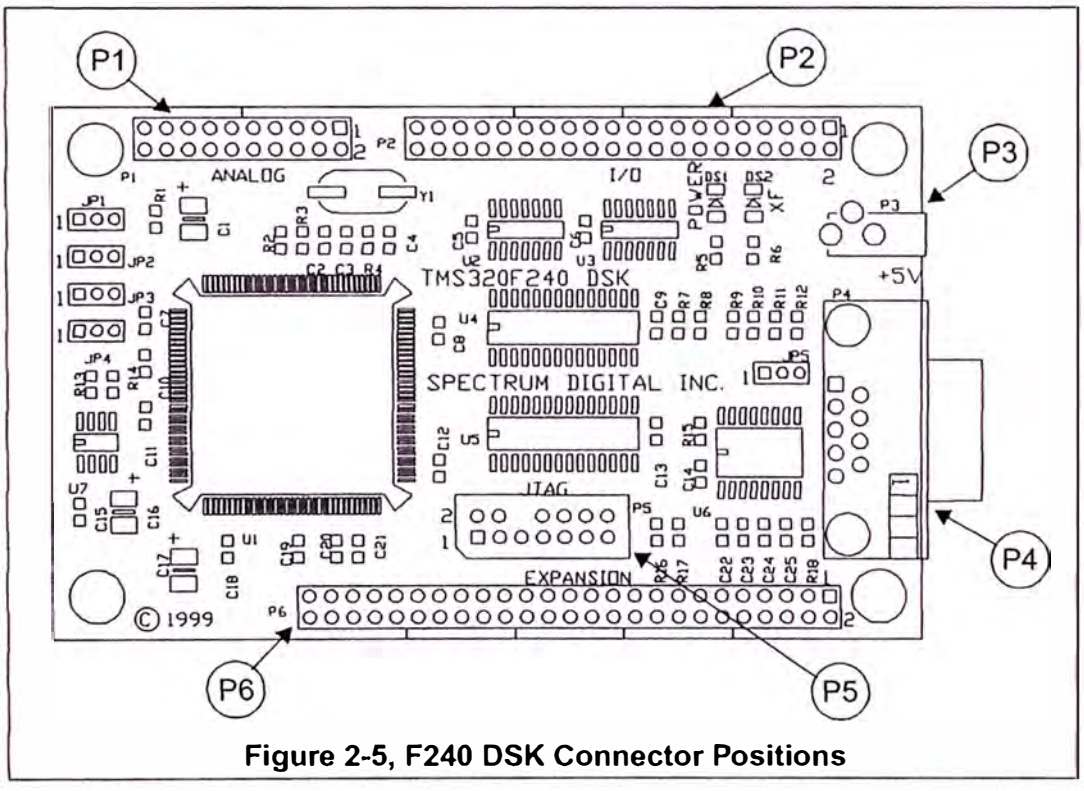


Figure 2-5, F240 DSK Connector Positions

2.4.1 P1, Analog Interface

The position of the 20 pins on the P1 connector is shown in the diagram below as viewed from the top of the DSK.

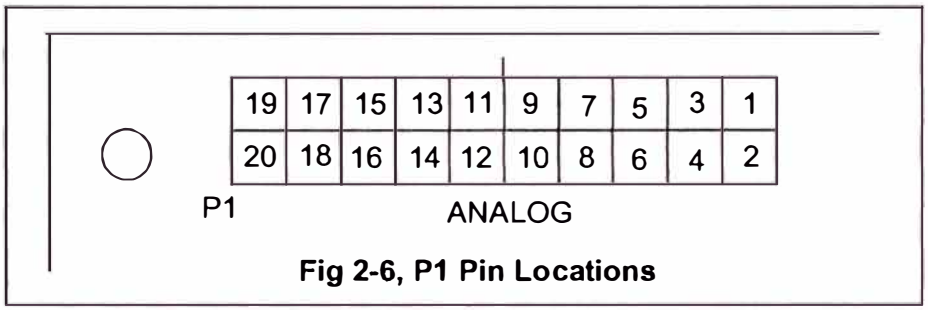


Fig 2-6, P1 Pin Locations

The definition of P1, which has the analog signals is shown below.

Table 2: P1, Analog Interface

Pin #	Signal	Pin #	Signal
1	GND	2	ADC2
3	GND	4	ADC3
5	GND	6	ADC4
7	GND	8	ADC5
9	GND	10	ADC10
11	GND	12	ADC11
13	GND	14	ADC12
15	GND	16	ADC13
17	GND	18	VREFLO
19	GND	20	VREFHI

2.4.2 P2, I/O Interface

The position of the 40 pins on the P1 connector is shown in the diagram below as viewed from the top of the DSK.

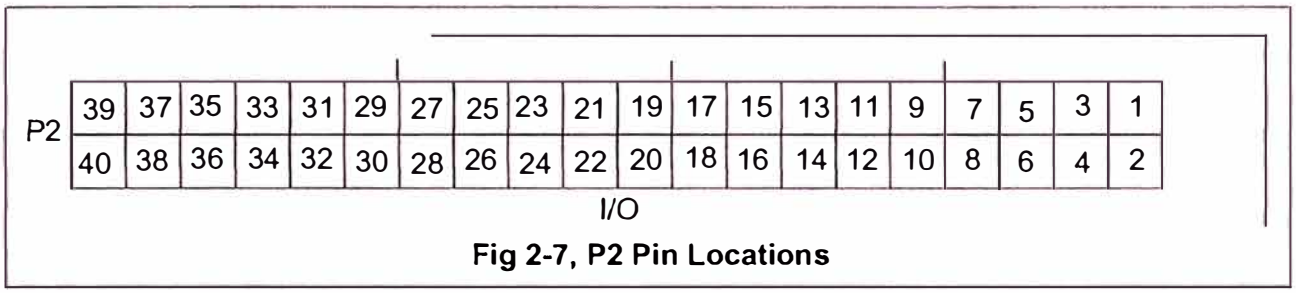


Fig 2-7, P2 Pin Locations

The definition of P2, which has the analog signals is shown below.

Table 3: P2, I/O Interface Connector

Pin #	Signal	Pin #	Signal
1	Vcc	2	Vcc
3	SCITXD/IO	4	SCIRXD/IO
5	XINT-	6	CAP1/QEP1/IOPC4
7	CAP2/QEP2/IOPC5	8	CAP3/IOPC6
9	PWM1/CMP1	10	PWM2/CMP2
11	PWM3/CMP3	12	PWM4/CMP4
13	PWM5/CMP5	14	PWM6/CMP6
15	T1PWM/T1CMP/IOPB3	16	T2PWM/T2CMP/IOPB4
17	TMRDIR/IOPB6	18	TMRCLK/IOPB7
19	GND	20	GND
21	XF/IOPC2	22	BIO-/IOPC3
23	SPISIMO/IO	24	SPISOMI/IO
25	SPICLK/IO	26	SPISTE/IO
27	PWM8/CMP8/IOPB1	28	PWM9/CMP9/IOPB2
29	CLKOUT/IOPC1	30	XINT2-/IO
31	CAP4/IOPC7	32	PWM7/CMP7/IOPB0
33	ADCIN0/IOPA0	34	ADCIN1/IOPA1
35	ADCIN9/IOPA2	36	ADCIN8/IOPA3
37	PDPINT-	38	T3PWM/T3CMP/IOPB5
39	GND	40	GND

2.4.3 P3, Power Connector

Power (5 volts) is brought onto the F240 DSK via the P3 connector. The connector has an outside diameter of xxx and an inside diameter of yyy. The position of the P3 connector is shown below.

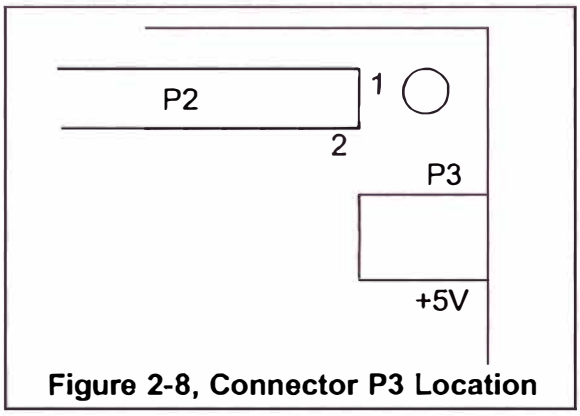


Figure 2-8, Connector P3 Location

The diagram of P3, which has the input power is shown below.

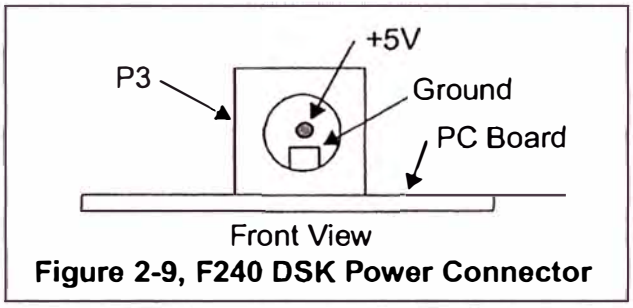


Figure 2-9, F240 DSK Power Connector

2.4.4 P4, RS-232 Interface

The F240 DSK has an on chip UART. The UART signals are brought out to the female DB9 connector P4. The position of the P4 connector is shown below.

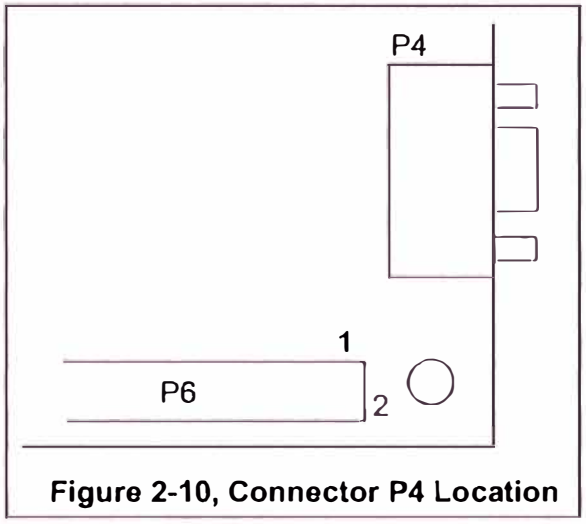


Figure 2-10, Connector P4 Location

Connector P4 is a DB9 female connector. The pin positions for the P4 connector as viewed from the edge of the F240 DSK are shown below.

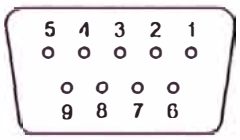


Figure 2-11, P4, DB9 Female Connector

The pin numbers and their corresponding signals are shown in the table below:

Table 4: P4, RS-232 Pinout

Pin #	F240 DSK
1	Reserved
2	Tx, output
3	Rx, input
4	Reset, input
5	GND
6	Reserved
7	Reserved
8	Reserved
9	reserved

2.4.5 P5, JTAG Interface

The F240 DSK is supplied with a 14 pin header interface, P5. This is the standard interface used by JTAG emulators to interface to Texas Instruments DSPs.

The position of the 14 pins on the P5 connector is shown in the diagram below as viewed from the top of the DSK.

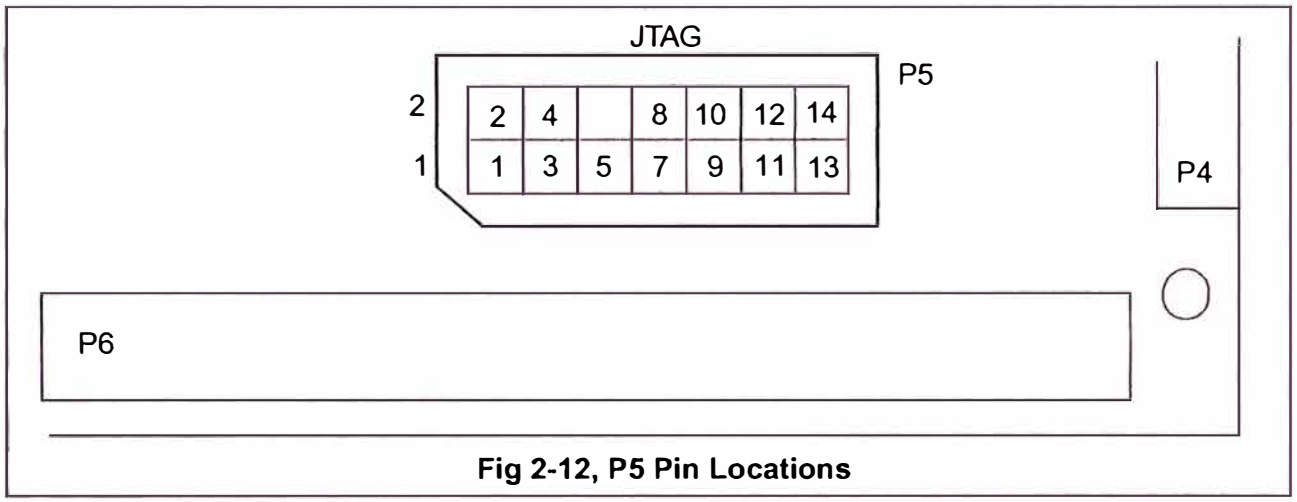


Fig 2-12, P5 Pin Locations

The definition of P5, which has the JTAG signals is shown below.

Table 5: P5, JTAG Interface

Pin #	Signal	Pin #	Signal
1	TMS	2	TRST-
3	TDI	4	GND
5	PD (+5V)	6	**no pin**
7	TDO	8	GND
9	TCK-RET	10	GND
11	TCK	12	GND
13	EMU0	14	EMU1

2.4.6 P6, Expansion Connector

The position of the 50 pins on the P6 connector is shown in the diagram below as viewed from the top of the DSK.

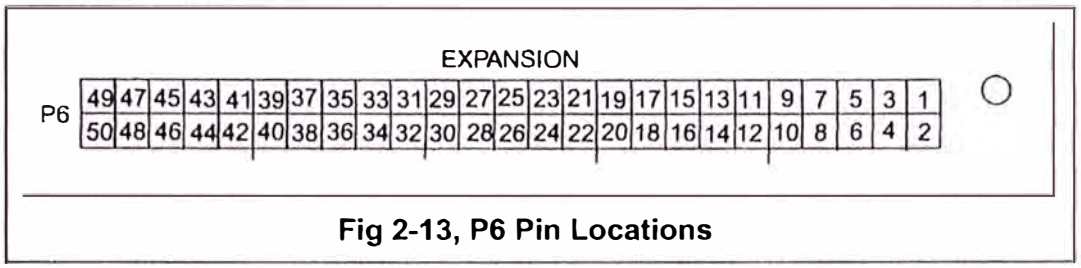


Fig 2-13, P6 Pin Locations

The definition of P6, which has the Host Port Interface signals is shown below.

Table 6: P6, Expansion Interface

Pin #	Signal	Pin #	Signal
1	Vcc	2	Vcc
3	D0	4	D1
5	D2	6	D3
7	D4	8	D5
9	D6	10	D7
11	D8	12	D9
13	D10	14	D11
15	D12	16	D13
17	D14	18	D15
19	A0	20	A1
21	A2	22	A3
23	A4	24	A5
25	A6	26	A7
27	A8	28	A9
29	A10	30	A11
31	A12	32	A13
33	A14	34	A15
35	GND	36	GND
37	PS-	38	DS-
39	READY	40	IS-
41	R/W-	42	STRB-
43	WE-	44	RD-
45	BR-	46	NMI-
47	RS-*	48	XINT3-/IO
49	GND	50	GND

* Bi-directional, **must** be driven with open collector

2.4.7 Connector Part Numbers

The table below shows the part numbers for connectors which can be used on the F240 DSK. Other part numbers from other manufacturers can be used.

Table 7: F240 DSK Suggested Connector Part Numbers

Connector	Part Number	Mating Part Number
P1	AMP 1-102973-0 (male)	HARWIN M20-9831022 (female)
P2	AMP 2-102973-0 (male)	HARWIN M20-9832022 (female)
P3	(supplied)	(comes with power supply)
P4	(supplied)	DB9 male
P5	(supplied)	(comes on emulator)
P6	AMP 2-102973-5 (male)	HARWIN M20-9832522 (female)

2.5 F240 DSK Jumpers

The F240 DSK has 5 jumpers which determine how features on the F240 DSK are utilized. The table below lists the jumpers and their function. The following sections describe the use of each jumper.

Table 8: F240 DSK Jumpers

Jumper #	Size	Function	Position As Shipped From Factory
JP1	1 x 3	MP/MC Mode	2-3
JP2	1 x 3	Vpp/Watchdog	1-2
JP3	1 x 3	VREFLO Source	1-2
JP4	1 x 3	VREFHI Source	1-2
JP5	1 x 3	UART Signal Source	2-3

Each jumper on the F240 DSK is a 1x3 jumper. Each 1x3 jumper must have the selection 1-2 or 2-3. The #2 pin is the center pin. The #1 pin has a square solder pad and can be seen from the solder side of the printed circuit board. This pin is usually marked with a '1' on the boards silkscreen. A top view of the 1x3 jumper is shown below:

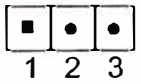


Figure 2-14, 1x3 Jumper Layout

WARNING !
Unless noted otherwise, all 1x3 jumpers must be installed in either the 1-2 or 2-3 position

The diagram below shows the positions of the five jumpers on the F240 DSK.

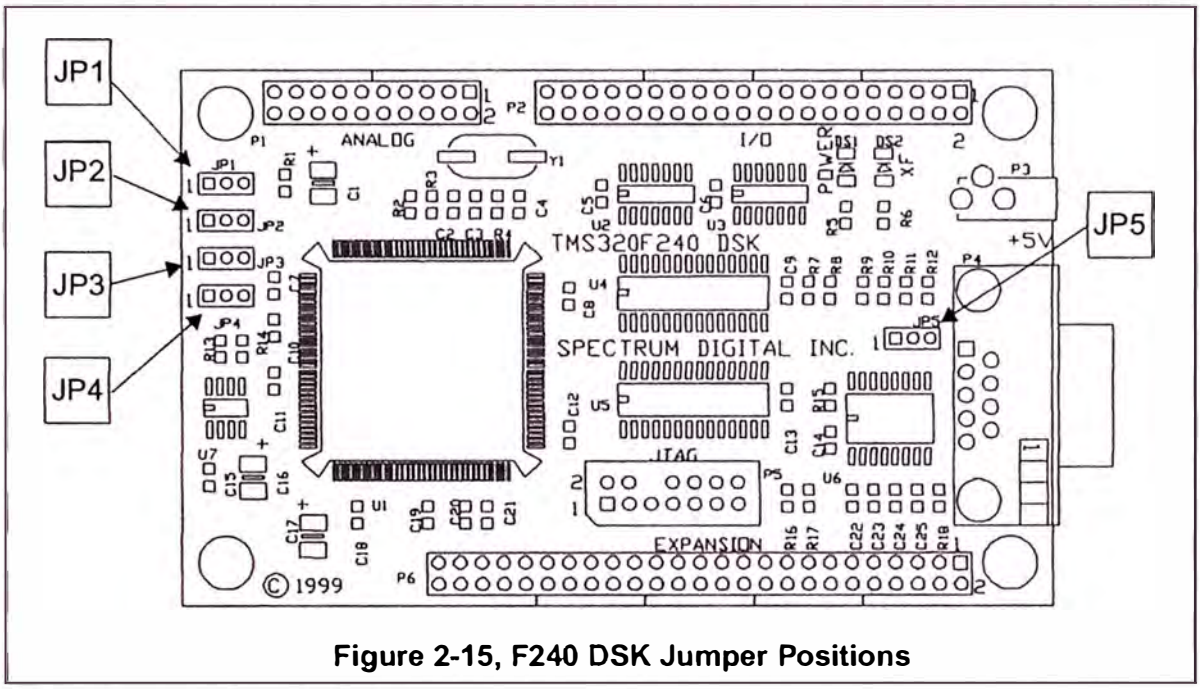


Figure 2-15, F240 DSK Jumper Positions

2.5.1 JP1, MP/MC Mode Select

Jumper JP1 is used to select which mode the TMS320F240 will operate in. If position 1-2 is selected the DSP will operate in the microprocessor mode. The 2-3 selection will operate the DSP in the microcontroller mode.

Table 9: JP1,

Position	Function
1-2	Microprocessor Mode
2-3*	Microcontroller Mode

* default configuration

2.5.2 JP2, Vpp/Watchdog Select

Jumper JP2 is used to select the Flash programming voltage and operation of the on chip watchdog timer. If position 1-2 is selected the Flash programming voltage will be enabled and the watchdog will be turned off. The 2-3 removes Vpp from the DSP and enables the on chip watchdog timer. The on chip debug agent turns off the watchdog in the default configuration

Table 10: JP2,

Position	Function
1-2*	Vpp enabled/Watchdog disabled
2-3	Vpp disabled/Watchdog enabled

* default configuration

2.5.3 JP3, VREFLO Source Select

Jumper JP3 is used to select the source of the VREFLO signal. If position 1-2 is selected the VREFLO signal will come from an on board source and is typically 0 volts. The 2-3 selection will allow the VREFLO source to come from an external source on pin 18 of connector P1 (Analog Interface).

Table 11: JP3,

Position	Function
1-2*	On board VREFLO
2-3	External VREFLO

* default configuration

2.5.4 JP4, VREFHI Source Select

Jumper JP4 is used to select the source of the VREFHI signal. If position 1-2 is selected the VREFHI signal will come from an on board source and is typically 5 volts. The 2-3 selection will allow the VREFHI source to come from an external source on pin 20 of connector P1 (Analog Interface).

Table 12: JP4,

Position	Function
1-2*	Internal VREFHI
2-3	External VREFLO

* default configuration

2.5.5 JP5, UART Signal Source

Jumper JP5 is used to select the source of the SCIRXD signal. This jumper allows the user to determine if this signal on DB9 connector P4 will go to the on chip UART or if the signal on pin 4 of expansion connector P2 (I/O Interface) will go to the UART. If position 1-2 is selected, the signal will come from connector P4. The 2-3 selection will allow the signal to come from pin 4 on connector P2.

Table 13: JP5, UART Signal Source

Position	Function
1-2	Pin 4 on P2
2-3*	P4

* default configuration

2.6 LEDs

The EVM320C54X EVM has two light emitting diodes. DS1 indicates the presence of +5 volts and is normally 'on' when power is applied to the board. DS2 is under software control. It is tied to the XF/IOPC2 pin on the DSP. These are shown in the table below:

Table 14: LEDs

LED #	Color	Controlling Signal	On Signal State
DS1	Green	+5 Volts	1
DS2	Red	XF/IOPC2 on DSP	1

2.7 Resets

The F240 DSK has a power on reset circuit and in the standard configuration will execute on the on chip debug monitor upon power up. The monitor disables the watchdog in normal operation.

2.8 DSK Resource Limitations

Because the the F240 DSK has a resident monitor to communicate with the debugger on the host PC some resources of the DSK are required. These resources should **NOT** be used by user programs. It will hinder the ability to debug software. These resources are shown below.

- Internal F24x data RAM (0x0060-0x007E) is allocated to hold the machine state.
- Interrupt vectors are remapped to program RAM at 0x8000-0x803F.
- The following instructions are loaded by the debug monitor at the above remapped interrupt vectors: “CLRC INTM” and “RTN”.
- Locations 0x8040-0x8042 in program RAM are used for I/O commands.
- Interrupt 5 is used for serial communications.
- The TRAP vector int 17 at location 0x0022 is used for breakpoints and single stepping.
- The RESET vector at location 0x0000 is used to execute the debug monitor on power up.
- The debug monitor runs with interrupts enabled.
- The debug monitor disables the watchdog timer.

ANEXO B

PROGRAMA FUENTE DE COMPRESIÓN DE DATOS

```
/*****************************************************************/
Listado del Programa de Compresión realizado en Lenguaje C
Nombre del Archivo: CODDSP.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Compresión de Datos utilizando Codificación Aritmética en una
              Tarjeta DSP.
/*****************************************************************/
#include <stdlib.h>

/* Valores que definen el número de muestras leídas de archivo */
#define kRegInSize   512
#define kRegOutSize  1

/* Declaraciones usadas para la Codificación y Decodificación Aritmética */
/* Tamaño de los valores del Código Aritmético */
#define Bits_Codigo 16                    /* Número de bits de un valor del código */
typedef long Tipo_Valor_Codigo;           /* Tipo de un valor del código aritmético */
#define Valor_Maximo (((long) 1 << Bits_Codigo) - 1)
                                          /* Valor mas grande del codigo, 2^16 - 1  */

/* Puntos mitad y cuartos en el rango de valores del código */
#define Primer_Cuarto (Valor_Maximo / 4 + 1) /* Punto siguiente al primer cuarto */
#define Mitad (2 * Primer_Cuarto)            /* Punto siguiente a la primera mitad */
#define Tercer_Cuarto (3 * Primer_Cuarto)    /* Punto siguiente al tercer cuarto */

/* Conjunto de símbolos a ser codificados */
#define Numero  Caracteres 256                          /* Número de caracteres */
#define Simbolo  EOF Numero  Caracteres
                                    /* Indice del símbolo EOF para Buffer de Salida */
#define Numero_Simbolos (Numero_Caracteres + 1)  /* Número total de símbolos */

/* Tabla de frecuencias acumuladas */
#define Maxima_Frecuencia (((long) 1 << (Bits_Codigo - 2)) - 1)
                                /* Contador de frecuencia máximo permitido, 2^14 - 1 */
/******* Declaración de Funciones Comunes *******/
void Inicializa_Buffers();
void Inicializar_Modelo();
void Actualizar_Modelo(unsigned char caracter);
void FileIO_In();
void FileIO_OriCfg();
void FileIO_Out();
void FileIO_CodCfg();
/********** Rutinas Propias de CODDSP **********/
void Escribir_Bit(char bit);
void Generar_Bit(unsigned char bit);
void Codificar  Simbolo(unsigned int simbolo);
void Escribir_Bits_Pendientes();
```

```
void Escribir_Ultimo_Byte();

/*  Buffer de Entrada, Salida de Datos */
unsigned char BufferIn[kRegInSize], BufferOut[kRegOutSize], SizeFileOri[2],
              SizeFileCod[2];
unsigned int  Indice_BufferIn = 0, Indice_BufferOut = 0, NumBytesIn = 0,
              NumBytesOut = 0;
unsigned int  NumIn = kRegInSize, NumOut = kRegOutSize, Control;

int frec_acum[Numero_Simbolos + 1];      /* Frecuencias de símbolos acumuladas */

/* Variables que definen los límites del intervalo actual del código */
static Tipo_Valor_Codigo lim_superior = Valor_Maximo;
static Tipo_Valor_Codigo lim_inferior = 0;

/* Contador de bits opuestos a enviar después del siguiente bit */
static Tipo_Valor_Codigo bits_pendientes = 0;

/* Modelo fuente adaptivo */
int frecs[Numero  Simbolos];

/***************** Definición de Funciones **********************/
void Inicializa_Buffers()
{
 int i;
 for ( i = 0; i < NumIn; i = i + 1 ) BufferIn[i] = 0;
 for ( i = 0; i < NumOut; i = i + 1 ) BufferOut[i] = 0;
 for ( i = 0; i < 2; i = i + 1 ) SizeFileOri[i] = 0;
 for ( i = 0; i < 2; i = i + 1 ) SizeFileCod[i] = 0;
 for ( i = 0; i < Numero_Simbolos + 1; i = i + 1 ) frec_acum[i] = 0;
 for ( i = 0; i < Numero_Simbolos; i = i + 1 ) frecs[i] = 0;
}
/* Inicializacion del Modelo */
void Inicializar  Modelo()
{
 int i;
 /* Inicializar a unos la matriz de frecuencias */
 for (i = 0; i < Numero_Simbolos; i++) { frecs[i] = 1; }

 /* Inicializar la matriz de frecuencias acumuladas */
 frec_acum[Numero_Simbolos] = 0;

 for (i = Numero_Simbolos; i > 0; i--) frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];

 /* Comprobar si los contadores estan dentro del límite */
 if (frec_acum[0] > Maxima_Frecuencia) exit(0);
}
```

```
/* Actualizar el modelo cuando se recibe un nuevo símbolo */
void Actualizar_Modelo(unsigned char caracter)
{
 int i;
/* Si los contadores de frecuencia estan en el máximo, se dividen a la mitad, con
cuidado de que no se conviertan en cero, y se recalculan las frecuencias acumuladas
*/
 if (frec_acum[0] == Maxima_Frecuencia)
 {
  for (i = 0; i < Numero_Simbolos; i++) frecs[i] = (frecs[i] + 1) / 2;

  frec_acum[Numero_Simbolos] = 0;
  for (i = Numero_Simbolos; i > 0; i--)
      frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];
 }
 /* Incrementar el contador de frecuencia del caracter y actualizarlas frecuencias
acumuladas */
 i = caracter;
 frecs[i] += 1;
 while(i >= 0)
 {
  frec_acum[i] += 1;
  i--;
 }
}
/********************* Rutinas Propias de CODDSP ************************/
void Escribir_Bit(char bit)
{
 static unsigned char posicion = 1;
 static unsigned char buffer = 0;
 if (bit >= 0)
 {
  if (bit) buffer |= (1<<(8 - posicion));
  posicion++;
  if (posicion > 8)
  {
   BufferOut[Indice_BufferOut] = buffer;
   Indice_BufferOut = Indice_BufferOut + 1;
   /* Escritura de Bloque de Datos al Archivo Comprimido implementado con
   FileIO en Tarjeta DSP */
   FileIO_Out();
   Indice_BufferOut = 0;
   NumBytesOut = NumBytesOut + NumOut;
   posicion = 1;
   buffer = 0;
  }
 }
```

```
 else
 {
  if (posicion != 1)
    {
        BufferOut[Indice_BufferOut] = buffer;
        Indice_BufferOut = Indice_BufferOut + 1;
        /* Escritura de Bloque de Datos al Archivo Comprimido implementado con
        FileIO en Tarjeta DSP */
        FileIO_Out();
        Indice_BufferOut = 0;
        NumBytesOut = NumBytesOut + NumOut;
    }
 }
}
void Generar_Bit(unsigned char bit)
{
 /* Sacar el bit */
 Escribir_Bit(bit);

 /* Sacar tantos bits contrarios como bits pendientes */
 while(bits_pendientes > 0)
 {
  Escribir  Bit(!bit);
  bits_pendientes--;
 }
}
void Codificar_Simbolo(unsigned int simbolo)
{
 Tipo_Valor_Codigo longitud_intervalo;
 unsigned char codificado = 0;

 /* Longitud de la región de código actual */
 longitud_intervalo = lim_superior - lim_inferior + 1;

 /* Reescalar la región de código según lo que corresponda a éste símbolo */
 lim_superior = lim_inferior +
                (longitud_intervalo * frec_acum[simbolo])/frec_acum[0] - 1;
 lim  inferior = lim  inferior +
                (longitud_intervalo * frec_acum[simbolo + 1])/frec_acum[0];

 /* Bucle para la salida de bits */
 while(!codificado)
 {
  if (lim_superior < Mitad)
  {
   /* Sacar un 0 y expandir la mitad inferior */
   Generar_Bit(0);
```

```
    lim_superior = 2 * lim_superior + 1;
    lim_inferior = 2 * lim_inferior;
   }
   else
   {
    if (lim_inferior >= Mitad)
    {
        /* Sacar un 1 y expandir la mitad superior */
        Generar_Bit(1);
        lim_superior = 2 * (lim_superior - Mitad) + 1;
        lim_inferior = 2 * (lim_inferior - Mitad);
    }
    else
    {
        if (lim_inferior >= Primer_Cuarto && lim_superior < Tercer_Cuarto)
        {
         /* Acumular bits pendientes y expandir la mitad formada por el segundo y
         tercer cuartos */
         bits_pendientes++;
         lim_superior = 2 * (lim_superior - Primer_Cuarto) + 1;
         lim_inferior = 2 * (lim_inferior - Primer_Cuarto);
        }
        else
        {
         codificado = 1;
        }
    }
   }
  }
}
void Escribir_Bits_Pendientes()
{
 bits_pendientes++;

 if (lim_inferior < Primer_Cuarto) Generar_Bit(0);
 else Generar_Bit(1);
}
void Escribir_Ultimo_Byte()
{
 Escribir_Bit(-1);
}
/*************** Rutinas de Archivos ***************/
void FileIO_In()                    /* Lectura de Bloque de Datos de Archivo Fuente */
{
 Control = 1;
}
```

```
void FileIO_Out()          /* Escritura de Bloque de Datos a Archivo Comprimido */
{
 Control = 2;
}
void FileIO_OriCfg()                    /* Lectura de Datos de Configuración */
{
 Control = 0;
}
void FileIO_CodCfg()                    /* Escritura de Datos de Configuración */
{
 Control = 3;
}
/************************ Programa Principal ***************************/
void main( )
{
  unsigned char chcod;
  unsigned int  Cont = 0, Salir = 1, TamFile = 0, TamFileIn = 0, TamFileOut = 0;

  /* Inicialización de variables */
  Inicializa_Buffers();

  /* Lectura de Longitud de Archivo Original */
  FileIO_OriCfg();

  TamFileIn = SizeFileOri[0]*256 + SizeFileOri[1];

  Inicializar  Modelo();

  while(Salir)
    {
     Indice_BufferIn = 0; Cont = 0;

     /* Lectura de Bloque de Datos del Archivo Original */
     FileIO_In();

     while( (Cont < NumIn) && (TamFile < TamFileIn) )
          {
           chcod = BufferIn[Indice_BufferIn];
           Indice_BufferIn = Indice_BufferIn + 1;
           Codificar  Simbolo(chcod);
           Actualizar  Modelo(chcod);
           TamFile = NumBytesIn + Indice_BufferIn;
           Cont = Indice_BufferIn;
          }
     NumBytesIn = NumBytesIn + NumIn;
     if ( NumBytesIn >= TamFileIn ) Salir = 0;
    }
```

```
 Codificar_Simbolo(Simbolo_EOF);
 Escribir_Bits_Pendientes();
 Escribir_Ultimo_Byte();
 TamFileOut = NumBytesOut + Indice_BufferOut;

 SizeFileCod[0] = TamFileOut/256; SizeFileCod[1] = TamFileOut%256;

/* Escritura de Longitud de Archivo Comprimido */
 FileIO_CodCfg();
}
```

ANEXO C

PROGRAMA FUENTE DE DESCOMPRESIÓN DE DATOS

```
/*****************************************************************/
Listado del Programa de Compresión realizado en Lenguaje C
Nombre del Archivo: DECDSP.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Descompresión de Datos utilizando Codificación Aritmética en una
              Tarjeta DSP.
/*****************************************************************/
#include <stdlib.h>

#define kRegInSize
#define kRegOutSize  512

/* Declaraciones usadas para la Codificación y Decodificación Aritmética */
/* Tamaño de los valores del Código Aritmético */
#define Bits_Codigo 16                    /* Número de bits de un valor del código */
typedef long Tipo_Valor_Codigo;           /* Tipo de un valor del código aritmético */
#define Valor_Maximo (((long) 1 << Bits_Codigo) - 1)
                                          /* Valor mas grande del codigo, 2^16 - 1  */

/* Puntos mitad y cuartos en el rango de valores del código */
#define Primer_Cuarto (Valor_Maximo / 4 + 1) /* Punto siguiente al primer cuarto */
#define Mitad (2 * Primer_Cuarto)            /* Punto siguiente a la primera mitad */
#define Tercer_Cuarto (3 * Primer_Cuarto)    /* Punto siguiente al tercer cuarto */

/* Conjunto de símbolos a ser codificados */
#define Numero  Caracteres 256                          /* Número de caracteres */
#define Simbolo  EOF Numero  Caracteres
                                    /* Indice del símbolo EOF para Buffer de Salida */
#define Numero_Simbolos (Numero_Caracteres + 1)  /* Número total de símbolos */

/* Tabla de frecuencias acumuladas */
#define Maxima_Frecuencia (((long) 1 << (Bits_Codigo - 2)) - 1)
                                  /* Contador de frecuencia máximo permitido, 2^14 - 1 */
/******* Declaración de Funciones Comunes *******/
void Inicializa_Buffers();
void Inicializar_Modelo();
void Actualizar_Modelo(unsigned char caracter);
void FileIO_In();
void FileIO_OriCfg();
void FileIO_Out();
void FileIO_CodCfg();
/********** Rutinas Propias de DECDSP **********/
void Leer_Primer_Valor();
unsigned int Decodificar_Simbolo();
unsigned char Leer_Bit();
```

```
/*  Buffer de Entrada, Salida de Datos */
unsigned char BufferIn[kRegInSize], BufferOut[kRegOutSize], SizeFileOri[2],
              SizeFileCod[2];
unsigned int  Indice_BufferIn = 0, Indice_BufferOut = 0, NumBytesIn = 0,
             NumBytesOut = 0;
unsigned int  NumIn = kRegInSize, NumOut = kRegOutSize, Control;

int frec_acum[Numero_Simbolos + 1];      /* Frecuencias de símbolos acumuladas */

/* Variables que definen los límites del intervalo actual del código */
static Tipo_Valor_Codigo lim_superior = Valor_Maximo;
static Tipo_Valor_Codigo lim_inferior = 0;

/* Contador de bits opuestos a enviar después del siguiente bit */
static Tipo_Valor_Codigo bits_pendientes = 0;

/* Valor actual dentro de la región del código */
static Tipo_Valor_Codigo valor_actual = 0;

/* Modelo fuente adaptivo */
int frecs[Numero_Simbolos];
/**************** Definición de Funciones **********************/
void Inicializa  Buffers()
{
 int i;
 for ( i = 0; i < NumIn; i = i + 1 ) BufferIn[i] = 0;
 for ( i = 0; i < NumOut; i = i + 1 ) BufferOut[i] = 0;
 for ( i = 0; i < 2; i = i + 1 ) SizeFileOri[i] = 0;
 for ( i = 0; i < 2; i = i + 1 ) SizeFileCod[i] = 0;
 for ( i = 0; i < Numero_Simbolos + 1; i = i + 1 ) frec_acum[i] = 0;
 for ( i = 0; i < Numero_Simbolos; i = i + 1 ) frecs[i] = 0;
}
/* Inicializacion del Modelo */
void Inicializar_Modelo()
{
 int i;
 /* Inicializar a unos la matriz de frecuencias */
 for (i = 0; i < Numero_Simbolos; i++) { frecs[i] = 1; }

 /* Inicializar la matriz de frecuencias acumuladas */
 frec_acum[Numero_Simbolos] = 0;

 for (i = Numero_Simbolos; i > 0; i--) frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];

 /* Comprobar si los contadores estan dentro del límite */
 if (frec_acum[0] > Maxima_Frecuencia) { exit(0); }
}
```

```
/* Actualizar el modelo cuando se recibe un nuevo símbolo */
void Actualizar_Modelo(unsigned char caracter)
{
 int i;
 /* Si los contadores de frecuencia estan en el máximo, se dividen a la mitad, con
cuidado
 de que no se conviertan en cero, y se recalculan las frecuencias acumuladas */
 if (frec_acum[0] == Maxima_Frecuencia)
 {
  for (i = 0; i < Numero_Simbolos; i++)
      frecs[i] = (frecs[i] + 1) / 2;

  frec_acum[Numero_Simbolos] = 0;
  for (i = Numero_Simbolos; i > 0; i--)
      frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];
 }
 /* Incrementar el contador de frecuencia del caracter y actualizar las frecuencias
acumuladas */
 i = caracter;
 frecs[i] += 1;
 while(i >= 0)
 {
  frec_acum[i] += 1;
  i--;
 }
}
/******************** Rutinas Propias de DECDSP ************************/
unsigned char Leer_Bit()
{
 static unsigned char posicion = 1;
 static unsigned char buffer = 0;
 static unsigned char bits_aleatorios = 0;
 unsigned char ch;
 if (posicion == 1)
   {
    Indice_BufferIn = 0;
    /* Lectura de Bloque de Datos del Archivo Original implementado con FileIO en
    Tarjeta DSP */
    FileIO_In();
    buffer = BufferIn[Indice_BufferIn];
    Indice_BufferIn  = Indice_BufferIn + 1;
    if ( Indice_BufferIn == NumIn )
      {
         if (bits_aleatorios == 0) bits_aleatorios = 1;
      }
   }
```

```
 if (bits_aleatorios)
 {
  /* Devolver bits arbitrarios después del fin de fichero, pero como máximo 16 */
  if (bits_aleatorios > Bits_Codigo) { /*exit(0);*/}
  bits_aleatorios++;
 }

 ch = buffer & (1<<(8 - posicion));
 posicion++;
 if (posicion > 8) posicion = 1;

 if (ch) return (1);
 else return(0);
}
void Leer_Primer_Valor()
{
 unsigned char i;
 for(i = 1; i <= Bits_Codigo; i++)
 { valor_actual = 2 * valor_actual + Leer_Bit(); }
}
unsigned int Decodificar_Simbolo()
{
 Tipo_Valor_Codigo longitud_intervalo;
 Tipo_Valor_Codigo acum;
 unsigned char decodificado = 0;
 unsigned int simbolo = 1;

 /* Longitud de la región de código actual */
 longitud_intervalo = lim_superior - lim_inferior + 1;

 /* Encontrar la frecuencia acumulada que corresponde al valor actual */
 acum = ((valor_actual - lim_inferior + 1) * frec_acum[0] - 1) /
        longitud_intervalo;

 /* Encontrar el símbolo que corresponde a esa frecuencia acumulada */
 while(frec_acum[simbolo] > acum) simbolo++;
 simbolo -= 1;

 /* Reescalar la región de código según lo que corresponda a éste símbolo */
 lim_superior = lim_inferior +
                (longitud_intervalo * frec_acum[simbolo])/frec_acum[0] - 1;
 lim  inferior = lim  inferior +
                (longitud_intervalo * frec_acum[simbolo + 1])/frec_acum[0];

 /* Bucle para la entrada de bits */
 while(!decodificado)
 {
```

```
  if (lim_superior < Mitad)
  {
   /* Expandir la mitad inferior */
   valor_actual = 2 * valor_actual + Leer_Bit();
   lim_superior = 2 * lim_superior + 1;
   lim_inferior = 2 * lim_inferior;
  }
  else
  {
   if (lim_inferior >= Mitad)
   {
       /* Expandir la mitad superior */
       valor_actual = 2 * (valor_actual - Mitad) + Leer_Bit();
       lim_superior = 2 * (lim_superior - Mitad) + 1;
       lim_inferior = 2 * (lim_inferior - Mitad);
   }
   else
   {
       if (lim_inferior >= Primer_Cuarto && lim_superior < Tercer_Cuarto)
       {
        /* Expandir la mitad formada por el segundo y tercer cuartos */
        valor_actual = 2 * (valor_actual - Primer_Cuarto) + Leer_Bit();
        lim_superior = 2 * (lim_superior - Primer_Cuarto) + 1;
        lim_inferior = 2 * (lim_inferior - Primer_Cuarto);
       }
       else
       {
        decodificado = 1;
       }
   }
  }
 }
 return(simbolo);
}
/***************** Rutinas de Archivos ****************/
void FileIO_In()            /* Lectura de Bloque de Datos de Archivo Comprimido */
{
 Control = 1;
}
void FileIO_Out()           /* Escritura de Bloque de Datos a Archivo Original */
{
 Control = 2;
}
void FileIO_OriCfg()                    /* Lectura de Datos de Configuración */
{
 Control = 0;
}
```

```
void FileIO_CodCfg()                    /* Escritura de Datos de Configuración */
{
 Control = 3;
}
/************************ Programa Principal ***************************/
void main( )
{
  unsigned int  chdec = 0;
  unsigned int  Cont = 0, Salir = 1, TamFile = 0, TamFileIn = 0, TamFileOut;

  /* Inicialización de variables */
  Inicializa_Buffers();

  /* Lectura de Longitud de Archivo Original */
  FileIO_OriCfg();

  TamFileIn = SizeFileOri[0]*256 + SizeFileOri[1];

  Inicializar_Modelo();
  Leer_Primer_Valor();

  while(Salir)
    {
     Cont = 0;

     while( (Cont < NumIn) && (TamFile < TamFileIn) )
         {
          chdec = Decodificar_Simbolo();
          BufferOut[Indice_BufferOut] = chdec;
          Indice_BufferOut = Indice_BufferOut + 1;
          if ( Indice_BufferOut >= NumOut )
            {
             /* Escritura de Bloque de Datos al Archivo Comprimido implementado
             con FileIO en Tarjeta DSP */
             FileIO_Out();
             NumBytesOut = NumBytesOut + NumOut;
             Indice_BufferOut = 0;
            }
          TamFile = NumBytesOut + Indice_BufferOut;
          Actualizar_Modelo(chdec);
          Cont = Indice_BufferIn;
         }
     if ( TamFile >= TamFileIn ) Salir = 0;
    }

  TamFileOut = NumBytesOut + Indice_BufferOut;
```

```
  SizeFileCod[0] = TamFileOut/256; SizeFileCod[1] = TamFileOut%256;

  /* Escritura de Bloque de Datos al Archivo Comprimido implementado con FileIO
  en Tarjeta DSP */
  FileIO_Out();
}
```

ANEXO D

PROGRAMAS COMPLEMENTARIOS

```
/*****************************************************************/
Listado del Programa de Codificación realizado en Lenguaje C
Nombre del Archivo: CODDAT.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Programa DOS para la PC que permite Codificar un Archivo en
              Formato Hexadecimal para su posterior lectura en la Tarjeta DSP.
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

FILE *fich_ent, *fich_sal, *fich_cfg;

void Comprobar_Parametros(int num_params)
{
 if (num_params != 4)
   { printf("Sintaxis: comp <fichero_entrada> <fichero_salida>\n"); exit (0); }
}

void Abrir_Ficheros(char *nombre_fich_ent, char *nombre_fich_sal, char
*nombre_fich_cfg)
{
 if ((fich_ent = fopen(nombre_fich_ent, "rb")) == NULL)
   {  printf("Imposible abrir el fichero de entrada. \n"); exit (0); }

 if ((fich_sal = fopen (nombre_fich_sal, "wb")) == NULL)
   { printf("Imposible abrir el fichero de salida. \n"); exit (0); }

 if ((fich_cfg = fopen (nombre_fich_cfg, "wb")) == NULL)
   { printf("Imposible abrir el fichero de salida. \n"); exit (0); }
}

void Cerrar_Ficheros()
{
 if ( fclose(fich_ent) ) printf("Imposible cerrar el fichero de entrada.\n");

 if ( fclose(fich_sal) ) printf("Imposible cerrar el fichero de salida.\n");

 if ( fclose(fich_cfg) ) printf("Imposible cerrar el fichero de salida.\n");
}

void main (int argc, char *argv[])
{
 unsigned char Car;
 int       nMSB, nLSB, Salir = 1;
 long      curpos, length;

 Comprobar_Parametros(argc);
```

```
Abrir_Ficheros(argv[1], argv[2], argv[3]);

// Genera Archivo de Configuración
fprintf(fich_cfg,"1651 1 0 1 0\n");

curpos = ftell(fich_ent);
fseek(fich_ent, 0L, SEEK_END);
length = ftell(fich  ent);
fseek(fich_ent, curpos, SEEK_SET);

nMSB = length/256; nLSB = length % 256;

fprintf(fich_cfg,"0x%.4X\n", nMSB ); fprintf(fich_cfg,"0x%.4X", nLSB );

// Genera Archivo de Salida en Formato Hexadecimal
fprintf(fich_sal,"1651 1 0 1 0");

while( Salir )
  {
   fscanf(fich_ent,"%c",&Car);
   if ( feof(fich_ent) )
    { Salir = 0; break; }
   fprintf(fich_sal,"\n0x%.4X",Car);
  }

Cerrar_Ficheros();
}
```

```
/*****************************************************************/
Listado del Programa de Decodificación realizado en Lenguaje C
Nombre del Archivo: DECDAT.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Programa DOS para la PC que permite Decodificador un Archivo
              generado por la  Tarjeta DSP en Formato Hexadecimal a Formato
              ASCII.
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>


FILE *fich_ent, *fich_sal, *fich_cfg;

void Comprobar_Parametros(int num_params)
{
 if (num_params != 4)
   { printf("Sintaxis: comp <fichero_entrada> <fichero_salida>\n"); exit (0); }
}

void Abrir_Ficheros(char *nombre_fich_ent, char *nombre_fich_sal, char
*nombre_fich_cfg)
{
 if ((fich_ent = fopen(nombre_fich_ent, "rb")) == NULL)
   {  printf("Imposible abrir el fichero de entrada. \n"); exit (0); }

 if ((fich_sal = fopen (nombre_fich_sal, "wb")) == NULL)
   { printf("Imposible abrir el fichero de salida. \n"); exit (0); }

 if ((fich_cfg = fopen (nombre_fich_cfg, "rb")) == NULL)
   { printf("Imposible abrir el fichero de salida. \n"); exit (0); }
}

void Cerrar_Ficheros()
{
 if ( fclose(fich_ent) ) printf("Imposible cerrar el fichero de entrada.\n");

 if ( fclose(fich_sal) ) printf("Imposible cerrar el fichero de salida.\n");

 if ( fclose(fich_cfg) ) printf("Imposible cerrar el fichero de salida.\n");
}

void main (int argc, char *argv[])
{
 unsigned  int Cont = 0, L = 0, Num = 0, cnMSB, cnLSB;
 char        sNum1[20], sNum2[20], sNum3[20], sNum4[20], sNum5[20];
```

```
long        curpos, length;
unsigned char nMSB, nLSB;
char        Path[20];

Comprobar_Parametros(argc);
Abrir_Ficheros(argv[1], argv[2], argv[3]);

// Obtiene Longitud de Archivo de Configuración
fscanf(fich_cfg, "%s%s%s%s%s",
                                    &sNum1,&sNum2,&sNum3,&sNum4,&sNum5);
fscanf(fich_cfg, "%x%x", &cnMSB, &cnLSB);
nMSB = (int)cnMSB; nLSB = (int)cnLSB%256;
L = nMSB*256 + nLSB;

// Forma Archivo Original
fscanf(fich_ent, "%s%s%s%s%s",
                                    &sNum1,&sNum2,&sNum3,&sNum4,&sNum5);
while( Cont < L )
  {
   fscanf(fich_ent, "%x", &Num);
   fprintf(fich_sal,"%c", Num);
   Cont++;
  }

Cerrar_Ficheros();
}
```

```
/*****************************************************************/
Listado del Programa de Compresión realizado en Lenguaje C
Nombre del Archivo: CODARI.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Compresión de Datos utilizando Codificación Aritmética para PC
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

/* Declaraciones usadas para la codificación y decodificación aritmética */
/* Tamaño de los valores del código aritmético */
#define Bits_Codigo 16                    /* Número de bits de un valor del código */
typedef long Tipo_Valor_Codigo;           /* Tipo de un valor del código aritmético */
#define Valor_Maximo (((long) 1 << Bits_Codigo) - 1)
                                          /* Valor mas grande del código, 2^16 - 1  */

/* Puntos mitad y cuartos en el rango de valores del código*/
#define Primer_Cuarto (Valor_Maximo / 4 + 1)  /*Punto siguiente al primer cuarto*/
#define Mitad (2 * Primer_Cuarto)             /* Punto siguiente a la primera mitad */
#define Tercer_Cuarto (3 * Primer_Cuarto)     /* Punto siguiente al tercer cuarto   */

/* Conjunto de símbolos a ser codificados */
#define Numero  Caracteres 256                /* Número de caracteres    */
#define Simbolo  EOF Numero  Caracteres       /* Indice del símbolo EOF  */
#define Numero_Simbolos (Numero_Caracteres + 1)  /* Número total de símbolos */

/* Tabla de frecuencias acumuladas */
#define Maxima_Frecuencia (((long) 1 << (Bits_Codigo - 2)) - 1)
                              /* Contador de frecuencia máximo permitido, 2^14 - 1 */

int frec_acum [Numero_Simbolos + 1];     /* Frecuencias de símbolos acumuladas */

/* Variables que definen los límites del intervalo actual del código */
static Tipo_Valor_Codigo lim_superior = Valor_Maximo;
static Tipo_Valor_Codigo lim_inferior = 0;

/* Contador de bits opuestos a enviar después del siguiente bit */
static Tipo_Valor_Codigo bits_pendientes = 0;

/* Valor actual dentro de la región del código */
static Tipo_Valor_Codigo valor_actual = 0;

/* Modelo Fuente Adaptivo*/
int frecs[Numero_Simbolos];

/* Inicialización del modelo */
void Inicializar_Modelo ()
```

```
{
 int i;
 /* Inicializar a unos la matriz de frecuencias */
 for (i = 0; i < Numero_Simbolos; i++)
     frecs[i] = 1;

 /* Inicializar la matriz de frecuencias acumuladas */
 frec_acum[Numero_Simbolos] = 0;

 for (i = Numero_Simbolos; i > 0; i--)
     frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];

 /* Comprobar si los contadores estan dentro del limite */
 if (frec_acum[0] > Maxima_Frecuencia)
   { printf ("Error en los valores de las frecuencias."); exit (0); }
}

/* Actualizar el modelo cuando se requiere un nuevo símbolo */
void Actualizar_Modelo (unsigned char caracter)
{
 int i;
 /* Si los contadores de frecuencia estan en el máximo, se dividen a la mitad, con
    cuidado de que no se conviertan en cero, y se recalculan las frecuencias
    acumuladas */
 if (frec_acum[0] == Maxima_Frecuencia)
 {
   for (i = 0; i < Numero_Simbolos; i++)
       frecs[i] = (frecs[i] + 1) / 2;

   frec_acum[Numero_Simbolos] = 0;
   for (i = Numero_Simbolos; i > 0; i--)
       frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];
 }

 /* Incrementar el contador de frecuencia del carácter y actualizar las frecuencias
    acumuladas */
 i = caracter; frecs[i] += 1;
 while (i >= 0)
 { frec_acum[i] += 1;   i--; }
}

FILE *fich_ent, *fich_sal;

void Comprobar_Parametros (int num_params)
{
 if (num_params != 3)
  { printf ("Sintaxis: comp <fichero_entrada> <fichero_salida> \n"); exit (0); }
```

```
}

void Abrir_Ficheros (char *nombre_fich_ent, char *nombre_fich_sal)
{
 if ((fich_ent = fopen (nombre_fich_ent, "rb")) == NULL)
 { printf ("Imposible abrir el fichero de entrada. \n"); exit (0); }

 if ((fich_sal = fopen (nombre_fich_sal, "wb")) == NULL)
 { printf ("Imposible abrir el fichero de salida. \n"); exit (0); }
}

void Cerrar_Ficheros ()
{
 if (fclose (fich_ent)) printf ("Imposible cerrar el fichero de entrada. \n");

 if (fclose (fich_sal)) printf ("Imposible cerrar el fichero de salida. \n");
}

void Escribir_Bit (char bit)
{
 static unsigned char posicion = 1;
 static unsigned char buffer = 0;

 if (bit >= 0)
 {
  if (bit) buffer |= (1<<(8 - posicion));
  posicion++;
  if (posicion > 8)
  {
   fprintf (fich_sal, "%c", buffer);
   posicion = 1; buffer = 0;
  }
 }
 else
 {
  if (posicion != 1)
   {
    fprintf (fich_sal, "%c", buffer);
   }
 }
}

void Generar_Bit (unsigned char bit)
{
 /* Sacar el bit */
 Escribir_Bit (bit);
```

```
 /* Sacar tantos bits contrarios como bits pendientes */
 while (bits_pendientes > 0)
 {
  Escribir_Bit (!bit);
  bits_pendientes--;
 }
}

void Codificar_Simbolo (unsigned int simbolo)
{
 Tipo_Valor_Codigo longitud_intervalo;
 unsigned char codificado = 0;

 /* Longitud de la región de código actual */
 longitud_intervalo = lim_superior - lim_inferior + 1;

 /* Reescalar la region de código según lo que corresponda a este símbolo */
 lim_superior = lim_inferior +
                (longitud_intervalo * frec_acum[simbolo])/frec_acum[0] - 1;
 lim  inferior = lim  inferior +
                (longitud_intervalo * frec_acum[simbolo + 1])/frec_acum[0];

 /* Bucle para la salida de bits */
 while (!codificado)
 {
  if (lim_superior < Mitad)
  {
   /* Sacar un 0 y expandir la mitad inferior */
   Generar_Bit (0);
   lim_superior = 2 * lim_superior + 1;
   lim_inferior = 2 * lim_inferior;
  }
  else
  {
   if (lim_inferior >= Mitad)
   {
       /* Sacar un 1 y expandir la mitad superior */
       Generar_Bit (1);
       lim_superior = 2 * (lim_superior - Mitad) + 1;
       lim_inferior = 2 * (lim_inferior - Mitad);
   }
   else
   {
       if (lim_inferior >= Primer_Cuarto && lim_superior < Tercer_Cuarto)
       {
        /* Acumular bits pendientes y expandir la mitad formada por el
           segundo y tercer cuartos */
```

```
            bits_pendientes++;
            lim_superior = 2 * (lim_superior - Primer_Cuarto) + 1;
            lim_inferior = 2 * (lim_inferior - Primer_Cuarto);

          }
          else
          {
           codificado = 1;
          }
      }
    }
  }
}

void Escribir_Bits_Pendientes ()
{
 bits_pendientes++;
 if (lim_inferior < Primer_Cuarto) Generar_Bit (0);
 else Generar  Bit (1);
}

void Escribir_Ultimo_Byte ()
{ Escribir_Bit (-1); }

main (int argc, char *argv[])
{
 unsigned char ch;
 int i;

 Comprobar_Parametros (argc);
 Abrir_Ficheros (argv[1], argv[2]);

 Inicializar_Modelo ();

 while ((fscanf (fich_ent, "%c", &ch)) != EOF)
 {
  Codificar_Simbolo (ch);
  Actualizar_Modelo (ch);
 }
 Codificar_Simbolo (Simbolo_EOF);

 Escribir_Bits_Pendientes ();
 Escribir_Ultimo_Byte ();*/
 Cerrar_Ficheros ();
}
```

```
/*****************************************************************/
Listado del Programa de Descompresión realizado en Lenguaje C
Nombre del Archivo: DECAR1.C
Desarrollado por : PEDRO JAVIER RAMOS MATTA
Descripción : Descompresión de Datos utilizando Codificación Aritmética para PC
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declaraciones usadas para la codificación y decodificación aritmética */
/* Tamaño de los valores del código aritmético */
#define Bits_Codigo 16                    /* Número de bits de un valor del código */
typedef long Tipo_Valor_Codigo;           /* Tipo de un valor del código aritmético */
#define Valor_Maximo (((long) 1 << Bits_Codigo) - 1)
                                          /* Valor mas grande del código, 2^16 - 1  */

/* Puntos mitad y cuartos en el rango de valores del código*/
#define Primer_Cuarto (Valor_Maximo / 4 + 1)  /*Punto siguiente al primer cuarto*/
#define Mitad (2 * Primer_Cuarto)             /* Punto siguiente a la primera mitad */
#define Tercer_Cuarto (3 * Primer_Cuarto)     /* Punto siguiente al tercer cuarto   */

/* Conjunto de símbolos a ser codificados */
#define Numero  Caracteres 256                /* Número de caracteres    */
#define Simbolo  EOF Numero  Caracteres       /* Indice del símbolo EOF  */
#define Numero_Simbolos (Numero_Caracteres + 1)  /* Número total de símbolos */

/* Tabla de frecuencias acumuladas */
#define Maxima_Frecuencia (((long) 1 << (Bits_Codigo - 2)) - 1)
                                  /* Contador de frecuencia máximo permitido, 2^14 - 1 */

int frec_acum [Numero_Simbolos + 1];     /* Frecuencias de símbolos acumuladas */

/* Variables que definen los límites del intervalo actual del código */
static Tipo_Valor_Codigo lim_superior = Valor_Maximo;
static Tipo_Valor_Codigo lim_inferior = 0;

/* Contador de bits opuestos a enviar después del siguiente bit */
static Tipo_Valor_Codigo bits_pendientes = 0;

/* Valor actual dentro de la región del código */
static Tipo_Valor_Codigo valor_actual = 0;

/* Modelo Fuente Adaptivo*/
int frecs[Numero_Simbolos];

/* Inicializar el modelo */
```

```
void Inicializar_Modelo ()
{
 int i;
 /* Inicializar a unos la matriz de frecuencias */
 for (i = 0; i < Numero_Simbolos; i++)
  frecs[i] = 1;

 /* Inicializar la matriz de frecuencias acumuladas */
 frec_acum[Numero_Simbolos] = 0;

 for (i = Numero_Simbolos; i > 0; i--)
  frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];

 /* Comprobar si los contadores están dentro del limite */
 if (frec_acum[0] > Maxima_Frecuencia)
  { printf ("Error en los valores de las frecuencias."); exit (0); }
}

/* Actualizar el modelo cuando se recibe un nuevo símbolo */
void Actualizar_Modelo (unsigned char caracter)
caracter;
{
 int i;
  /* Si los contadores de frecuencia están en el máximo, se dividen a la mitad, con
     cuidado de que no se conviertan en cero, y se recalculan las frecuencias
     acumuladas */
 if (frec_acum[0] == Maxima_Frecuencia)
 {
  for (i = 0; i < Numero_Simbolos; i++)
   frecs[i] = (frecs[i] + 1) / 2;

  frec_acum[Numero_Simbolos] = 0;
  for (i = Numero_Simbolos; i > 0; i--)
      frec_acum[i - 1] = frec_acum[i] + frecs[i - 1];
 }

 /* Incrementar el contador de frecuencia del carácter y actualizar las frecuencias
acumuladas */
 i = caracter; frecs[i] += 1;
 while (i >= 0)
 { frec_acum[i] += 1; i--; }
}

FILE *fich_ent, *fich_sal;

void Comprobar_Parametros (int num_params)
{
```

```
 if (num_params != 3)
  { printf ("Sintaxis: comp <fichero_entrada> <fichero_salida> \n"); exit (0); }
}

void Abrir_Ficheros (char *nombre_fich_ent, char *nombre_fich_sal)
{
 if ((fich_ent = fopen (nombre_fich_ent, "rb")) == NULL)
   { printf ("Imposible abrir el fichero de entrada. \n"); exit (0); }

 if ((fich_sal = fopen (nombre_fich_sal, "wb")) == NULL)
   { printf ("Imposible abrir el fichero de salida. \n"); exit (0); }
}

void Cerrar_Ficheros ()
{
 if (fclose (fich_ent)) printf ("Imposible cerrar el fichero de entrada. \n");

 if (fclose (fich_sal)) printf ("Imposible cerrar el fichero de salida. \n");
}

unsigned char Leer_Bit ()
{
 static unsigned char posicion = 1;
 static unsigned char buffer = 0;
 static unsigned char bits_aleatorios = 0;
 unsigned char ch;

 if (posicion == 1)
   if (fscanf (fich_ent, "%c", &buffer) == EOF)
     if (bits_aleatorios == 0) bits_aleatorios = 1;

 if (bits_aleatorios)
 {
  /* Devolver bits arbitrarios después del fin de fichero, pero como máximo 16 */
  if (bits_aleatorios > Bits_Codigo)
    { printf ("Fichero de entrada erroneo.\n"); exit (0); }
  bits_aleatorios++;
 }

 ch = buffer & (1<<(8 - posicion));

 posicion++;

 if (posicion > 8) posicion = 1;

 if (ch) return (1);
 else return (0);
```

```
}

void Leer_Primer_Valor ()
{
 unsigned char i;
 for (i = 1; i <= Bits_Codigo; i++)
     valor_actual = 2 * valor_actual + Leer_Bit ();
}

unsigned int Decodificar_Simbolo ()
{
 Tipo_Valor_Codigo longitud_intervalo;
 Tipo_Valor_Codigo acum;
 unsigned char decodificado = 0;
 unsigned int simbolo = 1;

 /* Longitud de la región de código actual */
 longitud_intervalo = lim_superior - lim_inferior + 1;

 /* Encontrar la frecuencia acumulada que corresponde al valor actual */
 acum = ((valor_actual - lim_inferior + 1) * frec_acum[0] - 1) /longitud_intervalo;

 /* Encontrar el símbolo que corresponde a esa frecuencia acumulada */
 while (frec_acum[simbolo] > acum) simbolo++;
 simbolo -= 1;

 /* Reescalar la región de código según lo que corresponda a este símbolo */
 lim_superior = lim_inferior +
                  (longitud_intervalo * frec_acum[simbolo])/frec_acum[0] - 1;
 lim  inferior = lim  inferior +
           (longitud_intervalo * frec_acum[simbolo + 1])/frec_acum[0];

 /* Bucle para la entrada de bits */
 while (!decodificado)
 {
  if (lim_superior < Mitad)
  {
   /* Expandir la mitad inferior */
   valor_actual = 2 * valor_actual + Leer_Bit ();
   lim_superior = 2 * lim_superior + 1;
   lim_inferior = 2 * lim_inferior;
  }
  else
  {
   if (lim_inferior >= Mitad)
   {
    /* Expandir la mitad superior */
```

```
        valor_actual = 2 * (valor_actual - Mitad) + Leer_Bit ();
       lim_superior = 2 * (lim_superior - Mitad) + 1;
       lim_inferior = 2 * (lim_inferior - Mitad);
      }
      else
      {
       if (lim_inferior >= Primer_Cuarto && lim_superior < Tercer_Cuarto)
       {
        /* Expandir la mitad formada por el segundo y tercer cuartos */
        valor_actual = 2 * (valor_actual - Primer_Cuarto) + Leer_Bit ();
        lim_superior = 2 * (lim_superior - Primer_Cuarto) + 1;
        lim_inferior = 2 * (lim_inferior - Primer_Cuarto);
       }
       else
       {
        decodificado = 1;
       }
      }
     }
   }
   return (simbolo);
}

main (int argc, char *argv[])
{
  unsigned int ch = 0;
  Comprobar_Parametros (argc);
  Abrir_Ficheros (argv[1], argv[2]);

  Inicializar_Modelo (); Leer_Primer_Valor ();

  while ((ch = Decodificar_Simbolo ()) != Simbolo_EOF)
  {
    fprintf (fich_sal, "%c", ch);
    Actualizar_Modelo (ch);
  }

  Cerrar_Ficheros ();
}
```

BIBLIOGRAFÍA

1. Steven W. Smith, Ph.D. "The Scientist and Engineer's Guide to Digital Signal Processing"
2. Hoffman, R.L., "Data Compression in Digital Systems"
3. Peter Abel, Lenguaje Ensamblador y Programación para PC IBM y Compatibles
4. Introducción al DSP
   http://eca.redeya.com
5. Compresión de Datos en las Comunicaciones
   http://intelec.dif.um.es/~aike/tic/compresion/Compress.htm
6. Compression FAQ:
   http://www.faqs.org/faqs/compression-faq/part1/preamble.html
7. Compression Pointers:
   http://www.internz.com/compressionpointers.htm
8. Página de William Ross:
   http://www.ross.net/compression
9. Archivos Fuente de compresores:
   http://www.dc.ee/Files/Programm.Packing
10. Texas Instruments, "TMS320C54x DSP Applications Guide", Reference Set, Volume 4, Literature Number: SPRU 173,
11. Texas Instruments, "TMS320C54x DSP CPU and Peripherals", Reference Set, Volume 1, Literature Number: SPRU 131D,

12. Texas Instruments, "TMS320C54x DSP Algebraic Instruction Set", Reference Set, Volume 3
13. Texas Instruments, "TMS320C54x Evaluation Module Technical", Reference Literature Number: SPRU135
14. Texas Instruments, "TMS32OC5x DSP Starter Kit".
15. Texas Instruments, "TMS320C54x DSKplus DSP Starter Kit"
16. Texas Instruments, "TMS320C24x DSP Controllers. Peripheral Library and Specific Devices", Reference Set, Vol 2, 1997
17. Data Compression Algorithms http://www.ccs.neu.edu/groups/honors-rogram/freshsem/19951996/jnl22/jeff.htm
18. SPE 2000 Project http://www.cs.princeton.edu/ugradpgm/spe/summer00/jon/
19. Mark Nelson, "Arithmetic Coding" http://www.dogma.net/markn/articles/arith.htm