24

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

FILTROS DIGITALES DE ONDA

T E S I S

QUE PARA OBTENER EL TITULO DE:
INGENIERO EN TELECOMUNICACIONES
P R E S E N T A :
MIGUEL ÁNGEL PALOMERA PÉREZ

DIRECTOR DE TESIS: Dr. BOHUMIL PŠENIČKA

CIUDAD UNIVERSITARIA

OCTUBRE 2002

Agradecimientos

A Dios, por cada amanecer, por cada oportunidad y reto, gracias.

A mi mamá, por confiar en mi, pero por sobretodo por ser mi madre, gracias má, por que si cree que tiene buenos hijos es por que tuvimos una gran mamá.

A mi hermana, por todas las travesuras compartidas, por los consejos, pero sobretodo por estar siempre cuando lo he necesitado, gracias gorda.

A Laura, por todo los abrazos y besos, por "el te quiero", gracias hermana.

A mi papá, por darme la vida, por los buenos ratos pasados juntos, gracias pá donde quiera que te encuentres.

A don Cande, por cuidar de mamá, por el consejo desinteresado y por todo su apoyo, gracias.

A ti amor, por enseñarme que una profesión no es una bata que se pueda quitar y poner, pero sobre todo por cada uno de los momentos compartidos, gracias Cynthia.

A los hermanos huastecos, por el camino compartido juntos, por los partidos de básquet, por las papas, por cinco años de trabajo, gracias Jonh, gracias Hugo.

Al señor roy, por cada locura compartida, por cada consejo a tiempo, gracias Rogelio.

Contenido

Introducción.

A grandes rasgos, se podría decir que un filtro es un sistema que, ante una señal de entrada que posee componentes de diferentes frecuencias, produce como salida una señal con componentes de sólo algunas de las frecuencias existentes en la señal de entrada. Durante el desarrollo del proyecto nos centraremos en los filtros digitales que admiten entradas discretas y producen salidas discretas. Como aplicaciones típicas de filtros digitales se puede citar: eliminación de ruido, conformación de la señal, tratamiento de imágenes, ecualización digital de sonido, etc.

En el caso que nos ocupa veremos el diseño de filtros digitales a través de una estructura de baja sensibilidad al ruido conocida como filtros de onda, la teoría detrás de este tipo de estructura fue desarrollada por Fetweiss[1].

Esta tesis surge como el resultado del análisis llevado a cabo por el Dr. Bohumil Pšenička sobre los filtros digitales de onda, al darse cuenta que las ecuaciones que describen a los distintos filtros poseían cierta periodicidad, lo cual hacia pensar que podría existir una manera de realizarlos de forma genérica, independientemente del orden del filtro y como se vio en el desarrollo de la tesis del tipo, paso bajas o paso altas. Es este el objetivo de la tesis, diseñar un algoritmo capaz de realizar los filtros digitales de onda independientemente de su orden o tipo. Dicho algoritmo deberá ser implementado en el TMS320C30, y además permitirá llevar a cabo la realización de distintos tipos de filtros sin tener que volver a cargar el programa dentro del DSP.

En el capítulo 1 se analiza, a grandes rasgos, la teoría fundamental de los filtros, pasando desde conceptos muy básicos, como es la definición de los filtros, hasta llegar al concepto de filtros digitales. En el capítulo 2 se discute la teoría de los filtros digitales de onda y la forma de cómo por medio de las tablas de aproximaciones ya conocidas, como la de Butterworth, es posible obtener estas estructuras. En el capítulo 3 se obtienen las ecuaciones que describen a los filtros digitales, así como, se plantean los requerimientos que debe tener el

[1] Alfred Fettweiss. "Digital filter structures related clasic filter networks" Arch. Elek. Ubertragung, vol 25, pp 79-89, Feb. 1971

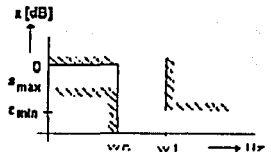
algoritmo que se esta buscando; por último se realiza una pequeña aproximación a este algoritmo en pseudocodigo.

En el capítulo 4, se analizan las características del DSP que nos van a permitir llevar a cabo la realización del algoritmo, posteriormente se muestra paso a paso como este se implementó.

Referencias:

[1] Alfred Fettweiss. "Digital filter structures related clasic filter networks" Arch. Elek. Ubertragung, vol 25, pp 79-89, Feb. 1971

Capítulo 1



INTRODUCCIÓN A LA TEORÍA DE FILTROS.

El presente capítulo pretende ser una pequeña introducción a la teoría de filtros, y como tal sólo aborda los temas que se consideran básicos para la comprensión de los capítulos siguientes.

1.1 ¿Qué es un filtro?

Un filtro es un dispositivo que permitirá el paso de señales que se encuentre dentro de un rango de frecuencias, la señal que no se encuentre dentro de este rango será atenuada, en el caso ideal tendrá un valor de cero. Al rango de frecuencias que deja pasar el filtro se le conoce como **banda de paso o ancho de banda**, a la frecuencia limite del rango se le denomina **frecuencia de corte.** De acuerdo a esta definición es posible tener distintos tipos de filtros dependiendo de la banda de paso que estos posean.

El filtro más sencillo es el **filtro paso bajas ideal**, el cual tiene una ganancia unitaria para las frecuencias comprendidas entre cero y la frecuencia de corte (ω_c), y un valor de cero para cualquier otra frecuencia. En términos de voltajes, podemos definir la función de transferencia de un filtro paso bajas mediante la siguiente relación:

$$\left|\frac{v_2}{v_1}\right| \qquad \text{..... (1.1)}$$

donde:

v_1 = al voltaje en la entrada.
v_2 = al voltaje a la salida.

Dentro de la banda de paso un filtro ideal dejará pasar íntegramente a la señal de entrada, por lo que se tiene que:

$$\left|\frac{v_2}{v_1}\right| = 1 \quad \text{..... (1.2)}$$

Para frecuencias mayores a la frecuencia de corte se tiene que $v_2 = 0$

Entonces la relación de voltajes quedaría como:

$$\left|\frac{v_2}{v_1}\right| = 0 \quad \text{.....(1.3)}$$

si se grafica la relación de voltajes anterior contra los valores de frecuencias se obtiene la gráfica que se muestra en la figura 1.1. Esta gráfica corresponde al comportamiento de un filtro ideal paso bajas.

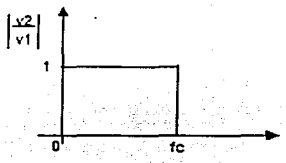


Figura 1.1: Gráfica de la respuesta de un filtro paso bajas ideal.

Continuando con la clasificación de los filtros tenemos: a los **filtros paso altas,** los cuales permiten el paso de señales con frecuencias mayores a la frecuencia de corte; los **filtros paso banda,** los cuales poseen dos frecuencias de corte una inferior (f_{-1}) y otra superior (f_1), lo que da como resultado una frecuencia central y un ancho de banda comprendido entre estas dos frecuencias. En la figura 1.2 se puede observar la respuesta de los filtros mencionados.

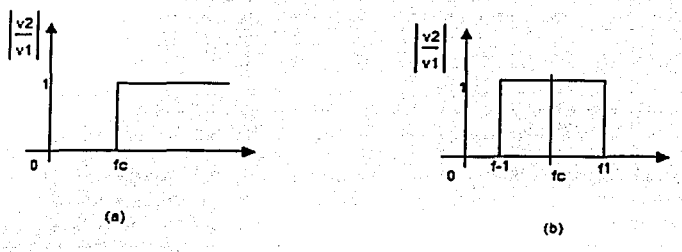


Figura 1.2: Respuesta de (a) un filtro paso altas ideal y (b) de un filtro paso bandas ideal.

Además de estos filtros existen otros como los supresores de banda o los filtros adaptables, los cuales no serán tratados en este documento.

1.2 Especificación de las plantillas de diseño.

A los filtros vistos anteriormente se les conoce como ideales porque su respuesta tal y como se observa en las gráficas no es posible de llevar a cabo. En filtro realizable, no es posible que la señal no se vea afectada en un determinado rango de frecuencias y sea suprimida por completo fuera de este, lo que es posible es que la señal sea mínimamente atenuada dentro de la banda de paso y, que sea, más atenuada (hasta un valor que sea considerado suficiente) fuera de ese intervalo, tampoco es posible que el valor de la atenuación cambie de forma tan abrupta prácticamente en el mismo valor de frecuencia, lo que se puede hacer es que la atenuación aumente paulatinamente dentro de un rango de frecuencias hasta alcanzar el valor deseado.

Las características que se proporcionan normalmente para el diseño de filtros realizables son: la frecuencia a partir de la cual se debe filtrar, que es llamada frecuencia de corte (ω_c), y que generalmente corresponde a una atenuación (a_{max}) de tres decibeles de la magnitud del voltaje de la señal de salida con respecto al voltaje a la entrada del filtro; el otro parámetro es la frecuencia a la cual se obtiene la atenuación mínima requerida (a_{min}). Por lo general estas características se expresan en forma gráfica, a esta gráfica se le conoce como **plantilla de diseño del filtro o simplemente plantilla del filtro.** La plantilla del filtro se representa en un plano cartesiano, el eje de las abscisas representa la frecuencia ω y en el de las ordenadas la atenuación a. La frecuencia generalmente se especifica en hertz, pero para facilitar la manipulación

matemática de la función que representa la respuesta del filtro, se usa también como unidad a los radianes sobre segundo. La atenuación puede expresarse en nepers o en decibeles, durante el presente trabajo se especificará la atenuación en decibeles a menos que se indique lo contrario.

En la plantilla del filtro existen tres regiones de frecuencias: la de la banda de paso, la banda o bandas de supresión y la región de transición. La banda de paso, es el rango de frecuencias de la señal en las que el voltaje de salida es igual o casi igual al de la entrada, es decir, la señal no es atenuada (la atenuación es cero decibeles) o en el peor de los casos es atenuada a_{max} decibeles. La banda o bandas de supresión son aquellos rangos de frecuencias en los que la señal debe ser atenuada al menos a_{min} decibeles. La región de transición es el o los rangos de frecuencia en donde existe un cambio de la atenuación de la señal, entre una banda de paso y una banda de supresión, o viceversa. En la figura 1.3 se pueden apreciar las plantillas para filtros descritos anteriormente.

Por ser el filtro paso bajas el más sencillo de describir matemáticamente, además de que todos los filtros pueden transformarse a filtros paso bajas normalizados[1], es el que se empleará en los temas y capítulos siguientes. Si se desea conocer las ecuaciones de transformación para los distintos tipos de filtros, se remite al lector al libro de Procesamiento de Señales del Dr. Bohumil Pšenička capítulo tres "Transformación de las plantillas".

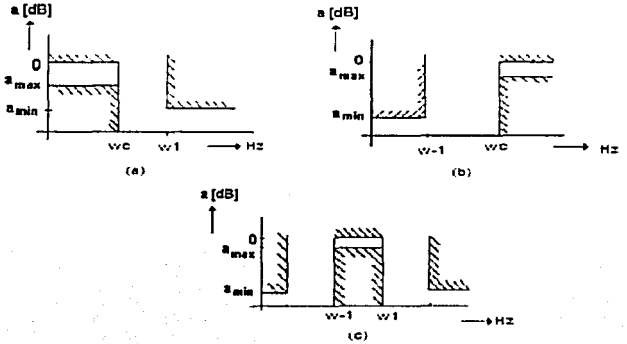


Figura 1.3: Plantillas para el diseño de los filtros (a) paso bajas, (b) paso altas, (c) paso bandas.

[1] Un filtro paso bajas normalizado tiene su frecuencia de corte $\Omega = 1$

1.3 Aproximación a las plantillas de los filtros.

Una vez que se conocen las características del diseño, es necesario resolver el problema de la aproximación, que consiste en encontrar una función para representar el sistema tal que, por un lado aproxime la curva dada con las tolerancias especificadas en la plantilla de diseño y, sea realizable por una red de la forma deseada. Para la aproximación de las plantillas se emplea la ecuación característica[2].

$$G(s)G(-s) = 1 + \phi(s)\phi(-s) \qquad \text{..... (1.4)}$$

donde G(s) es la función de transferencia definida por:

$$G(s) = \sqrt{\frac{U_1 I_1}{U_2 I_2}} \qquad \text{.....(1.5)}$$

La función característica $\phi(s)$ debe ser cualquier función positiva. Por su parte la función de transferencia debe cumplir las siguientes condiciones:

- El numerador de G(s) debe ser un polinomio Hurwitz. Esto significa que los ceros deben estar en el lado izquierdo del plano complejo s.
- Para todas las frecuencias ω la función de transferencia G(s) debe cumplir la condición $|G(j\omega)| \geq 1$.

La prueba de G(s) como una función de transferencia válida no es fácil, pero si $\phi(s)$ es una función positiva podemos ver de (1.4) que se cumple la condición $|G(j\omega)| \geq 1$. Si se eligen los ceros de la ecuación de transferencia en el lado izquierdo del plano complejo de G(s) entonces se cumplen las dos condiciones necesarias[3].

1.3.1 Aproximación de Butterworth[4]

Para la aproximación de Butterworth se elige la función característica:

$$\phi(s) = \varepsilon s^n \qquad \text{.....(1.6)}$$

Por lo que la función de transferencia (1.4) queda de la forma:

$$G(s)G(-s) = 1 + (-1)^n \varepsilon^2 s^{2n} \qquad \text{.....(1.7)}$$

[2] Dr. Bohumil Pšenička, Procesamiento de Señales, pp 65
[3] Dr. Bohumil Pšenička, Procesamiento de Señales, pp 65
[4] Estos mismos cálculos los puede encontrar en libro antes descrito, páginas 66-70

$$|G(\Omega)|^2 = G(j\Omega)G(-j\Omega) = 1 + (-1)^n \epsilon^2 \Omega^{2n} \quad \text{.....(1.8)}$$

Cuando se pide diseñar un filtro, lo primero que se debe determinar, es el orden mínimo n de la función de transferencia que cumpla con las características pedidas. Para calcular el orden del filtro normalizado de la figura 1.4 necesitamos conocer la atenuación máxima en a_{max} en Ω_1 y la atenuación mínima a_{min} en Ω_2.

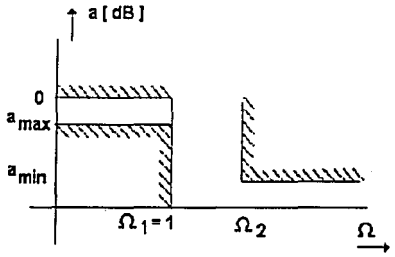


Figura 1.4: Plantilla normalizada del filtro paso bajas.

Para $\Omega=\Omega_1$ se tiene que la atenuación $a = a_{max}$ y de (1.8) obtenemos.

$$e^{2a_{max}} = 1 + \epsilon^2 \qquad 10^{\frac{a_{max}}{10}} = 1 + \epsilon^2$$

Despejando ε

$$\epsilon = \sqrt{e^{2a_{max}} - 1} \qquad \epsilon = \sqrt{10^{\frac{a_{max}}{10}} - 1}$$

Para $\Omega = \Omega_2$ se tiene que la atenuación $a = a_{min}$ y de (1.8) se puede calcular n.

$$e^{2a_{min}} = 1 + \epsilon^2 \Omega_2^{2n} \qquad 10^{\frac{a_{min}}{10}} = 1 + \epsilon^2 \Omega_2^{2n}$$

Despejando mediante logaritmos se obtiene que:

$$n > \frac{ln \frac{e^{2a_{min}} - 1}{e^{2a_{max}} - 1}}{2 \ln \Omega_2} \quad \text{.....(1.9)}$$

En la ecuación (1.9) la atenuación tiene unidades de Nepers. Para sustituir la atenuación en Decibeles se utilizan las siguientes ecuaciones.

$$n > \frac{ln\frac{e^{0.23a_{min}}-1}{e^{0.23a_{max}}-1}}{2\ln\Omega_2} \quad \text{.....(1.10)}$$

$$n > \frac{log\frac{10^{\frac{a_{min}}{10}}-1}{10^{\frac{a_{max}}{10}}-1}}{2\log\Omega_2} \quad \text{.....(1.11)}$$

Para calcular la función de transferencia G(s) se calculan los ceros de la ecuación:

$$G(s)G(-s) = 1 + (-1)^n\epsilon^2 s^{2n} = 0 \quad \text{.....(1.12)}$$

Para n par se tiene:

$$1 + \epsilon^2 s^{2n} = 0$$

Si ε=1, los ceros de la función de transferencia G(s) para el orden n par se calculan mediante la ecuación (1.13) y se ubican en el círculo unitario.

$$s_k = e^{j\frac{\pi+2k\pi}{2n}} = \cos\frac{\pi+2k\pi}{2n} + j\sin\frac{\pi+2k\pi}{2n} \quad \text{.....(1.13)}$$

Ahora bien para n impar se obtiene:

$$1 - \epsilon^2 s^{2n} = 0$$

Si ε = 1, los ceros de la función de transferencia G(s) para el orden n impar se calculan mediante la ecuación (1.14).

$$s_k = e^{j\frac{\pi k}{n}} = \cos\frac{k\pi}{n} + j\sin\frac{k\pi}{n} \quad \text{.....(1.14)}$$

En ambos casos k varia de cero hasta 2n-1.

Ejemplo 1:

Calcular la función de transferencia de un filtro paso bajas normalizado mediante la aproximación de Butterworth para el orden n=3.

Solución:

Como el orden del filtro es impar empleamos la ecuación (1.14). Variando k de 0 a 2, se obtiene:

$$s_0 = \cos 0 + j \sin 0 = 1$$

$$s_1 = \cos \frac{\pi}{3} + j \sin \frac{\pi}{3} = \frac{1}{2} + j\frac{\sqrt{3}}{2}$$

$$s_2 = \cos \frac{2\pi}{3} + j \sin \frac{2\pi}{3} = -\frac{1}{2} + j\frac{\sqrt{3}}{2}$$

Los otros ceros no son necesarios calcularlos, porque son conjugados y están ubicados en la parte derecha del eje imaginario. Los ceros de G(s) y G(-s) se muestran en la figura 1.5. La función de transferencia debe tener los ceros en la parte izquierda del plano s, esto para que sea realizable, por lo tanto se tiene:

$$G(s) = (s - s_2)(s - s_3)(s - s_4)$$

Sustituyendo el valor de los ceros encontrados:

$$G(s) = (s + \frac{1}{2} - j\frac{\sqrt{3}}{2})(s + \frac{1}{2} + j\frac{\sqrt{3}}{2})(s + 1)$$

$$G(s) = s^3 + 2s^2 + 2s + 1$$

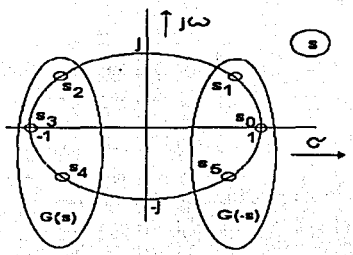


Figura 1.5: Ceros de la función de transferencia de Butterworth para n=3

Ejemplo 2:

Calcular la impedancia de entrada para el filtro paso bajas normalizado del ejercicio anterior.

Solución:

Del problema anterior se tiene que la función de transferencia es

$$G(s) = s^3 + 2s^2 + 2s + 1$$

y la función característica

$$\phi(s) = \epsilon s^n = s^3.$$

sustituyendo estos valores en la ecuación (1.15)

$$Z_{entrada}(s) = \frac{G(s) + \phi(s)}{G(s) - \phi(s)} \qquad \text{.....(1.15)}$$

se obtiene

$$Z_{entrada} = \frac{2s^3 + 2s^2 + 2s + 1}{2s^2 + 2s + 1}$$

por último, desarrollando esta ecuación mediante un quebrado de escalera se tiene:

$$Z_{entrada} = s + \frac{1}{2s + \frac{1}{s+\frac{1}{1}}}$$

El filtro Butterworth resultante se observa en la figura 1.6. Está terminado en la puerta de salida con una resistencia de 1 Ohm y en la frecuencia de corte ω_1=1 tiene una atenuación de 3 dB.

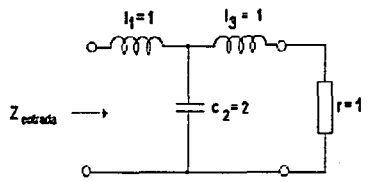


Figura 1.6: Filtro Butterworth de tercer orden.

1.3.2 Aproximación Chebychev[5]

La función característica para la aproximación de Chebychev es de la forma

$$\phi(\Omega) = \cos(n\ arc\cos(\Omega)) \qquad para\ \Omega \leq 1 \qquad .....(1.16)$$

$$\phi(\Omega) = \cosh(n\ arg\cosh(\Omega)) \qquad para\ \Omega \geq 1 \qquad .....(1.17)$$

La ecuación característica toma la forma

$$G(\Omega)G(-\Omega) = 1 + \epsilon^2 \cos^2(n\ arc\cos(\Omega)) \qquad .....(1.18)$$

$$G(\Omega)G(-\Omega) = 1 + \epsilon^2 \cos^2 h(n\ arg\cosh(\Omega)) \qquad .....(1.19)$$

El orden del filtro para la aproximación Chebychev se obtiene de manera similar que en el caso del filtro Butterworth.

Para $\Omega = \Omega_1$ se tiene la atenuación a = a_{max} y de (1.19) se puede escribir

$$e^{2a_{max}} = 1 + \epsilon^2 cos^2(n.argcosh(1))$$

y calculando ε

$$\epsilon = \sqrt{e^{2a_{max}} - 1}$$

Para $\Omega = \Omega_2$ se tiene la atenuación a = a_{min} y de (1.19) se puede calcular n.

$$e^{2a_{min}} = 1 + \epsilon^2 cos^2 h(n\ argcosh(\Omega_2))$$

Si se calcula el logaritmo natural de ambos lados de la ecuación anterior se obtiene la ecuación que nos permite calcular el orden del filtro para la aproximación Chebychev.

$$n \geq \frac{argcosh\sqrt{\frac{e^{2a_{min}}-1}{e^{2a_{max}}-1}}}{argcosh(\Omega_2)} \qquad .....(1.20)$$

En la ecuación (1.20) la atenuación a se sustituye en Nepers. Para emplear la atenuación en Decibeles es necesario emplear cualquiera de las ecuaciones:

$$n \geq \frac{argcosh\sqrt{\frac{e^{0.23a_{min}}-1}{e^{0.23a_{max}}-1}}}{argcosh(\Omega_2)} \qquad .....(1.21)$$

[5] Estos mismos cálculos los puede encontrar en libro de Procesamiento de Señales, del Dr. Bohumil Pšenička, pp 75-80

$$n \geq \frac{argcosh\sqrt{\frac{10^{\frac{a_{min}}{10}}-1}{10^{\frac{a_{max}}{10}}-1}}}{argcosh(\Omega_2)} \quad \text{.....(1.22)}$$

Para calcular la función de transferencia G(s) se calcula los ceros de la ecuación

$$G(\Omega)G(-\Omega) = 1 + \epsilon^2 \cos^2(n arccos(\Omega)) \quad \text{.....(1.23)}$$

Si se sustituye:

$$\arccos(\Omega) = \theta_1 + j\theta_2$$

se obtiene:

$$\Omega = cos(\theta_1 + j\theta_2) = cos(\theta_1)cosh(\theta_2) + jsen(\theta_1)senh(\theta_2) \quad \text{.....(1.24)}$$

De la ecuación (1.23) se tiene:

$$\cos(n\theta_1 + jn\theta_2) = \frac{j}{\epsilon}$$

Desarrollando la función anterior de forma trigonométrica se obtiene:

$$\cos(n\theta_1)cosh(n\theta_2) + jsen(n\theta_1)scnh(n\theta_2) = \frac{j}{\epsilon}$$

Comparando las partes real e imaginaria de ambos miembros de la ecuación anterior tenemos:

$$\cos(n\theta_1)cosh(n\theta_2) = 0 \quad \text{.....(1.25)}$$

$$scn(n\theta_1)senh(n\theta_2) = \frac{1}{\epsilon} \quad \text{.....(1.26)}$$

La función cosh(nθ2) para ningún valor de nθ2 nθ2 es igual a cero. Por lo que el primer término de la ecuación (1.25) es el que debe ser igual a cero. Esta ecuación es igual a cero para:

$$\theta_1 = \frac{(2k+1)}{2}\pi \quad \text{.....(1.27)}$$

si se sustituye (1.27) en (1.26) se obtiene:

$$sen(\frac{(2k+1)}{2}\pi)senh(n\theta_2) = \frac{1}{\epsilon}$$

pero sen((2k+1)π/2) = 1; por lo que la ecuación anterior toma la forma:

$$\theta_2 = \frac{1}{n} argsenh(\frac{1}{\epsilon}) \quad \text{.....(1.28)}$$

De la ecuación (1.24) se obtiene la siguiente ecuación:

$$s_k = j\Omega_k = -sen(\theta_1).senh(\theta_2) \pm jcos(\theta_1).cos(\theta_2) \quad \text{.....(1.29)}$$

Mediante (1.29) se calculan los ceros de la función de transferencia del filtro Chebychev para cualquier orden del filtro y cualquier atenuación a_{max}. Los ceros del filtro Chebychev están ubicados en una elipse y en la parte izquierda del plano s. En el siguiente ejemplo se muestra como se calcula un filtro de tercer orden mediante la aproximación Chebychev.

Ejemplo 3:

Diseñar un filtro que tenga una a_{max} = 2 dB y el orden sea de 3, mediante la aproximación de Chebychev:

Primero se calcula ε:

$$\epsilon = \sqrt{10^{a_{max}/10} - 1} = \sqrt{10^{2/10} - 1} = 0,764$$

$$\theta_2 = \frac{1}{3} argsenh(\frac{1}{0,764}) = 0,358$$

$$cosh(0,358) = 1,06 \qquad senh(0,358) = 0,366$$

y los ceros de la función de transferencia son:

$$s_0 = -0,366.sen(\frac{\pi}{6}) \pm j1,065.cos(\frac{\pi}{6}) = -0,183 \pm j0,922$$

$$s_1 = -0,366.sen(\frac{3\pi}{6}) \pm j1,065.cos(\frac{3\pi}{6}) = -0,366$$

El resto de los ceros no son necesarios calcularlos, ya que son complejos

$$G(s) = (s + 0,366)(s + 0,183 + j0,922)(s + 0,183 - j0,922)$$

$$G(s) = s^3 + 0,73329s^2 + 1,01859s + 0,324665$$

conjugados y están ubicados en la elipse, como se ve en la figura 1.7. La función de transferencia es:

A primera vista se observa que para w=0 la atenuación no es igual a cero. Por lo que es necesario multiplicar la ecuación anterior por una constante (k) tal que para s=0 se tenga G(s) = 1. La constante se obtiene de la ecuación siguiente:

$$G(s)|_{s=0} = K(s^3 + 0,73392s^2 + 1,01859s + 0,324665)|_{s=0} = 1$$

Entonces

$$K = \frac{1}{0,324066} = 3,0857855$$

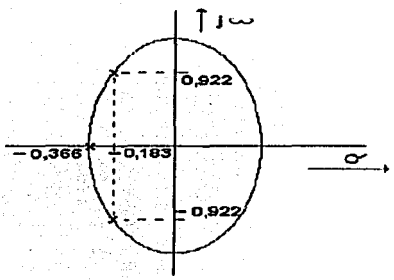


Figura 1.7: Ubicación de los ceros del filtro Chebychev.

La función de transferencia toma la forma:

$$G(s) = 3,085588855s^3 + 2,261678s^2 + 3,143172.1s + 1 \quad .....(1.30)$$

Para verificar la validez de (1.30) evaluamos en w=1, donde la atenuación debe ser igual a 2 dB.

$$G(j) = -j3,0855855 - 2,2616 + 3,14317j + 1 = -1,2616 + j0,05758$$

$$a = 20.log\sqrt{1,2616^2 + 0,0575869^2} = 2,03\ [dB]$$

Para calcular la impedancia de entrada mediante (1.31) es necesario calcular la función característica $\phi(s)$ en la forma polinomial.

$$Z_{entrada} = \frac{G(s) \pm \phi(s)}{G(s) \mp \phi(s)} \quad .....(1.31)$$

Para n=3 la función característica toma la forma:

$$\phi_3(s) = \sqrt{e^{0,23a_{max}} - 1}(4s^3 + 3s)$$

$$\phi_3(s) = \sqrt{e^{0,23.2} - 1}(4s^3 + 3s) = 3,0855855s^3 + 2,3141892s$$

y la impedancia de entrada del filtro Chebychev es:

$$Z_{entr}(s) = \frac{G(s) + \phi(s)}{G(s) - \phi(s)} = \frac{6,171s^3 + 2,2616s^2 + 5,457s + 1}{2,2616s^2 + 0,8289s + 1}$$

si se desarrolla la impedancia Z(s) en el quebrado de escalera, se obtiene el circuito que se muestra en la figura 1.8

$$Z(s) = 2,728s + \frac{1}{0,8289s + \frac{1}{2,728s + \frac{1}{1}}}$$

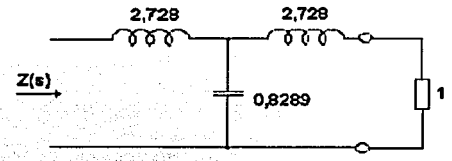


Figura 1.8:Filtro Chebychev de tercer orden.

Si se requiere que el filtro tenga en la salida impedancia infinita, se calcula la impedancia del filtro mediante la ecuación (1.32)

$$Z(s)_{entr} = \frac{\text{parte impar de G(s)}}{\text{parte par de G(s)}} \quad \text{.....(1.32)}$$

$$Z_{entr}(s) = \frac{3,08558s^3 + 3,14317s}{2,2116s^2 + 1}$$

$$Z_{entr} = 1,36429s + \frac{1}{1,2714s + \frac{1}{1,2788s}}$$

. El circuito que trabaja con la salida abierta se muestra en la figura 1.9b. El circuito en la figura 1.9ª no se puede utilizar, porque la última inductancia no

influye en la función de transferencia y es necesario utilizar el circuito dual que se muestra en la figura 1.9b.

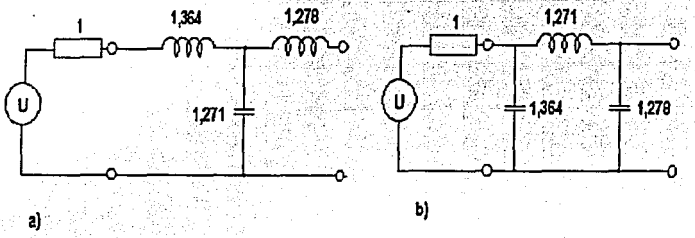


Figura 1.9: Filtro Chebychev de tercer orden.

Además de las aproximaciones vistas, existen algunas otras como la de Chebychev inverso o la elíptica, pero estas están fuera del objetivo de este documento, si el lector desea conocerlas se le recomienda consulte el libro del Dr. Bohumil Pšenička, Procesamiento de Señales.

1.4 Filtros digitales.

Hasta el momento sólo se ha visto la implementación de filtros mediante la realización de las impedancias a través de una red eléctrica, pero también es posible llevarlos a cabo mediante software, con la ventaja de que no es necesario construir los materiales eléctricos, (como los inductores) que en la mayoría de los casos son de difícil construcción. Los filtros digitales también son empleados para simular el funcionamiento del filtro antes de llevar a cabo su realización física.

Entenderemos como filtro digital a un filtro que trabaja con señales digitales, como por ejemplo el sonido almacenado en la computadora. En términos computacionales un filtro toma una secuencia de números (señal de entrada) y produce una nueva secuencia (señal de salida).

Los filtros digitales pueden realizarse empleando elementos correspondientes a las operaciones de multiplicación, suma y almacenamiento de datos. El almacenamiento de datos consiste en retrasar su uso normalmente una cantidad de tipo igual z^{-1} al periodo de muestreo. Este retraso se representa mediante z^{-1} (retraso de una unidad), z^{-2} (dos unidades), etc. Los símbolos que representan dichas operaciones se muestran en la figura 1.10.

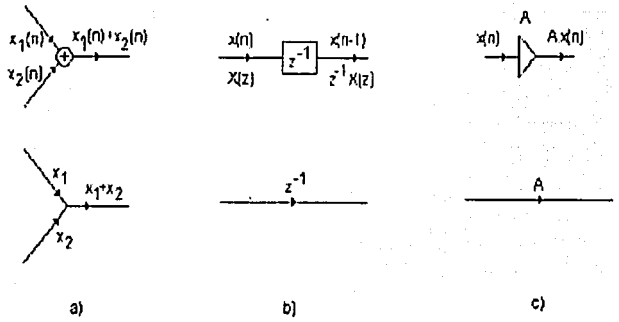


Figura 1.10:a) El sumador, b) elemento de retardo, c)el multiplicador.

Ejemplo 4:

El filtro paso bajas mas sencillo (esto no significa que sea ideal) esta dado por la siguiente ecuación diferencial[6]:

$$y(n) = x(n) + x(n-1) \qquad \text{.....(1.33)}$$

donde x(n) es la amplitud de la señal de entrada (muestra) en el tiempo n, y y(n) es la amplitud de la salida en el tiempo n. El diagrama del sistema para este pequeño filtro se observa en la figura 1.11.

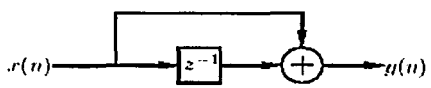


Figura 1.11:Diagrama para el filtro y(n) = x(n) + x(n-1)

[6]Introducción a los filtros digitales (DRAFT), J.O. Smith, CCRMA, Stanford. La última versión de este documento esta disponible en http://www-ccrma.stanford.edu/~jos/filters/.

Yendo un poco más lejos en nuestro ejemplo, permítanme escribir una subrutina que implemente la ecuación (1.33). Para la computadora, x(n) y y(n) son dos arreglos de datos y n es el índice del arreglo. Por lo general los datos de entrada son mayores a la cantidad de memoria que se dispone en la computadora, por lo que se deben procesar en bloques de un tamaño razonable. Llamaremos M a la cantidad de muestras que se procesaran en una iteración. En lenguaje C la subrutina quedaría de la siguiente manera:

```
/*************************************************************
Implementación del filtro paso bajas.

y(n) = x(n) + x(n-1)

*************************************************************/

double filtpb(double *x, double *y, int M, double xn1)
{
    int n;
  y[0]=x[0] + xn1;
  for(n=1;n<M;M++)
    y[n]=x[n]+x[n-1];
  return x[M-1];
}
```

En esta implementación el primer valor de x(n-1) es dado por el argumento xn1. Los arreglos x y y también son pasados como argumentos y por conveniencia se regresa el valor de xn1 apropiado para la siguiente llamada de filtpb. En la figura 1.11 se puede observar un sencillo programa que manda llamar a filtpb. La señal de entrada de longitud de 10 muestras es procesada en dos bloques de 5.

```
/*************************************************************
Programa principal para probar a filtpb
*************************************************************/
void main(void){
    double x[10] = {1,2,3,4,5,6,7,8,9,10};
  double y[10];
  int i;
  int M=5; /* Tamaño del bloque*/
  double xn1 = 0 /*valor inicial de x(n-1)*/
  xn1 = filtpb(x,y,M,xn1);
  xn1 = filtpb(&x[M],&y[M],M,xn1);
  /*Imprimimos el valor de y*/
  for(i=0; i<2*M; i++)
    printf("x[%d]=%f\ty[%d]=%f\n",i,x[i],i,y[i]);
  exit(0);
```

```
}
/* A la salida tenemos:
x[0]=1.000000     y[0]=1.000000
x[1]=2.000000     y[1]=3.000000
x[2]=3.000000     y[2]=5.000000
x[3]=4.000000     y[3]=7.000000
x[4]=5.000000     y[4]=9.000000
x[5]=6.000000     y[5]=11.000000
x[6]=7.000000     y[6]=13.000000
x[7]=8.000000     y[7]=15.000000
x[8]=9.000000     y[8]=17.000000
x[9]=10.000000    y[0]=19.000000
*/
```

Figura 1.11: Programa principal que manda a llamar a la función filtpb.

Como se observa en el ejemplo 4, para la realización de filtros digitales es necesario primero conocer la estructura que lo representa, ya que de esta se obtendrán las ecuaciones que describen dicho filtro, las cuales podrán ser implementadas mediante algún algoritmo.

La manera más simple de diseñar un filtro digital es a través de la respuesta impulso, es decir, se busca una función de transferencia discreta de un filtro digital que tenga la misma respuesta a un impulso que el circuito analógico. A este método se le conoce como impulse Invariance[7] (Invariancia de impulsos). Si se conoce la forma analítica de la respuesta a un impulso la transformación puede llevarse a cabo usando la transformada zeta.

$$H(z) = z\{f(t)\}$$

[7] Procesamiento de Señales, Dr. Bohumil Pšenička, pp 292.

Ejemplo 5:

Diseñar el filtro digital cuya respuesta impulso sea igual a la respuesta del circuito de la figura 1.12. Elija la frecuencia de muestreo fm = 8000 Hz.

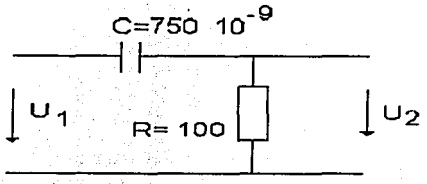


Figura 1.12: Circuito RC

Primero se calcula la función de transferencia H(s) del circuito y después mediante la transformada de Laplace se calcula la respuesta a un impulso discreto h(nT). Mediante la transformada Zeta se calcula H(z).

$$H(s) = \frac{U_2}{U_1} = \frac{s}{s + \frac{1}{CR}} = \frac{s}{s + 1333,3}$$

$$h(t) = L^{-1}\left\{1 - \frac{1333,3}{s + 1333,3}\right\} = \delta(t) - 1333,3e^{-1333,3t}$$

$$h(nT) = \delta(nT) - 1333,3e^{-1333,3nT} = \delta(nT) - \frac{1333,3}{8000}e^{-\frac{1333,3}{8000}n}$$

La función de transferencia H(z) se calcula mediante la transformada z:

$$H(z) = Z\left\{\delta(nT) - 0,1666e^{-0,1666n}\right\} = 1 + \frac{-0,16666}{1 - e^{-0,1666}z^{-1}}$$

$$H(z) = 1 + \frac{-0,16666}{1 - 0,846z^{-1}}$$

La estructura correspondiente a la función de transferencia H(z) se muestra en la figura 1.13. Del diagrama se pueden obtener las ecuaciones que modelan al circuito, para ello llamaremos N al nodo que se encuentra a la salida del sumador, haciendo esta consideración tenemos que:

$$y(n) = x(n) - 0.1666N(n) \quad \text{.....(1.34)}$$

$$N(n) = x(n) + 0.846N(n-1) \quad \text{.....(1.35)}$$

Las ecuaciones (1.34) y (1.35) pueden ser fácilmente implementadas mediante software.

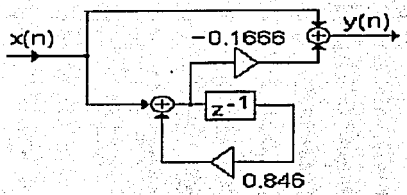
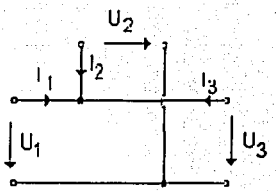


Figura 1.13: Circuito discreto equivalente al circuito RC.

Referencias.

[2] Dr. Bohumil Pšenička, Procesamiento de Señales, pp 65
[3] Dr. Bohumil Pšenička, Procesamiento de Señales, pp 65
[4] IDEM, pp 66-70
[5] IDEM, pp 75-80
[6]Introducción a los filtros digitales (DRAFT), J.O. Smith, CCRMA, Stanford. La última versión de este documento esta disponible en http://www-ccrma.stanford.edu/~jos/filters/.

[7] Dr. Bohumil Pšenička, Procesamiento de Señales, pp 292



Capítulo 2

Teoría de los Filtros digitales de onda.

En este capítulo, se discute la teoría básica de los filtros digitales de onda, así como la sustitución de los elementos analógicos de los filtros por elementos digitales. Por último se analizan los adaptadores serie y paralelo, que son los elementos constitutivos de los filtros de onda.

2.1 ¿Qué son los filtros digitales de onda?

Los filtros digitales de onda surgen como una forma de discretizar a los filtros analógicos que contienen resistencias, capacitores, inductores, transformadores, etc.

Una manera de abordar el problema de la discretización, es el de llevar a cabo la integración numérica de las ecuaciones diferenciales (comúnmente llamadas ODE, ordinary differential equation) que describen al circuito analógico. Como ejemplo consideremos lo que ocurre al discretizar el siguiente filtro LC:

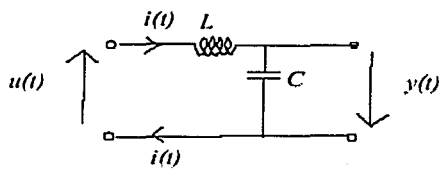


Figura 2.1: Filtro LC.

Donde: u(t) es el voltaje de entrada.
y(t) es el voltajes de salida.
i(t) es la corriente y
v(t) es el voltaje a en el inductor.

Para este circuito se tienen tres ecuaciones diferenciales que lo describen:

$$v(t) = L\frac{di}{dt}$$
$$i(t) = C\frac{dy}{dt}$$
$$u(t) = v(t) + y(t)$$

Las primeras dos ecuaciones provienen de la definición del inductor y del capacitor respectivamente, la tercera de la ley de voltajes de Kirchoff's. Discretizando estas ecuaciones por medio de la regla del trapezoide para la integración numérica (esto es equivalente a la transformada bilinial para pasar del plano s al plano z), se obtiene:

$$i_n = i_{n-1} + \frac{T}{2L}(v_n + v_{n-1}) \quad \text{.....(2.1)}$$
$$y_n = y_{n-1} + \frac{T}{2C}(i_n + i_{n-1}) \quad \text{.....(2.2)}$$
$$v_n = -(u_n + y_n) \quad \text{.....(2.3)}$$

Donde T es el periodo de muestreo.
De las ecuaciones anteriores se tiene el siguiente diagrama:

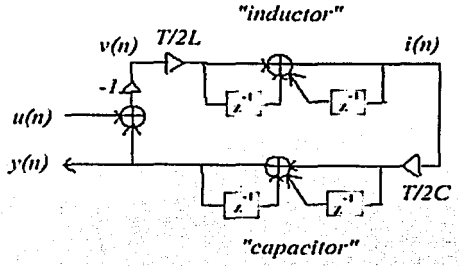


Figura 2.2: Esquema del circuito digital.

Desgraciadamente la no existencia de un elemento de retardo en un lazo hace imposible la implementación directa de este circuito.

Para resolver este problema, Alfred Fettweis[1] introdujo las variables de onda:

$$a_n = v_n + i_n R_0$$
$$b_n = v_n - i_n R_0$$

donde R_0 es un parámetro arbitrario llamado resistencia de puerto. Sumando las ecuaciones anteriores tenemos que:

$$v_n = \frac{a_n + b_n}{2} \quad \text{.....(2.4)}$$

y si se restan se obtiene:

$$i_n = \frac{a_n - b_n}{2R_0} \quad \text{.....(2.5)}$$

2.2 Obtención del circuito equivalente del inductor.

Sustituyendo las ecuaciones (2.4) y (2.5) en la ecuación que define al inductor (2.1) se tiene que:

$$\frac{a_n - b_n}{2R_0} = \frac{a_{n-1} - b_{n-1}}{2R_0} + \frac{T}{2L}\left(\frac{a_n + b_n}{2} + \frac{a_{n-1} + b_{n-1}}{2}\right)$$

Ahora bien, seleccionando $R_0 = 2L/T$ se obtiene:

$$\begin{aligned} a_n - b_n &= a_{n-1} - b_{n-1} + (a_n + b_n + a_{n-1} + b_{n-1}) \\ &= 2a_{n-1} + a_n + b_n \end{aligned}$$

por último, simplificando se llega a:

$$b_n = -a_{n-1} \quad \text{.....(2.6)}$$

[1] Wave Digital Filter, Stefan Bilbao, Julios O Smith, Stanford University

que es la ecuación de onda del inductor. En términos de z podemos reescribir (2.6) como:

$$B = -Az^{-1} \quad \text{.....(2.7)}$$

La ecuación (2.7) puede realizarse mediante un multiplicador de valor –1 y con un elemento de retardo en cascada. La realización de un inductor en la forma discreta se muestra en la figura 2.3.

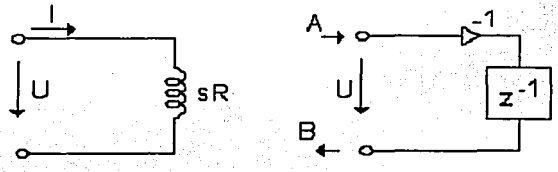


Figura 2.3: Sustitución del inductor por un circuito discreto.

2.3 Obtención del circuito equivalente del capacitor.

De la definición del capacitor se tiene que:

$$i = c\frac{dv}{dt} \quad \text{.....(2.8)}$$

de forma integral se obtiene:

$$v = \frac{1}{c}\int idt \quad \text{.....(2.9)}$$

integrando por medio de la regla del trapezoide tenemos:

$$v_n = v_{n-1} + \frac{T}{2c}(i_n + i_{n-1}) \quad \text{.....(2.10)}$$

sustituyendo las ecuaciones (2.4) y (2.5) en (2.10) se llega a:

$$\frac{a_n+b_n}{2}=\frac{a_{n-1}+b_{n-1}}{2}+\frac{T}{2c}\left(\frac{a_n-b_n}{2R_o}+\frac{a_{n-1}-b_{n-1}}{2R_o}\right) \qquad ..(2.11)$$

multiplicando ambos miembros de (2.11) por 2 y seleccionando $R_o=T/2c$ se obtiene:

$$a_n+b_n=a_{n-1}+b_{n-1}+a_n-b_n+a_{n-1}-b_{n-1}$$
$$a_n+b_n=2a_{n-1}+a_n-b_n$$

por último tenemos:

$$b_n=a_{n-1} \qquad .....(2.12)$$

que es la ecuación de onda del capacitor. En términos de z se tiene que:

$$B=Az^{-1} \qquad .....(2.13)$$

La ecuación (2.13) puede realizarse mediante un elemento de retardo. De manera similar podremos encontrar los circuitos discretos que sustituyen al transformador, al girador, etc. El lector podrá encontrar estas sustituciones de circuitos analógicos en la bibliografía FET[2]

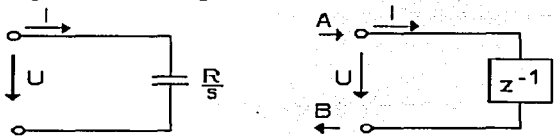


Figura 2.4: Sustitución del capacitor por un circuito discreto.

2.4 Los adaptadores.

En un filtro de onda los sumadores y multiplicadores pueden ser agrupados en bloques de n puertas, a estos bloques se les llama adaptadores. Estos adaptadores son los principales componentes que conforman un filtro digital de onda. Los podemos dividir en los adaptadores serie y paralelo dependientes y los no dependientes. Si los adaptadores son terminados en una de sus puertas con un elemento de retraso representan un capacitor o un inductor.

[2] Digital Filters Structures Related to Clasical Filter Networks, Fettweis A., Arch. Elektr. Uebertragung, vol 25,pp 78-89

2.4.1 Adaptador paralelo dependiente.

El adaptador paralelo es un circuito independiente de la frecuencia. Si se conecta a un puerto un elemento de retardo se obtiene un circuito dependiente de la frecuencia. El adaptador paralelo se utiliza con el fin de conectar los elementos L y C en cascada.

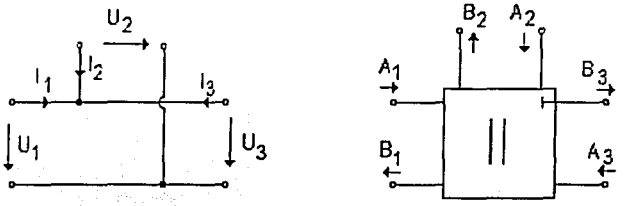


Figura 2.5: Gráfica del bloque y símbolo del adaptador dependiente.

La onda reflejada B y transmitida A en las puertas de los adaptadores se obtiene a partir del voltaje U y la corriente I, mediante las ecuaciones:

$$A = U + RI \qquad B = U - RI \quad \text{.....(2.14)}$$

para el adaptador paralelo de la figura 2.5 se tiene que:

$$U_1 = U_2 = U_3 \qquad I_1 + I_2 + I_3 = 0 \quad \text{....(2.15)}$$

aplicando (2.14) a cada una de las puertas del adaptador se tiene:

$$\begin{aligned} A_1 &= U_1 + R_1 I_1 & B_1 &= U_1 - R_1 I_1 \\ A_2 &= U_2 + R_2 I_2 & B_2 &= U_2 - R_2 I_2 \\ A_3 &= U_3 + R_3 I_3 & B_3 &= U_3 - R_3 I_3 \end{aligned} \quad \text{.....(2.16)}$$

ahora bien considerando que $U_1=U_2=U_3$ las ecuaciones anteriores se pueden reescribir de la forma:

$$\begin{aligned} A_1 &= U_1 + R_1 I_1 & B_1 &= U_1 - R_1 I_1 \\ A_2 &= U_1 + R_2 I_2 & B_2 &= U_1 - R_2 I_2 \\ A_3 &= U_1 + R_3 I_3 & B_3 &= U_1 - R_3 I_3 \end{aligned} \quad \text{.....(2.17)}$$

Despejando las corrientes de las ecuaciones anteriores se tiene que:

$$I_1 = \frac{A_1 - U_1}{R_1}$$
$$I_2 = \frac{A_2 - U_1}{R_2}$$
$$I_3 = \frac{A_3 - U_1}{R_3}$$

Ahora bien, aplicando $I_1+I_2+I_3=0$ y despejando U_1 de las ecuaciones anteriores se tiene que:

$$U_1 = \frac{A_1G_1 + A_2G_2 + A_3G_3}{G_1 + G_2 + G_3} \quad \text{.....(2.18)}$$

donde:

$$G_i = \frac{1}{R_i}$$

Sumando una a una las ecuaciones (2.17), sustituyendo U_1 y despejando B_i se obtiene:

$$B_1 = (\alpha_1A_1 + \alpha_2A_2 + \alpha_3A_3) - A_1$$
$$B_2 = (\alpha_1A_1 + \alpha_2A_2 + \alpha_3A_3) - A_2 \quad \text{.....(2.19)}$$
$$B_3 = (\alpha_1A_1 + \alpha_2A_2 + \alpha_3A_3) - A_3$$

donde los coeficientes α_i están expresados por:

$$\alpha_i = \frac{2G_i}{G_1 + G_2 + G_3} \qquad i = 1,2,3. \quad \text{.....(2.20)}$$

Si la puerta tres se encuentra libre de reflexión, esto es $G_3=G_1+G_2$, se tiene que:

$$\alpha_3 = \frac{2(G_1 + G_2)}{2(G_1 + G_2)} = 1$$

Por lo que en el circuito α_3=1. Para α_3=1 las ecuaciones (2.19) toman la forma:

$$\begin{aligned} B_1 &= (\alpha_1 A_1 + \alpha_2 A_2 + A_3) - A_1 \\ B_2 &= (\alpha_1 A_1 + \alpha_2 A_2 + A_3) - A_2 \\ B_3 &= \alpha_1 A_1 + \alpha_2 A_2 \end{aligned} \qquad \text{.....(2.21)}$$

De las ecuaciones (2.21) se puede dibujar la estructura que se muestra en la figura 2.6. Los coeficientes α_1 y α_2 son los valores de los multiplicadores.

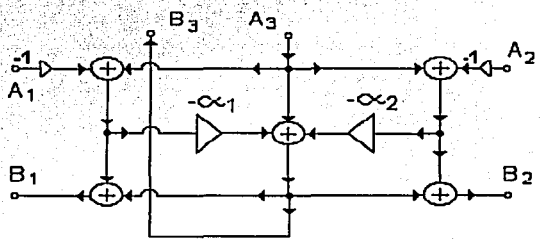


Figura 2.6: Adaptador paralelo dependiente.

2.4.2 Adaptador serie dependiente.

Un adaptador serie, representado en la figura 2.7, sirve par simular las conexiones de circuitos en serie.

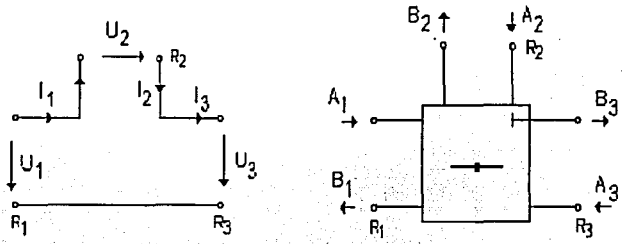


Figura 2.7: Gráfica y símbolo del adaptador serie.

Del esquema se obtiene la relación $I_1=I_2=I_3$, aplicando esta a las ecuaciones (2.16) se obtiene:

$$\begin{aligned} A_1 &= U_1 + R_1 I_1 & B_1 &= U_1 - R_1 I_1 \\ A_2 &= U_2 + R_2 I_1 & B_2 &= U_2 - R_2 I_1 \\ A_3 &= U_3 + R_3 I_1 & B_3 &= U_3 - R_3 I_1 \end{aligned} \qquad \text{.....(2.22)}$$

ahora bien, si se despejan los voltajes de las primeras tres ecuaciones de (2.22), y se aplica la relación $U_1+U_2+U_3=0$, se obtiene que:

$$I_1 = \frac{A_1 + A_2 + A_3}{R_1 + R_2 + R_3} \qquad \text{.....(2.23)}$$

por último si se restan las ecuaciones (2.22) de la forma, $A_i - B_i$, sustituyendo el valor de I_1 y despejando B_i se llega a:

$$\begin{aligned} B_1 &= A_1 - \beta_1(A_1 + A_2 + A_3) \\ B_2 &= A_2 - \beta_2(A_1 + A_2 + A_3) \\ B_3 &= A_3 - \beta_3(A_1 + A_2 + A_3) \end{aligned} \qquad \text{.....(2.24)}$$

donde los coeficientes β_i, son los valores de los multiplicadores, y se calculan mediante la ecuación:

$$\beta_i = \frac{2R_i}{R_1 + R_2 + R_3} \quad \text{para i = 1,2,3} \qquad \text{.....(2.25)}$$

Nuevamente si la última puerta se encuentra libre de reflexión, esto es $R_3=R_1+R_2$, por tanto $\beta_3=1$ y las ecuaciones (2.24) toman la forma:

$$\begin{aligned} B_1 &= A_1 - \beta_1(A_1 + A_2 + A_3) \\ B_2 &= A_2 - \beta_2(A_1 + A_2 + A_3) \\ B_3 &= -A_1 - A_2 \end{aligned} \qquad \text{.....(2.26)}$$

La estructura del adaptador serie que realiza las ecuaciones (2.26) se muestra en la figura (2.8)

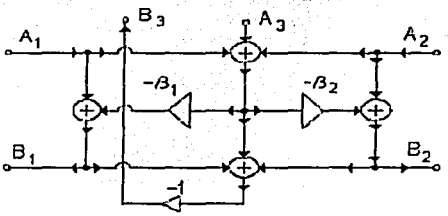


Figura 2.8: Estructura del adaptador serie dependiente.

2.4.3 Adaptadores Elementales.

A los adaptadores que contienen sólo un multiplicador se le llama elementales. Estos adaptadores son de particular interés en el diseño de Filtros Digitales de Onda, ya que son la base para la construcción de estos. Se dividen también en serie y paralelo. Este tipo de estructura no puede emplearse al final de los circuitos donde la impedancia de salida se encuentra definida.

El adaptador serie elemental se muestra en la figura 2.9 y el valor del multiplicador β se calcula mediante la ecuación:

$$\beta = \frac{R_1}{R_1 + Z_2(\omega_1)} \qquad \text{.....(2.27)}$$

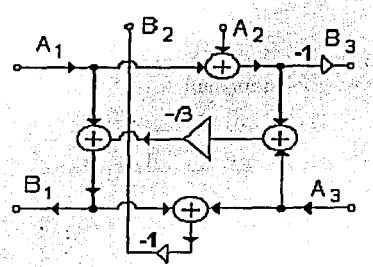


Figura 2.9: Adaptador serie elemental.

El adaptador paralelo elemental se muestra en la figura 2.10 y el valor del multiplicador se calcula usando la ecuación:

$$\alpha = \frac{G_1}{G_1 + Y_2(\omega_1)} \qquad \text{.....(2.28)}$$

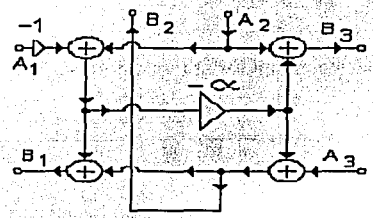


Figura 2.10: Adaptador paralelo elemental.

Ejemplo 1[3]:

Diseñar un filtro digital de onda paso bajas de orden N=2. La ganancia debe ser de 3 dB en la frecuencia ω= 1 rad/sec. Utilice la aproximación Butterworth.

Solución:

De tablas encontramos los valores L C para el filtro Butterworth de segundo orden. El filtro paso bajas se muestra en la figura 2.11. De la misma figura se observa que el circuito requiere un adaptador paralelo elemental y un adaptador serie dependiente.

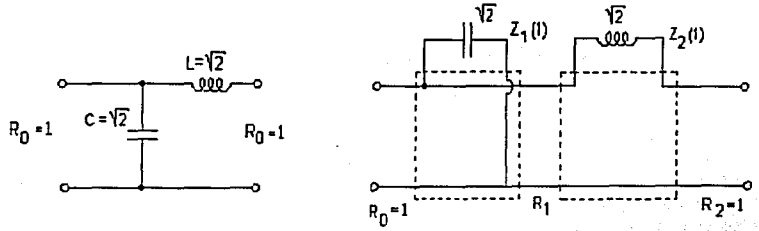


Figura 2.11: Filtro Butterworth paso bajas

[3] El presente ejemplo se tómo del libro: Procesamiento de Señales, Dr. Bohumil Pšenička, pp 362

El valor del coeficiente del adaptador paralelo elemental se obtiene mediante la ecuación (2.28), en esta caso toma la forma:

$$A_1 = \frac{G_0}{G_0 + Y_1(1)} = \frac{1}{1 + \sqrt{2}} = 0,414213$$

Los valores del adaptador serie dependiente se obtienen mediante la ecuación (2.25) y son:

$$B_{21} = \frac{2R_1}{R_1 + Z_2(1) + R_2} = \frac{2 \times 0,414213}{0.41421 + \sqrt{2} + 1} = 0.292893$$

$$B_{22} = \frac{2}{0,41423 + \sqrt{2} + 1} = 0,707106$$

La estructura resultante se observa en la figura 2.12

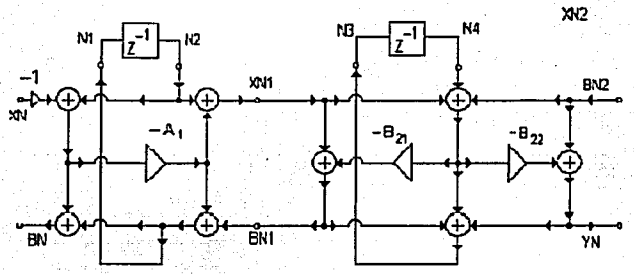


Figura 2.12: Filtro Digital de Onda de orden dos.

Como se puede observar en el ejemplo, el diseño de los filtros digitales de onda es muy simple, porque podemos emplear el catálogo de los filtros LC directamente sin la necesidad de emplear la transformada bilineal para calcular la función de transferencia H(z). La respuesta de estos filtros no es tan sensible a redondeo o truncamiento de los coeficientes como lo es en el caso de los filtros calculados mediante la transformada z. Para implementar estos filtros se pueden emplear con un microcontrolador que trabaje con punto fijo y es posible expresar los coeficientes con pocos bits.

Ejemplo 2:

Diseñar un filtro digital de onda paso altas de orden N=3. La ganancia debe ser de 3 dB en la frecuencia ω= 1 rad/sec. Utilice la aproximación Butterworth.

Solución:

De tablas encontramos los valores L C para el filtro Butterworth de tercer orden. El filtro paso altas se muestra en la figura 2.13. De la misma figura se observa que el circuito requiere un adaptador paralelo elemental, adaptador serie elemental y un adaptador paralelo independiente.

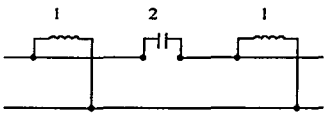


Figura 2.13: Filtro Butterworth de 3 orden, paso altas.

El valor del coeficiente del adaptador paralelo elemental se obtiene mediante la ecuación (2.28), en esta caso toma la forma:

$$A_1 = \frac{1}{1+1} = 0.5$$

Para el adaptador elemental serie se tiene que:

$$B_2 = \frac{R_1}{R_2 + R_1} = \frac{0.5}{2+0.5} = 0.2$$

Por último se obtienen los valores para el adaptador paralelo dependiente a partir de la ecuación (2.20):

$$A_{31} = \frac{0.8}{0.4+1+1} = 0.33333$$

$$A_{32} = \frac{2}{2.4} = 0.83333$$

La estructura resultante se observa en la figura 2.14.

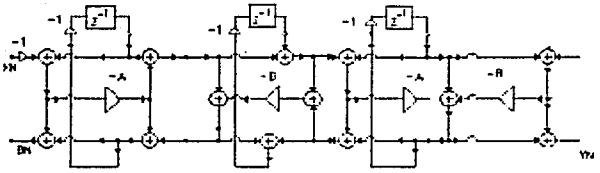
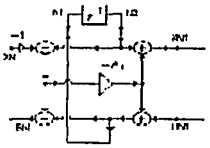


Figura 2.14: Filtro Digital de Orden Tres Paso Altas.

Si se comparan las estructuras de las figuras 2.12 y 2.14 se observa que la única diferencia entre ellas es que, en el filtro paso altas se observa un factor menos uno a la entrada de los retardadores, este se debe a que para obtener un filtro LC paso altas a partir de un paso bajas se intercambian los capacitores por inductores y viceversa, y esto se ve reflejado en la estructura del filtro digital con la aparición de este factor, ya que si se recuerdan los cálculos de los circuitos equivalentes, presentados al inicio de este capítulo, se observará que el factor menos uno aparece en el circuito equivalente del inductor. La importancia de este hecho se retomará en los capítulos siguientes, cuando se desarrollen las ecuaciones para calcular los filtros paso altas.

Referencias:

[1] Wave Digital Filter, Stefan Bilbao, Julios O Smith, Stanford University
[2] Digital Filters Structures Related to Clasical Filter Networks, Fettweis A., Arch. Elektr. Uebertragung, vol 25,pp. 78-89
[3] Dr. Bohumil Pšenička, Procesamiento de Señales pp 362



Capítulo 3

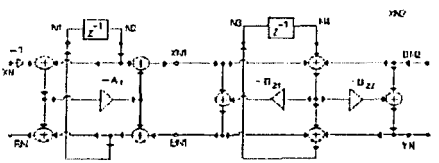
SÍNTESIS DE LAS ECUACIONES.

En el capítulo anterior se observó la forma de cómo a partir de un filtro LC se obtiene la estructura del filtro digital de onda, por lo que en el presente capítulo sólo se presentarán las estructuras omitiendo los cálculos empleados para obtenerlas, y a partir de dichas estructuras se deducirán las ecuaciones que modelan al filtro digital; también se analizan dichas ecuaciones para el desarrollo de los algoritmos que nos permitan llevar a cabo su implementación de manera genérica, es decir, se buscará un algoritmo para la realización del filtro digital de onda independientemente de su orden o tipo (paso altas o paso bajas).

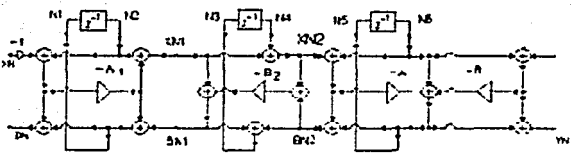
3.1 Estructuras de los filtros digitales de onda de orden dos, tres y cuatro.

Para la obtención del algoritmo se analizarán las estructuras y ecuaciones de los filtros de onda de menor orden, esto con el fin de encontrar similitudes entre ellos que nos permitan establecer las estructuras genéricas para la realización de filtros más complejos.

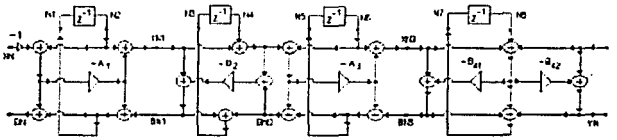
Las estructuras de los filtros digitales de onda paso bajas de segundo, tercer y cuarto orden se pueden observar a continuación:



(a)



(b)



(c)

Figura 3.1: Estructuras de los filtros paso bajas de segundo(a), tercer(b) y cuarto (c) orden.

3.2 Ecuaciones características.

Una vez obtenidas las estructuras el siguiente paso es obtener las ecuaciones que caracterizan a dichos filtros, las cuales nos permitirán llevar a cabo su realización. Para el filtro de segundo orden, y de acuerdo a la estructura observada en la figura 3.1, tenemos:

$$
\begin{aligned}
&XN_1 = XN * A_1 + N_2 - N_2 * A_1 \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&BN_1 = XN_1 - XN_1 * B_{21} - N_4 * B_{21} \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&N_1 = XN * A_1 - N_2 * A_1 + BN_1 \\
&N_3 = XN_1 - XN_1 * B_{22} + N_4 - N_4 * B_{22} + BN_1 \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&YN = -XN_1 * B_{22} - N_4 B_{22} \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&N_2 = N_1 \\
&N_4 = N_3
\end{aligned}
\qquad (3.1)
$$

Basándonos en las estructuras mostradas en la figura 3.1 es posible obtener las ecuaciones que describen a los filtros de tercer y cuarto orden, las cuales se muestran en la figura 3.2

Tercer Orden

$$XN_1 = XN^* A_1 + N_2 - N_2 * A_1$$
$$XN_2 = XN_1 + N_4$$
.....
$$BN_2 = XN_2 - XN_2 * A_{31} + 2 * N_6 - N_6 * A_{31} - N_6 * A_{32}$$
$$BN_1 = XN_1 - B_2 XN_2 - B_2 * XN_2$$
.....
$$N_1 = XN^* A_1 - N_2 * A_1 + BN_1$$
$$N_3 = BN_1 + BN_2$$
$$N_5 = N_6 - N_6 * A_{31} - N_6 * A_{32} - XN_2 * A_{31}$$
.....
$$YN = -XN_1 * A_{31} + 2 * N6 - N_6 * A_{32} - A_{31} * N_6$$
.....
$$N_2 = N_1$$
$$N_4 = N_3$$
$$N_6 = N_5$$

Cuarto Orden.

$$XN_1 = XN * A_1 + N_2 - N_2 * A_1$$
$$XN_2 = XN_1 + N_4$$
$$XN_3 = -XN * A_3 + N_6 - N_6 * A_3$$
.....
$$BN_3 = XN_3 - XN_3 * B_{41} - N_8 * B_{41}$$
$$BN_2 = XN_2 - XN_2 * A_3 + N_6 - N_6 * A_3 + BN_3$$
$$BN_1 = XN_1 - B_2 XN_2 - B_2 * XN_2$$
.....
$$N_1 = XN * A_1 - N_2 * A_1 + BN_1$$
$$N_3 = BN_1 + BN_2$$
$$N_5 = -XN_2 * A_3 - N_6 * A_3 + BN_3$$
$$N_7 = XN_3 - XN3 * B_{41} + N_8 - N_8 * B_{42}$$
.....
$$YN = -XN_1 * B_{41} - * N_8 * B_{42}$$
.....
$$N_2 = N_1$$
$$N_4 = N_3$$
$$N_6 = N_5$$
$$N_8 = N_7$$

Figura 3.2: Ecuaciones de los filtros de tercer y cuarto orden.

Analizando los grupos de ecuaciones se aprecia que algunas se repiten, esto es en cierta manera obvio, ya que si observamos detalladamente las estructuras de los filtros encontraremos que existen componentes que se presentan con cierta periodicidad. En primer lugar, tenemos al adaptador paralelo elemental terminado en uno de sus puertos en un capacitor, el cual podemos encontrar: al inicio de todas las estructuras antes vistas y como tercer elemento del Filtro de Onda de cuarto orden. Luego tenemos al adaptador serie elemental terminado en uno de sus puertos en un inductor, a este elemento lo encontramos como el segundo

componente de los filtros de tercer y cuarto orden. Por último tenemos a las estructuras que se encuentran al final de los diagramas, por un lado tenemos al adaptador serie dependiente con dos multiplicadores terminado en un inductor, el cual se encuentra al final de los filtros de orden par, al final de los filtros de orden impar encontramos a los adaptadores paralelos dependientes con dos multiplicadores terminados en un capacitor.

De acuerdo a lo antes dicho, una primera aproximación sería plantear cada estructura elemental como un objeto independiente y autodefinido; desgraciadamente esto no es posible ya que si observamos las ecuaciones 3.1 y 3.2 nos percatamos que estas deben realizarse en un orden especifico, e intercalando ecuaciones de uno y otro elemento, especialmente para calcular los valores de XN_i y BN_i, lo que imposibilita realizar los componentes independientemente. No obstante esta limitante, si analizamos el orden en que se llevan a cabo las ecuaciones detectaremos que para calcular las XN_i se realiza en orden ascendente pasando del elemento inicial al final y para el cálculo de las BN_i se tiene un orden inverso. Por su parte el cálculo de las N_i no importa en que orden se realice, siempre y cuando se hayan obtenido primero los valores XN_i y Bn_i, el cálculo de YN es igualmente independiente.

Resumiendo, el algoritmo que buscamos debe tener las siguientes características:

- Separar a los Filtros Digitales de Onda en estructuras más simples y diferenciar entre estas.
- Establecer la cantidad y tipo de estructuras elementales que conforman al filtro.
- Ejecutarse siguiendo un orden establecido. Primero ejecutando las ecuaciones para calcular los valores de XN_i, a continuación, en orden descendente ejecutar las ecuaciones para calcular los valores de BN_i y por último obtener los valores de YN y N_i.

Para tener completas las características que debe reunir nuestro algoritmo nos faltaría observar que es lo que ocurre en los filtros paso altas, para ello pensemos en el ejemplo mencionado al inicio de este artículo, si se tuviera un filtro analógico LC paso bajas y se deseará transformar en paso altas, la manera más simple de llevarlo a cabo sería intercalando sus elementos; en un Filtro Digital de Onda la manera de hacerlo es más sencilla aún, ya que los circuitos discretos equivalentes al inductor y al capacitor, sólo difieren en un signo menos, por lo que la estructura digital para el filtro de onda paso altas quedaría como se observa en la figura 3.3. Por lo que la última característica sería:

- Debe identificar el tipo de filtro de que se trate y hacer el cambio de signos pertinente.

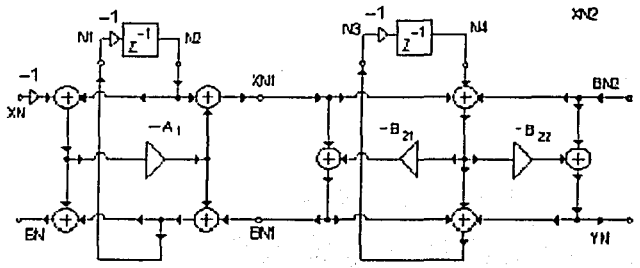


Figura 3.3 Estructura del Filtro Digital de Onda paso altas de segundo orden.

3.3 Propuesta y desarrollo del algoritmo.

Una vez identificadas las características que debe poseer el algoritmo, empezaremos por llevarlas a cabo. Para cumplir con el primer requisito que consiste en la identificación de los tipos de estructuras se propone asignar una bandera la cual nos indique el tipo de estructura de la que estamos hablando, ver tabla 3.1.

Bandera(Bool)	Valor	Adaptador
Es_paralelo	0	Elemental serie
	1	Elemental paralelo
Es_impar	1	Dependiente paralelo
	0	Dependiente serie

Tabla 3.1 Identificadores de los elementos básicos.

Una primera aproximación al algoritmo en pseudocódigo sería:

```
programa: filtro de onda
var Es_paralelo
Si Es_paralelo es igual con 0
Inicio
    hacer  Elemento serie
en caso contrario
    hacer Elemento paralelo
Fin
```

De manera similar es posible implementar la segunda bandera. Para que el algoritmo sea capaz de identificar la cantidad de elementos a emplear y su tipo, basta que observemos dos cosas: primero el número de elementos a emplear es igual al orden del filtro, y segundo, los adaptadores elementales serie y paralelo se intercalan, empezando por el adaptador paralelo. Siguiendo este criterio podemos establecer la siguiente secuencia que determinan la estructura del filtro, el número de la secuencia esta basada en el valor de las banderas antes mencionadas, y esta separada en cada una de las banderas.

Orden	Secuencia(Es_paralelo)	Terminador(Es_par)
2	1	0
3	1,0	1
4	1,0,1	0
5	1,0,1,0	1

Tabla 3.2 Secuencias para la realización de los filtros.

De acuerdo a estas consideraciones el problema se reduce a diseñar un algoritmo capaz de establecer las secuencias antes descritas, una forma de realizarlo sería a través del operador modulo[1]. Posteriormente se almacena la secuencia obtenida en un arreglo, para mediante un ciclo procesarla. Una parte importante que nos falta establecer, es el de cómo llevar a cabo el orden de las ecuaciones, para ello nos valdremos nuevamente de una bandera que nos indique que valor calcular; XN (en orden ascendente) y BN (en orden descendente).

Bandera	Valor	Calcular
Calcular	0	XN
	1	BN

Tabla 3.3. Bandera que establece el orden de las ecuaciones

[1] El operador modulo consiste en llevar a cabo una división entera, teniendo como resultado el residuo de dicha operación.

Reuniendo lo anterior el algoritmo quedaría:

```
Programa: filtro digital de onda
Inicio
var  Orden, Calcular, Es_tipo
arreglo Secuencia
Secuencia = Obtener_Secuencia(Orden)

Calcular = 0
Desde que i igual a 1  hasta que i sea menor que Orden -1, incrementa i
Inicio
  Si Secuencia[i] es igual a 1
   Inicio
     hacer  Elemento_serie(Calcular)
   en caso contrario
     hacer Elemento_paralelo(Calcular)
  Fin
Fin


Obtener_terminador(Orden)
Calcular=1
Desde que i igual a Orden -1 hasta que i sea igual a 1, decrementa i
Inicio
  Si Secuencia[i] es igual a 1
   Inicio
     hacer  Elemento_serie(Calcular)
   en caso contrario
     hacer Elemento_paralelo(Calcular)
  Fin
Fin

calculo de Yn
calculo de los Ni
Si Es_tipo es igual a Paso Altas
Inicio
        Ni=-Ni
Fin
Fin
```

```
Funcion Obtener_Secuencia(Orden)
Inicio
   arreglo secuencia
   Desde que i igual a 1 hasta que i sea igual a Orden
    Inicio
        secuencia[i]= i modulo 2
    Fin
regresar secuencia
Fin

Funcion Elemento_serie(Calcular)
Inicio
  Si Calcular es igual a 0
  Inicio
       obtener XN
  en otro caso
       obtener BN
   Fin
Fin

Funcion Elemento_paralelo(Calcular)
Inicio
  Si Calcular es igual a 0
  Inicio
       obtener XN
  en otro caso
       obtener BN
   Fin
Fin

Funcion Obtener_Terminador()
Inicio
   Si  Orden modulo dos es igual con 0
Inicio
    hacer  Dependiente serie
en caso contrario
    hacer Dependiente paralelo
Fin
```

Básicamente el algoritmo anterior reúne las características obtenidas durante el análisis de las ecuaciones; podemos observar el comportamiento ascendente de este en la primera parte del cálculo, luego realizar el cálculo del terminador y posteriormente realizar el cálculo en orden descendente, por último se calcula el valor de YN y los valores de N_i necesarios para la próxima iteración.

Otro detalle que podemos observar en el código planteado son los datos que son necesarios que el usuario proporcione al programa para que este pueda trabajar adecuadamente. Estos son:

- Orden del filtro.
- Tipo del filtro: paso altas, paso bajas.
- Los coeficientes del filtro, esto es, los valores A_i y B_i que se observan en las estructuras.

El algoritmo planteado en pseudocódigo se implementó mediante C++, partes del código y las gráficas obtenidas se pueden apreciar a continuación.

```
// Funcion para calcular el filtro
void  filtro::getFiltro(void){
        //Definimos las variables intermedias para calcular el filtro
  double BN,XN=1;
  double *X,*B,*N;
  //Definicion de las variables para el calculo de la respuesta en frecuencia
  double inc = 3.1416/(MAX_M*2);
  double *W;

  X = new double[orden-1];
  B = new double[orden-1];
  N = new double[orden];

for(int i=0;i<orden;i++)
        N[i]=0.0;
  //Calculo del filtro
 if(orden == 2){
  for(int i=0;i<MAX_M;i++){
    X[0] = get_EID(-XN,coef[0],N[0],1);
    B[0] = get_FD(X[0],coef[1],coef[2],N[1],1);
     get_FI(X[0],coef[1],coef[2],N[1],&Yn[i],&N[1],1);
     get_EII(-XN,coef[0],N[0],B[0],&BN,&N[0],1);
     XN=0;
     if(tipo == 1){ /*Si es filtro paso altas se invierten los signos*/
         for(int r=0;r<orden;r++)N[r]=-N[r];
      }
    }
  }
  else{/* Se obtiene la secuencia*/
  estructura = get_fil(orden);
    for(int i=0;i<MAX_M;i++){
    X[0] = get_EID(-XN,coef[0],N[0],1);
    for(int k=0;k<orden-2;k++)
    X[k+1] = get_EID(X[k],coef[k+1],N[k+1],estructura[k]);
    B[orden-2] = get_FD(X[orden-2],coef[orden-1],coef[orden],N[orden-1],estructura[orden-2]);
     get_FI(X[orden-2],coef[orden-1],coef[orden],N[orden-1],&Yn[i],&N[orden-1],estructura[orden-2]);
     for(int k=orden-2;k>0;k--)
         get_EII(X[k-1],coef[k],N[k],B[k],&B[k-1],&N[k],estructura[k-1]);
     get_EII(-XN,coef[0],N[0],B[0],&BN,&N[0],1);
     XN=0;
```

```
    if(tipo == 1){ /*SI es filtro paso altas se invierten los signos*/
        for(int r=0;r<orden;r++)N[r]=-N[r];
      }
    }
    }
  // Calculo de la respuesta en frecuencia
        W = new double[MAX_M*2];
  W[0]=0;
  for(int i=1;i<MAX_M*2;i++)W[i]=W[i-1]+inc;
  for(int i=0;i<MAX_M*2;i++)
        Yf[i]=freqz(Yn,W[i],MAX_M);

//Liberamos la memoria empleada
delete [ ] X;
delete [ ] B;
delete [ ] N;
delete [ ] W;
}
```

Tabla 3.4: Código del método getfiltro que implementa el algoritmo obtenido.

En el código de la tabla 3.4 se observan varias funciones, el código de estas puede ser consultado en el apéndice A. La respuesta a un impulso unitario puede observarse en las siguientes gráficas.

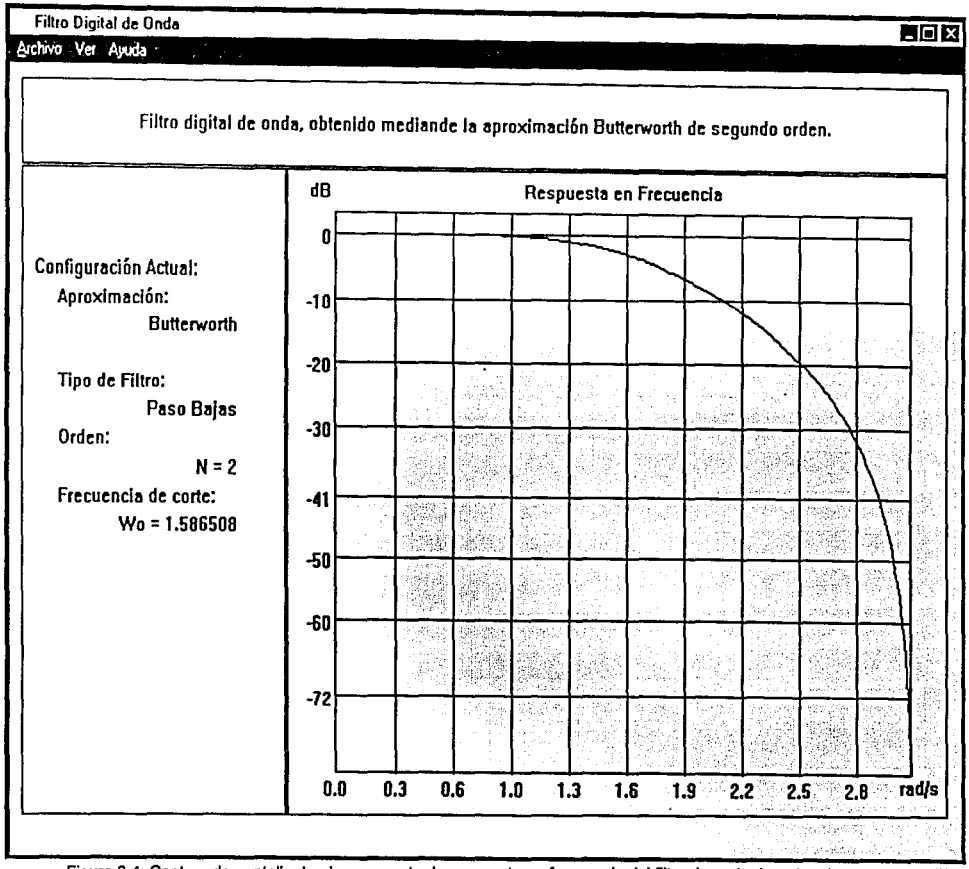


Figura 3.4: Captura de pantalla donde se muestra la respuesta en frecuencia del filtro de onda de orden dos a un impulso unitario.

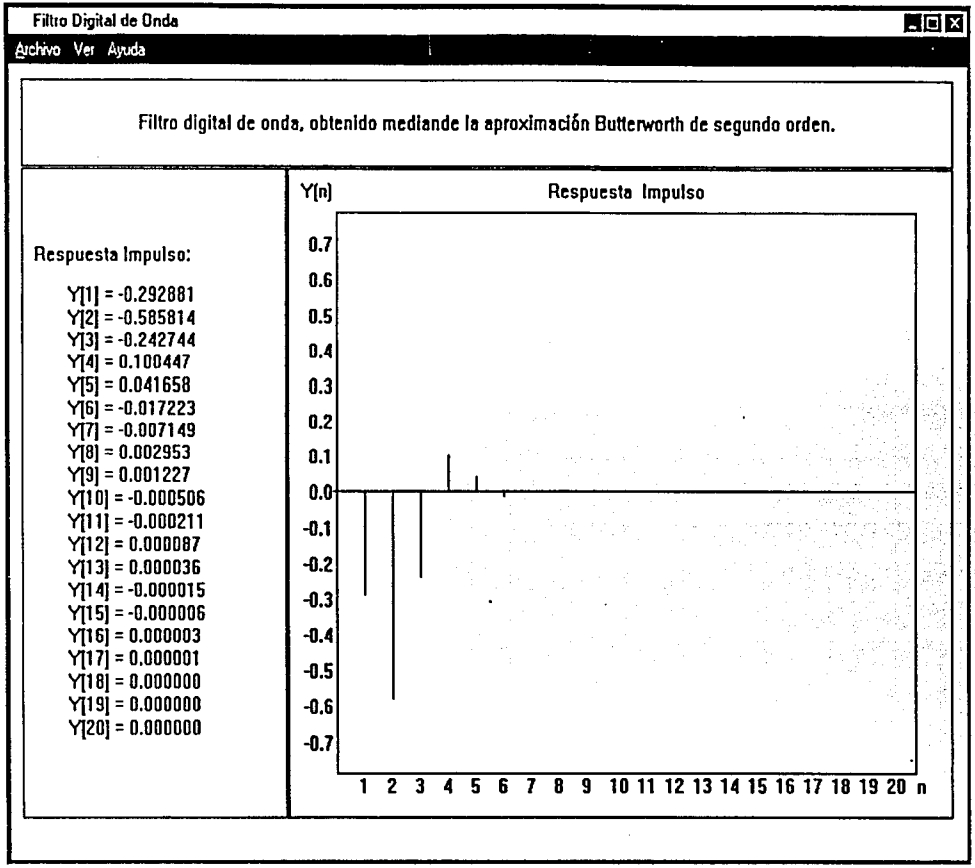
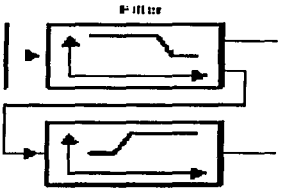


Figura 3.5: Captura de pantalla donde se observa la respuesta impulso del filtro digital de onda de segundo orden.



Capítulo 4

IMPLEMENTACIÓN.

Una vez que se obtuvo el algoritmo que permite realizar los filtros digitales de onda, solo resta llevar a cabo su implementación, para ello se seleccionó al módulo de evaluación del TMS320C30 de Texas Instrument. Se optó por este dispositivo porque, además de las características propias del TMS320C30, cuenta con un puerto ISA, lo que permite conectarlo directamente a una computadora haciendo posible la transmisión de datos en tiempo real, sin tener que volver a cargar el programa dentro del DSP.

En el presente capítulo se discutirá las características fundamentales del módulo que hicieron factible la implementación del algoritmo discutido en el capítulo anterior. También se analizará la manera de cómo la PC se comunica con el DSP para poder realizar los filtros de distintos ordenes y coeficientes.

4.1 Características Generales del TMS320C30

El TMS320C30 es el más reciente miembro de la familia de DSPs TMS320C3x de Texas Instruments. El TMS320C30 es un procesador de 32 bits de punto flotante fabricado con tecnología 0.7-mm triple-level-metal CMOS[1].

El bus de datos interno y el conjunto especial de instrucciones del TMS320C30 poseen la velocidad y flexibilidad para realizar más de 50 Millones de operaciones de punto flotante por segundo (MFLOPS). Este DSP optimiza la velocidad de procesamiento implementando varias de sus funciones en hardware a diferencia de otros procesadores que las realizan mediante software.

El TMS320C30 es capaz de realizar múltiples operaciones en ALU sobre datos enteros o flotantes en un solo ciclo de reloj. Cada procesador cuenta, además con un registro de propósito general, una memoria cache, registros

[1] SPRS032A – APRIL 1996 – REVISED JUNE 1997, Texas Instrument, pp 1-2.

auxiliares, memorias de acceso dual, y un canal DMA que soporta concurrente entrada y salida.

El soporte para lenguajes de alto nivel (lenguaje C) se logra mediante una arquitectura basada en registros, direcciones de memoria más grandes, eficaces modos de direccionamiento, un conjunto de instrucciones flexibles, y una bien soportada aritmética de punto flotante.

Estas características convierten al TMS320C30 en una plataforma de desarrollo para una variedad de aplicaciones de muy diversos tipos, incluyendo el desarrollo de diferentes tipos de filtros, como son filtros adaptivos, FIR, y en el caso que nos ocupa filtros digitales de onda.

Las principales características del TMS320C30 se listan a continuación:[2]

- 60 ns, tiempo de ejecución de una instrucción.
- 33.3 MFLOPS (Millones de operaciones de punto flotante por segundo).
- 16.7 MIPS (Millones de instrucciones por segundo).
- Una ROM (memoria de solo lectura) de 4k x 32 bits.
- Dos RAMs (memoria de acceso aleatorio) de 1k x 32 bits.
- Una memoria cache de 64 x 32 bits.
- Instrucciones y datos de 32 bits.
- Direcciones de 24 bits.
- Ocho registros extendidos (acumuladores).
- Dos generadores de direcciones con ocho registros auxiliares y dos registros para la realización de aritméticas operaciones.
- Un canal de DMA para concurrente I/O.
- Una ALU paralela y múltiples instrucciones en un solo ciclo de reloj.
- Capacidad para repetir bloques enteros de instrucciones.
- Saltos condicionales.
- Dos buses de datos de 32 bits.
- Dos puertos seriales que soportan transmisiones de 8/16/24/32 bits.
- Dos contadores de 32 bits.

4.2 Arquitectura del TMS320C30

La arquitectura del DSP responde a la demanda de los sistemas que están basados en complejos algoritmos y que requieren una solución tanto en hardware como software. La alta eficiencia se alcanza a través de precisión y gran manejo de operaciones de punto flotante, memoria suficiente, alto grado de paralelismo y el empleo de acceso directo a memoria (DMA).

En la figura 4.1 se muestra el diagrama del TMS320C30.

4.2.1 Unidad Central de Procesamiento (CPU).

[2] TMS320C3X, User's Guide, Texas Instruments, pp1-6,1-7.

El TMS320C30 posee una unidad central de procesamiento basada en registros. Las partes que componen al CPU son:

- Multiplicador de punto flotante/entero.
- Unidad aritmética lógica (ALU) para realizar operaciones de punto flotante, entero y operaciones lógicas.
- Un registro de corrimientos de 32 bits.
- Buses internos (CPU1/CPU2 y REG1/REG2).
- Unidades aritméticas auxiliares (ARAUs).
- Registros del CPU.

La figura 4.2 muestra los distintos componentes del CPU, los cuales se describen en las siguientes secciones.

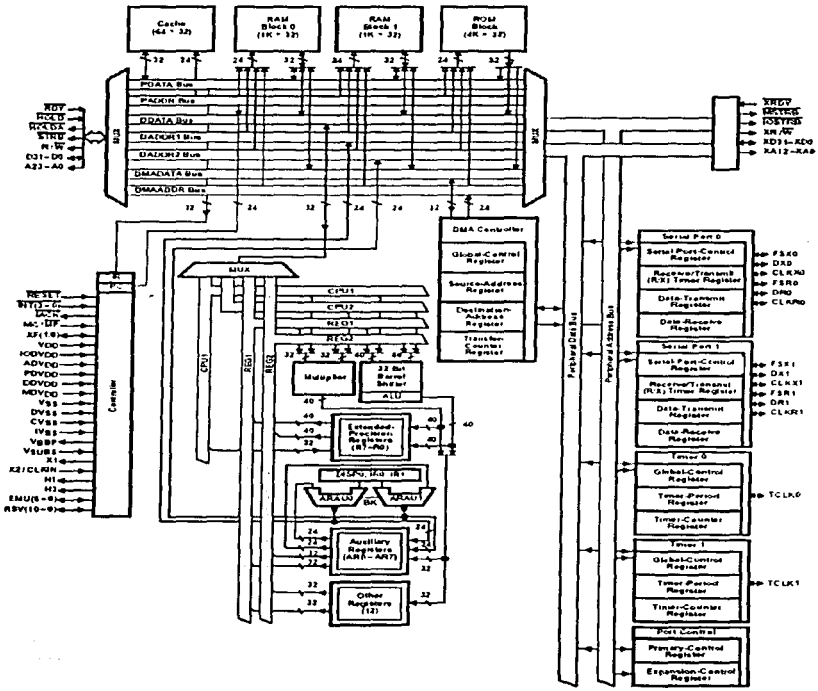


Figura 4.1. Diagrama del TMS320C30.

4.2.1.1 Multiplicador.

Realiza multiplicaciones en un solo ciclo de reloj sobre datos de 24 bits para valores enteros y de 32 bits para punto flotante. La implementación de la aritmética de punto flotante permite llevar a cabo operaciones con instrucciones que requieren 50 ns de tiempo de ejecución, haciendo posible que se puedan realizar múltiples instrucciones en un solo ciclo de reloj de 60 ns.

Cuando el multiplicador realiza multiplicaciones de punto flotante, la entrada consta de valores de 32 bits y el resultado es un numero de 40 bits. Cuando se realiza una multiplicación con datos enteros, a la entrada se tiene un dato de 24 bits y el resultado es de 32 bits.

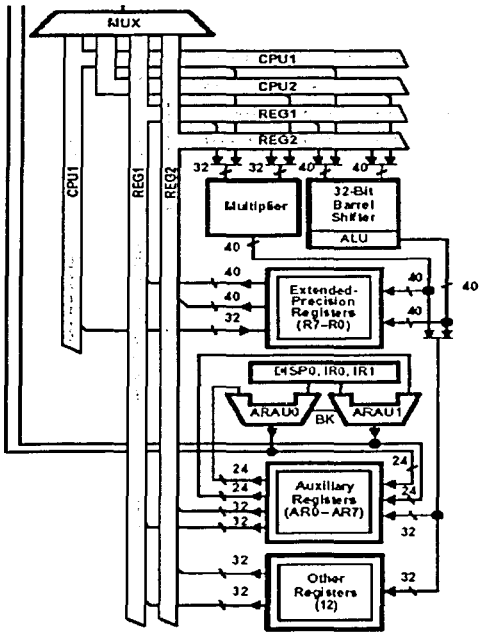


Figura 4.2. Componentes del CPU.

4.2.1.2 Unidad Aritmética Lógica (ALU).

La ALU realiza operaciones en un ciclo de reloj con datos enteros de 32 bits, lógicos de 32 bits, de punto flotante de 40 bits. Los resultados de la ALU siempre son de 32 bits para enteros y de 40 bits para fracciones. El registro de

corrimiento es empleado para desplazar 32 bits a la izquierda o a la derecha en un solo ciclo de reloj.

Los buses internos, CPU1/CP2 y REG1/REG2, tienen acceso a dos operadores en la memoria y dos operadores en los registros, respectivamente, esto nos permite realizar simultáneamente multiplicaciones y sumas o restas sobre cuatro operadores.

4.2.1.3 Unidades aritméticas auxiliares (ARAUs)

Dos unidades aritméticas auxiliares (ARAU0 y ARAU1) son capaces de generar dos direcciones de memoria en un solo ciclo de reloj. Las ARAUs operan simultáneamente con el multiplicador y la ALU. Estas soportan direccionamiento mediante desplazamiento, indexación de los registros (IR0 y IR1), direccionamiento circular y direccionamiento inverso.

4.2.1.4 Registros del CPU.

El TMS320C30 cuenta con 28 registros. Todos estos registros pueden ser operados por el multiplicador y la ALU y pueden ser usados como registros de propósito general. De cualquier forma, los registros poseen funciones específicas. Por ejemplo, los ocho registros de extendidos están diseñados para almacenar los resultados de las operaciones de punto flotante. Los ocho registros auxiliares soportan distintos modos de direccionamiento y pueden ser usados como registros de propósito general para datos de 32 bits, enteros o lógicos. El resto de los registros proveen lo mismo funciones del sistema como direccionamiento, manejo del stack, estado de los procesos, interrupciones y repetición de bloques de código.

El nombre de los registros y sus funciones asignadas se pueden observar en la tabla 4.1.

Los **registros extendidos (R0-R8)** son capaces de almacenar y realizar operaciones con números de 32 bits, para enteros, y 40 bits, para fraccionarios. Cualquier instrucción que requiera números de punto flotante, empleará los bits 39-0. Si los operadores son enteros con signo o sin signo, solamente los bits 31-0 serán usados; los bits 39-32 permanecerán sin cambios. Esto es valido para todas las operaciones.

Los **registros auxiliares (AR7-AR0)** pueden ser accesados por el CPU y modificados mediante las dos ARAUs. La principal función de estos registros es generar direcciones de 24 bits. También pueden emplearse como contadores o como registros de 32 bits de propósito general que a su vez pueden ser manipulados a través del multiplicador y la ALU.

El **apuntador a la página de datos (DP)** es un registro de 32 bits. Los ocho bits menos significativos del DP los emplea el modo de direccionamiento

directo[3] como un apuntador al dato de inicio de la página. Las páginas de datos poseen una longitud de 64k palabras de 32 bits, haciendo un total de 256 páginas.

Los **registros índices (IR0 y IR1)** almacenan los valores usados por las ARAUs para calcular una dirección a través del índice.Las ARAUs emplean el **tamaño del bloque** en direccionamiento circular para especificar el tamaño del bloque de datos.

El **apuntador al stack del sistema (SP)** es un registro de 32 bits que contiene la última dirección del stack del sistema. El SP siempre apunta al último elemento colocado dentro del stack. Agregar un elemento al stack produce un incremento del SP; retirar un elemento produce en decremento. El SP es manipulado por las interrupciones, saltos condicionales, y las instrucciones PUSH y POP.

El **registro de estados (ST)** contiene información general relacionada al estado del CPU. Las operaciones generalmente modifican alguna de las banderas del registro de estado de acuerdo a sí el resultado fue 0, negativo, etc. Esto incluye tanto a la carga de registros y operaciones de almacenamiento, como funciones aritméticas y lógicas.

El **registro para habilitar interrupciones CPU/DMA (IE)** es un registro de 32 bits. Los bits para 10-0 habilitan las interrupciones del CPU, mientras que los bits 26-16 son para las interrupciones del DMA. Un 1 habilita la interrupción correspondiente. Un 0 deshabilita dicha interrupción. Este registro se tratará más a detalle cuando se vea el uso del canal DMA y el manejo de interrupciones.

El registro de las **banderas de las interrupciones (IF)** es también un registro de 32 bits. Un 1 en determinado bit indica que esta interrupción a ocurrido. Un 0 indica que la interrupción no se ha llevado a cabo.

El **registro de banderas de entrada salida** controla el funcionamiento de dedicados puertos externos, XF0 y XF1. Estos puertos pueden ser configurados como entradas o salidas, por lo que se puede leer o escribir en ellos según sea el caso.

El **contador de repeticiones (RC)** es un registro de 32 bits que se emplea para especificar el número de veces que se debe repetir un bloque de código. Cuando el procesador esta operando en modo de repetición, el registro de **dirección de inicio del bloque (RS)** contiene la dirección de inicio del bloque de programa a repetir, por su parte el registro de **dirección final del bloque (RE)** contiene la última dirección del bloque.

[3] Ver el apéndice B donde se discuten los distintos tipos de direccionamiento.

Nombre del Registro	Función
R0	Registro extendido 0.
R1	Registro extendido 1.
R2	Registro extendido 2.
R3	Registro extendido 3.
R4	Registro extendido 4.
R5	Registro extendido 5.
R6	Registro extendido 6.
R7	Registro extendido 7.
AR0	Registro auxiliar 0.
AR1	Registro auxiliar 1.
AR2	Registro auxiliar 2.
AR3	Registro auxiliar 3.
AR4	Registro auxiliar 4.
AR5	Registro auxiliar 5.
AR6	Registro auxiliar 6.
AR7	Registro auxiliar 7.
DP	Apuntador a la página de datos.
IR0	Registro índice 0.
IR1	Registro índice 1.
BK	Tamaño del bloque.
SP	Apuntador al stack del sistema.
ST	Registro de estados.
IE	Habilitador de interrupciones CPU/DMA.
IF	Banderas de las interrupciones CPU.
IOF	Banderas de entrada y salida.
RS	Dirección de inicio del bloque a repetir.
RE	Dirección final del bloque a repetir.
RC	Número de veces que se debe repetir el bloque.

Tabla 4.1. Registros del CPU.

El **contador del programa (PC)** es un registro de 32 bits que contiene la dirección de la próxima instrucción a ser ejecutada. Cabe aclarar que este registro no forma parte de los registros del CPU, sólo puede ser modificado por las instrucciones que controlan la secuencia del programa, como son los saltos condicionales o los manejadores de interrupciones.

4.2.2 Organización de la memoria.

El espacio total de memoria del TMS320C30 es de 16M (millones) de palabras de 32 bits. Los programas, datos y registros de entrada y salida están dentro de este espacio de direcciones, esto permite que las tablas, coeficientes, código de programa, o datos puedan ser almacenados tanto en la RAM con en la ROM.

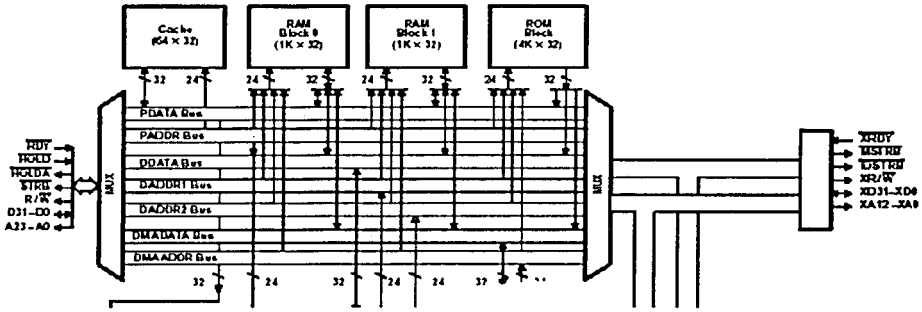


Figura 4.3 Organización de la memoria.

4.2.2.1 RAM, ROM y Cache

La figura 4.3 muestra como la memoria se organiza en el TMS320C30. Los bloques RAM 0 y 1 son de 1k x 32 bits cada uno. El bloque de ROM es de 4k x 32 bits. Ambos bloques (RAM y ROM) son capaces de soportar dos accesos del CPU en un solo ciclo. El contar con buses separados para los datos, código de programa y DMA, permite realizar lectura y escritura de datos, y operaciones con el DMA de forma simultanea. Por ejemplo: el CPU puede acceder a dos datos almacenados en un bloque de la RAM y realizar una instrucción del programa, al mimo tiempo que el chip de DMA almacena otro dato en la RAM, todo en un solo ciclo de reloj.

Además se cuenta con una memoria cache de 64 x 32 bits, esto con el fin de almacenar las partes del código que más se repiten, esto reduce en gran medida el número de veces que se tienen que acceder las instrucciones, además de que liberas los buses para que puedan ser usados por la DMA, u otro dispositivos en el sistema.

4.2.2.2 Mapas de memoria.

El mapa de memoria depende del modo en el que este trabajando el procesador, en modo microprocesador (MC/MP o MCBL/MP = 0) o en modo

microcontrolador (MC/MP o MCBL/MP = 1). No obstante los mapas de memoria para estos modos son similares (ver la figura 4.4). Las localidades del 800000h – 801FFFh están asignadas al bus de expansión. Cuando esta sección es accedida, el MSTRB se activa. Las localidades 802000h-803FFFh son reservadas. Las localidades del 804000h – 805FFFh están asignadas al bus de expansión. Cuando esta sección es accedida, el IOSTRB se activa. Las localidades 806000h-807FFFh son reservadas. A todos los periféricos se les ha asignado las direcciones de 808000h-809FFFh. En ambos modos, el bloque 0 de RAM esta localizado a partir de la dirección 809800h a la 809BFFh, a su vez el bloque 1 se encuentra de la dirección 809C00h a la 809FFFh. Los localidades 80A000h-0FFFFFFh son accedidas por el puerto de memoria externa (STRB activado).

En modo microprocesador, los 4k de memoria ROM no se encuentran asignados dentro del mapa de memoria del TMS320C30. Las localidades 0h-0BFh contienen al vector de interrupciones, al vector de interrupciones por software, y direcciones reservadas, todas ellas son accedidas a través del puerto de memoria externa 8STRB activada). Las direcciones 0C0h-7FFFFFh son accedidas por el mismo puerto.

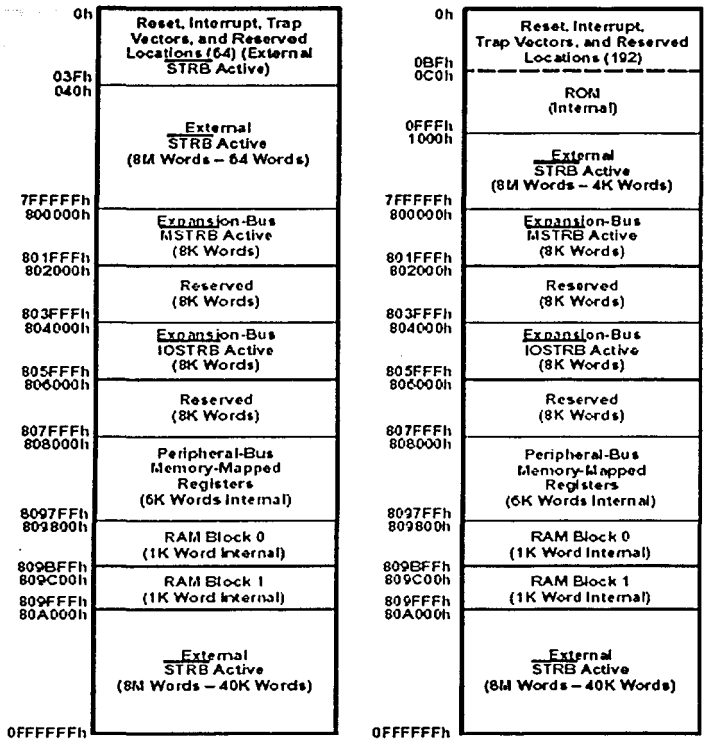
En modo microcontrolador, los 4k de memoria ROM se encuentran asignados dentro del mapa de memoria del TMS32C30. Existen 192 direcciones(0h-0BFh) para el vector de interrupciones, el vector de interrupciones por software, y espacio reservado. Las localidades 1000h-7FFFFFh son accedidas a través del puerto de memoria externa (STRB activado). Ambos mapas se pueden apreciar en la figura 4.4.

4.2.2.3 Modos de acceso a la memoria.

El TMS320C30 es capaz de soportar cinco grupos de formas diferentes de acceder a la memoria.[4] Existen seis tipos de direccionamiento que pueden utilizarse dentro de estos grupos, tal y como se lista a continuación.

- Modo de direccionamiento general.
  - A través de registros. El operador del CPU es un registro.
  - Inmediato corto. El operador es un dato de 16 bits.
  - Directo. El operador se encuentra especificado a través de su dirección de memoria.
  - Indirecto. Un registro auxiliar determina la dirección del operador.

[4] Consultar el apéndice para obtener información más detallada acerca de las distintas formas de direccionamiento.



(a) Microprocesador (b) Microcontrolador

Figura 4.4. Mapas de Memoria del TMS320C30

- Modo de direccionamiento de tres operadores.
  - A través del registro. Igual que en el modo de direccionamiento general.
  - Indirecto. Igual que en el modo de direccionamiento general.
- Modo de direccionamiento paralelo.
  - A través de registros. El operador es un registro extendido.
  - Indirecto. Igual que en el modo de direccionamiento general.
- Modo de direccionamiento inmediato largo.

En este modo el operador es un dato de 24 bits.

- Modo de direccionamiento a través de saltos condicionales.
  - A través del registro. Igual que en el modo de direccionamiento general.
  - PC relativo. Un desplazamiento con signo de 16 bits es agregado al PC.

4.2.3 Operación del bus interno.

La mayor parte del eficaz funcionamiento del TMS320C30 se debe a la manera en que trabaja el bus interno y al alto grado de paralelismo. Al contar con buses separados de programa(PADDR y PDATA), de datos (DADDR1, DADDR2 Y DDATA), y de DMA (DMAADDR y DMADATA) permite alcanzar altos grados de paralelismo. Estos buses conectan a todos los componentes físicos (circuitos de memoria, circuitos de procesamiento, y periféricos) soportados por el DSP. La figura 4.3 muestras estos internos buses y sus conexiones.

El PC esta conectado al bus de programa de 24 bits (PADDR). El registro de instrucciones (IR) esta conectado al bus de datos de 32 bits (PDATA). Estos buses pueden obtener una palabra de instrucción cada ciclo de reloj.

Los buses de direcciones de datos de 24 bits (DADDR1 y DADDR2) y el bus de datos de 32 bits (DDATA) son capaces de obtener dos datos cada ciclo de reloj. El bus DDATA carga datos al CPU a través de los buses CPU1 y CPU2. Los buses CPU1 y CPU2 proporcionan dos operadores para el multiplicador, la ALU y los registros cada ciclo de reloj. De la misma manera los registros dentro del CPU REG1 y REG2, proporcionan igual número de datos para el multiplicador y la ALU. En la figura 4.2 se pueden observar estos buses.

El canal de DMA cuenta con un bus de 24 bits para las direcciones (DMAADDR) y un bus de 32 bits para los datos (DMADATA). Estos buses permite que el DMA acceda a una dirección de memoria simultáneamente con la obtención de datos y código del programa por parte del CPU.

4.2.4 Operación del bus externo.

El bus externo cuenta con un área de direccionamiento de 13 bits y al igual que el bus interno puede emplearse para cargar datos de programa o como espacio de entrada y salida.

4.2.5 Interrupciones.

Este DSP soporta cuatro interrupciones externas (INT3-INT0), varias interrupciones internas, y una señal externa de RESET. Estas se emplean para interrumpir al DMA o al CPU. Cuando el CPU responde alguna interrupción, el

pin IACK puede emplearse como una señal externa para una confirmación de la interrupción.

4.2.6 Periféricos.

Todos los dispositivos del TMS320C30 se controlan mediante registros asignados en memoria y un bus dedicado. Este bus esta compuesto por un canal de datos de 32 bits y un canal de direcciones de 24 bits. Los dispositivos de este DSP incluyen dos contadores y dos puertos seriales. La figura 4.5 muestra los distintos periféricos con sus buses y señales.

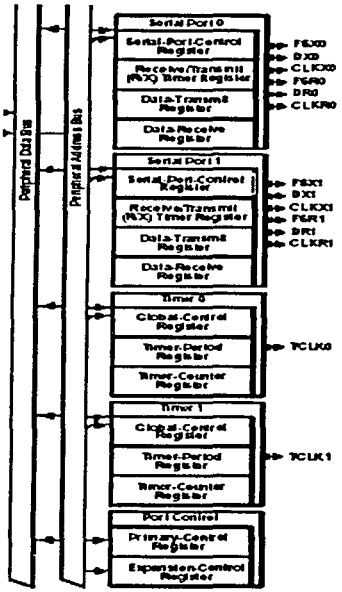


Figura 4.5. Periféricos.

4.2.6.1 Contadores.

Los dos contadores de 32 bits son de propósito general. Cada contador cuenta con un pin de entrada/salida que se puede emplear tanto, como para obtener una señal de reloj, como para generarla; también se puede utilizar como un pin de datos.

4.2.6.2 Puertos Seriales.

Los dos puertos series bidireccionales son totalmente independientes. Cada puerto puede ser configurado para transferir palabras de datos de 8, 16, 24 o 32 bits. La señal de reloj de cada puerto puede ser generada interna o externamente.

4.2.7 Acceso Directo a Memoria (DMA, Direct Memory Access).

El chip de DMA puede leer o escribir a cualquier localidad de memoria sin interferir con la operación del CPU. El canal de DMA cuanta con sus propios generadores de direcciones, registros fuente y destinos, y contador de transferencias. Dedicados canales de direcciones y de datos para el DMA minimizan los posibles conflictos con el CPU. La operación del DMA consiste en la transferencia de un bloque o una única palabra de una región de memoria a otra. La figura 4.6 muestra al canal DMA con sus buses asociados. Ver la figura 4.1 donde se muestra al canal DMA con sus buses asociados.

4.3 Obtención de datos en el TMS320C30.

Una parte importante al desarrollar una aplicación en tiempo real es la manera de cómo se adquieren los datos, para esto el TMS320C30 cuanta con un completo sistema de conversión analógico-digital y digital-analógico, en un solo chip. El TLC32044[5] (AIC, analog interface circuit) integra un filtro paso bandas, capacitores variables, filtros antialiasing, convertidores A/D y D/A con una resolución de 14 bits, filtro paso bajas, filtros de reconstrucción, todo en un chip CMOS. Esto lo podemos observar en la figura 4.6.

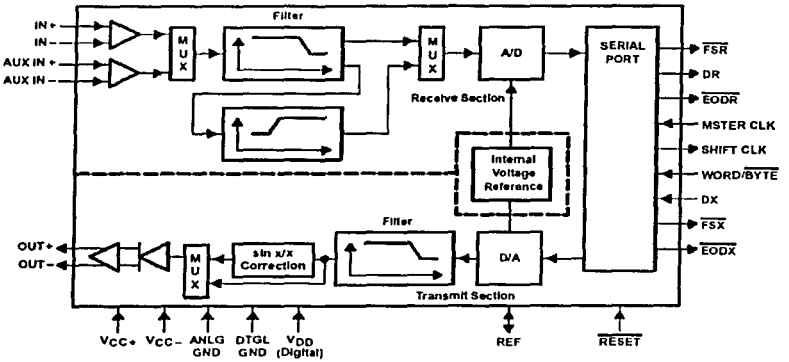


Figura 4.6. Bloque funcional del TLC32044.

Cuando se accede al AIC por medio de un puerto serie del TMS320C30, no se requiere ningún otro circuito. La manera que estos dos dispositivos se

[5] SLAS017F - MARCH 1988 - REVISED MAY 1995

encuentran conectados se puede ver en la figura 4.7. El puerto serial y las señales de control se encuentran conectados directamente, la señal maestra del reloj del AIC es controlada por el TCLK0, una de las salidas de los contadores internos del DSP. La entrada WORD/BYTE del AIC, se encuentra en alto para seleccionar palabras de 16 bits para el puerto de transferencia. El pin XF0, configurado como salida, del DSP esta conectado al pin de reinicio (RST) del AIC, esto permite que el AIC pueda ser reiniciado a través del TMS320C30. Logrando además que el contador y el puerto serial puedan ser configurados antes de cualquier conversión.

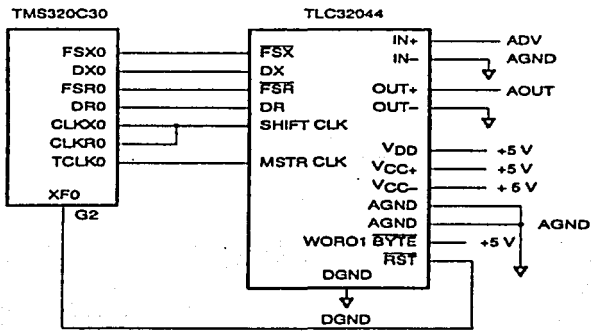


Figura 4.7. Conexión entre el puerto serie y el TLC32044.

Para lograr una adecuada comunicación con el AIC, el puerto serie del TMS320C30 debe estar correctamente configurado. Primero, se debe reiniciar al puerto colocando un 2170300h en el registro de control. Al mismo tiempo el AIC debe ser reinicializado, más adelante se muestra como se puede llevar a cabo esto. Durante el reinicio del puerto se deben configurar distintos parámetros de este como son, polaridad a emplear, longitud de la palabra de transferencia, habilitar las interrupciones, entre otras cosas. Los registros del control y transmisión del puerto también deben ser configurados. Para esta aplicación se debe colocar un 111h en ambos registros, lo cual establece a todos los pines en modo de entrada y salida. Cuando todas estas operaciones se han realizado, las interrupciones deben ser activadas, y el vector de interrupciones debe estar correctamente cargado, la transmisión del puerto serial iniciará en cuanto el puerto salga del reset.

Para iniciar las operaciones de conversión dentro del AIC y la transferencia de datos con el puerto serie, primero es necesario reiniciar el AIC para ello se debe mandar a tierra al registro XF0, para hacer esto se debe colocar un 2 en el registro IOF. Una vez que el puerto serie y el contador han sido adecuadamente configurados, se debe mandar a alto al registro XF0 para

ello se coloca un 6 en IOF. Esto le indica al AIC que inicie a trabajar de acuerdo a su configuración preestablecida, esto es, todos los filtros internos están activados, la frecuencia de muestreo se fija aproximadamente a 6.4 kHz, y la transmisión y recepción de datos trabaja en forma sincrona.

Para configurar el AIC de manera diferente, se debe enviar primero una palabra de datos con sus dos bits menos significados iguales a 1. Esta palabra le indica al AIC que la próxima transmisión del puerto serie no contiene un dato sino información de control. Esta información la emplea el AIC para cargar distintos registros y especificar opciones de configuración interna. Existen cuatro tipos de información de control.

A continuación se presenta el código empleado para llevar a cabo la programación adecuada para hacer trabajar al AIC junto con el primer puerto. También se muestra el código de la interrupción empleada por el puerto serie que permite, además de obtener las distintas muestras, se lleve a cabo la configuración del AIC por medio de las palabras de control[6].

Función para configurar al AIC.

```
void init_aic(void)
{
  volatile int i;
  /*-------------------------------------------------------*/
  /* Establecemos  la confugaración del AIC de la siguiente forma:
  /*  1. Frecuencia de muestreo a  8 kHz y el filtro antialiasing a los 3.6 kHz. */
  /*  2. Se desactiva el filtro paso bajas                                        */
  /*  3. Se sincroniza transmisión y recepción.                                   */
  /*  4. Se activa el filtro rectificador sinx/x                                  */
  /*  5. Se coloca el voltaje de referencia a 1.5 V                               */
  /*-------------------------------------------------------*/
  aic_command_0.command  = 0;      /*Indicamos que es la palabra de código 0    */
  aic_command_0.ra       = 9;      /* Se establece la frecuencia de muestreo a 8 kHz */
  aic_command_0.ta       = 9;      /*    y el filtro antialiasing a 3.6 kHz.*/
  aic_command_1.command  = 1;      /* Se establece la configuración preestablecida  */
  aic_command_1.ra_prime = 1;              /*para la palabra de código 1*/
  aic_command_1.ta_prime = 1;
  aic_command_1.d_f      = 0;
 aic_command_2.command   = 2; /*Se establece la configuración preestablecida /
 aic_command_2.rb        = 33;                    /*para la palabra de código 2*/

  aic_command_2.tb       = 33;
  aic_command_3.command  = 3;
  aic_command_3.highpass = OFF;     /* Se desactiva el filtro paso bajas    */
  aic_command_3.loopback = OFF;     /* Se desactiva el loopback             */
  aic_command_3.aux      = OFF;     /* Se desactiva el registro auxiliar    */
  aic_command_3.sync     = ON;      /* Se sincronizan la transmisión y la recepción */
  aic_command_3.gain     = LINE_V; /* Se establece el nivel de referencia   */
  aic_command_3.sinx     = ON;      /* Se activa al filtro rectificador */
```

[6] Las siguientes funciones están basadas en los ejemplos proporcionados parra el TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT PROGRAMS (C) 1990 TEXAS INSTRUMENTS, HOUSTON.

```
/*-------------------------------------------------------*/
 /* Configuración del contador 0                         */
 /* El contador se configura de la siguiente forma       */
 /*      1. El contador es reinicado y activado          */
 /*  2. El contador se establece en modo pulso           */
 /*  3. El periodo del contador se establece a dos instrucciones por ciclo */
 /*-------------------------------------------------------*/
 timer[0][PERIOD] = 0x1;
 timer[0][GLOBAL] = 0x2C1;
 /*-------------------------------------------------------*/
 /*Configuración del puerto serie 0                      */
 /*      1. FSK,FSRM,CLKX,CLKR, configurados como externos. */
 /*      2. Frecuencia de recepción y transmisión variable. */
 /*      3. Handshake desactivado. */
 /*      4. Fin de transmisión en alto. */
 /*      5. Se activan en bajo FSX, FSR. */
 /*      6. Palabras de transmisión y recepción de 16 bits. */
 /*      7. Se activa la interrupción de transmisión. */
 /*      8. Se activa la interrupción por recepción.de dato. */
 /*      9. FSX, FSR, CLKX, CLKR, DX, DR configurados como */
 /*        pines del puerto.                              */
 /*-------------------------------------------------------*/
 serial_port[0][X_PORT] = 0x111;
 serial_port[0][R_PORT] = 0x111;

 asm(" LDI 2,IOF");        /* Reinicio del AIC colocando XF0 a bajo */

 for(i = 0; i < 50; i++);   /* Se conserva la señal en bajo por un periodo de tiempo*/
 serial_port[0][GLOBAL] = 0x0e970300; /* Se escribe la configuración para el puerto
                                         serie */
 serial_port[0][X_DATA] = 0x0;  /* Se inicializa el puerto los datos del puerto serie */
 asm(" LDI 6,IOF");             /*Se coloca al AIC fuera del reset */
 asm(" LDI    0,IF"); /* Se inicializan todas las banderas de las interrupciones */
 asm(" LDI  410h,IE"); /* Se activan las interrupciones para DMA y puerto serie */
 asm(" OR  2000h,ST"); /* Se activan las interrupciones globales */
 /*-------------------------------------------------------*/
 /*Se modifica la configuración del AIC por medio de las palabras de código. */
 /*-------------------------------------------------------*/
 configure_aic(*((int *) &aic_command_0));
 configure_aic(*((int *) &aic_command_3));
}
```

Función empleada por la interrupción del puerto serie, activada cada vez que se tiene un dato nuevo.

```
/*=========================================================================*/
/* C_INT05()                                               */
//*     1. Si secondary_transmit es igual a uno se transmite una palabra de código. */
/*      2. Si send_command es igual a uno se indica al AIC que en la próxima transmisión será
una pa labra de código. */
/*      3. En otro caso se escribe el dato de salida y se lee el de entrada. */
/*=========================================================================*/
```

```
void c_int05(void)
{
if (secondary_transmit)
{
serial_port[0][X_DATA] = aic_secondary;
secondary_transmit    = OFF;
}
else if (send_command)
{
serial_port[0][X_DATA] = 3;
secondary_transmit    = ON;
send_command          = OFF;
}
else
{
serial_port[0][X_DATA] = output << 2;
input = serial_port[0][R_DATA] << 16 >> 18;
}
}
```

4.4 Host y DMA.

Hasta el momento hemos visto como es posible obtener las muestras a partir de una señal analógica y también como se puede regresar un elemento procesado a la salida. En esta sección veremos como es que el usuario podrá enviar los coeficientes y valores necesarios para seleccionar el filtro que desee implementar y como esto se hace sin tener que reiniciar al DSP.

Como se recordará del capítulo anterior existen una serie de valores que el usuario debe proporcionar al programa como argumentos para que este pueda realizar los cálculos, estos datos son:

- Orden del filtro.
- Tipo del filtro: paso altas, paso bajas.
- Los coeficientes del filtro, esto es, los valores A_i y B_i que se observan en las estructuras.

Para pasar estos datos al DSP se contó con la facilidad de que se decidió implementar dicho algoritmo en un modulo de evaluación el cual cuenta con un bus ISA que se conecta directamente a la computadora, a la cual a partir de este momento la conoceremos con el nombre de host. Así que aprovechando esta facilidad se desarrollo un programa que se encarga de solicitar los valores al usuario y posteriormente enviarlos al puerto donde se encuentra conectada la tarjeta, a la tarjeta se le indica por medio de una interrupción que existe una nueva configuración y que se prepare para recibir los datos. Veremos primeramente el programa empleado para solicitar los datos y posteriormente veremos las funciones empleadas por el módulo para recibirlos y sincronizar la transferencia todo empleando DMA.

Por cuestiones prácticas el problema se dividió en dos partes: primero se creo un programa sencillo que se encarga de crear un archivo con los datos del filtro de tal manera que se puede reutilizar más tarde y otro programa que

recibe como argumento los valores del filtro y se encarga de llevar a cabo la transferencia con el microcontrolador, posteriormente desde una interfaz gráfica de usuario se mandan a llamar a estos programas mediante funciones asignadas a los métodos de la interfaz, todo esto quedará más claro una vez se analice el código.

Programa crea_wdf empleado para crear un archivo con los valores proporcionados por el usuario.

```
/*****************************************************************
REALIZADO POR: Miguel Angel Palomera Perez
FECHA DE REALIZACION: 29/05/2002
DESCRIPCION: crea_wdf.c contiene las estructuras y funciones necesarias
para crear un archivo de acceso aleatorio para almacenar las características
de un filtro digital de onda
*****************************************************************/

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include "crea_wdf.h" /*Cabecera que define las estructuras a emplear */

int main(void){
  FILE *out;
  char filename[256],*ext=".wdf";
  wdf_file new_filter;
  int i,c;
  float aux;

 /*Se solicitan los valores */
 clrscr();
 printf("\t\t }}}}} Filtro digital de onda {{{{{"
       "\n\n  Introduzca los valores del nuevo filtro.\n"
       "\nNombre del archivo *[.wdf]?  ");
 scanf("%s",filename);

 if(!findext(filename,ext))  /*Corroboramos que la extensión del archivo sea .wdf */
    strcat(filename,ext);

/*Se obtiene el resto de los valores */
 do{
        do{
                printf("\nTipo de aproximacion, (0)Butterworth"
                                        "\n                (1)Chevychev  ? ");
                scanf("%d",&new_filter.aprox);
        }while(!(new_filter.aprox == 0 || new_filter.aprox == 1));
        do{
                printf("Tipo de filtro, (0)Paso Bajas"
                                        "\n             (1)Paso Altas  ? ");
                scanf("%d",&new_filter.tipo);
        }while(!(new_filter.tipo == 0 || new_filter.tipo == 1));
        do{
                printf("Orden del filtro (2-9) ? ");
        scanf("%d",&new_filter.orden);
        }while(!(new_filter.orden > 1 && new_filter.orden < 10));
        printf("Introduzca los valores de los coeficientes:\n");
```

```
            for(i=0;i<=new_filter.orden;i++){
                    printf("coeficiente[%d] = ",i+1);
            scanf("%f",&aux);
          new_filter.coef[i]=(double)aux;
            }
            printf("\n\t\t Los datos introducidos son correctos ? (s/n): ");
      c=getche();
    }while(!(c == 115 || c == 83) );
    if((out=fopen(filename,"w")) == NULL){
            printf("\nNo es posible crear el archivo %s",filename);
      return 1;
    }

    /*Se almacenan los valores en el archivo especificado*/
    fwrite(&new_filter,sizeof(new_filter),1,out);
    fclose(out);
    printf("\n\t\t %s ha sido creado.",filename);
    getch();
            return 0;
  }


  /*Función empleada para verificar que la extensión del archivo sea la correcta*/
   int findext(char *str1,char *ext){
            int x,y;
    char *ptr;
    ptr=strchr(str1,ext[0]);
    if(ptr){
            for(x=1;x<=strlen(ext);x++)
      {
       y=ptr-str1;
       ptr=strchr(&str1[y],ext[x]);
       if(!ptr)
            return 0;
      }
    }
    else
            return 0;
  return 1;
  }

   /*****************************************************************
  REALIZADO POR: Miguel Angel Palomera Perez
  FECHA DE REALIZACION: 29/05/2002
  DESCRIPCION: crea_wdf.h contiene las estructuras y funciones necesarias
  para crear un archivo de acceso aleatorio para almacenar las características
  de un filtro digital de onda.
  *****************************************************************/

  typedef struct{
  int aprox; /* tipo de aproximacion: O Butterworth, 1 Chevichev*/
  int orden; /* orden del filtro */
  int tipo;  /* tipo de filtro: 0 paso bajas, 1 paso altas;*/
  double coef[10];
  }wdf_file;

  /*Funcion para encontrar si la cadena 1 tiene la extencion*/
  int findext(char *str1, char *ext);
```

El programa crea_wdf es realmente muy sencillo lo único que hace es pedir el nombre del archivo donde se almacenaran los valores, dicho archivo deberá tener la extensión wdf, en caso contrario enviara un mensaje de error. A continuación se piden los valores del nuevo filtro y por último se almacenan en el archivo. Para almacenarlos se emplea una estructura la cual nos facilitará su posterior consulta.

Programa car_wdf empleado para transferir los valores del nuevo filtro al módulo de evaluación.

```
/*************************************************************************
 REALIZADO POR: Miguel Angel Palomera Perez
 FECHA DE REALIZACION: 25/06/2002
 FECHA DE ULTIMA MODIFICACION: 25/06/2002
 DESCRIPCION: carga_wdf.c se encargara de la comunicación con
        el modulo de evaluación, además de establecer la configuración
  actual.

*************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <bios.h>
#include <dos.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include "car_wdf.h"
#include "wf_cmd.h"

/*Variables Globales*/
int iobase = IOBASE_DEFAULT;
int orden,tipo;
double coef[10];

void main(int argc,char *argv[]){
   char *str;
   int i;
   /*Inicializamos el DSP*/
   init_evm();  /* iniciamos la comunicación con el módulo */

   split(str,argv[1],','); /* obtenemos los datos a partir de los argumentos pasados al
                              programa */
   orden=(int)strnum(str);
   split(str,argv[1],',');
   tipo=(int)strnum(str);
   for(i=0;i<=orden;i++){
         split(str,argv[1],',');
    coef[i]=strnum(str);
   }
   /*Enviamos configuracion*/
   send_conf(NUEVO);
}
```

```
/*Funciones varias para procesar el flujo de entrada*/

void split(char *out, char *in,char separador){
        int i,j;
  for(i=0;in[i] && (in[i] != separador);i++)
        out[i] = in[i];
  out[i] = '\0';
  if(in[i])
        ++i;
  for(j=0; in[j];)
        in[j++]=in[i++];
}

/*Función empleada para convertir un caracter a un entero */
int car_val(char c){
        if(c > 47 && c < 58) return c-48;
  if(c > 64 && c < 91) return c-55;
  if(c > 96 && c < 123) return c-87;
        return 0;
}

/*Función empleada para convertir un valor entero a un caracter  */
char val_car(int val){
        if(val < 10)return val+48;
  else return val+55;
}

/*Función empleada para convertir una cadena a su valor numérico */
double strnum(char *s){
  int len,*aux,i=0,j=0,exp;
  double ent=0,frac=0;
  len=strlen(s);
  aux=(int *)calloc(len+1,sizeof(int));
  for(i=0;i<len;i++){
    if(s[i] != 46)aux[i]=car_val(s[i]);
    if(s[i] == 46)aux[i]=10;
  }
  aux[i]=10;
  while(aux[j++]<10);
  exp=j-2;
  j=j-2;
  for(i=0;i<=exp;i++)ent=ent+aux[i]*pow(10,j--);
  j=1;
  while(aux[exp+1+j++]<10);
  j=j-2;
  i=exp+1+j;
  for(;i>exp+1;i--)frac=frac+aux[i]*pow(10,-j--);
  return ent+frac;
}
```

```
/*=====================================================================*/
/* Las siguientes funciones están basadas en los programa de demostración que */
/* acompañan al  módulo de evaluación*/
/* INIT_EVM(): Inicialización del EVM  (C) 1990 TEXAS INSTRUMENTS, HOUSTON*/
/*=====================================================================*/
void init_evm(void)
{
   time_t start,finish;
   double delta;
   get_iobase();          /* Se obtiene la dirección donde se encuentra colocado el
                                 módulo          */
   time(&start); delta = 0.0;
   UPDATE_STATUSO; /*Se envía un mensaje al módulo */
   while(!IS_READ_ACK)  /*Se espera por la respuesta del módulo */
   {
         time(&finish);
         delta = difftime(finish,start);
         if(delta > 3.0){
         printf("Error al establecer la comunicacion con el modulo");
         exit(-1);}
         UPDATE_STATUSO;
   }
   CLR_READ_ACK; /* se le indica al módulo que el mensaje fue recibido*/
   WRITE_DATA(NONE);       /* Se prepara para la transmisión      */
}
/*=====================================================================*/
/* GET_IOBASE(): Obtiene la dirección del puerto donde se encuentra el  módulo
TMS320C30 */ /* EVM  (C) 1990 TEXAS INSTRUMENTS, HOUSTON
*/
/*=====================================================================*/
void get_iobase(void)
{
   char *d_options  = getenv("D_OPTIONS");  /* Se consulta la variable de ambiente
                                                D_OPTIONS  */
   int  found_base  = OFF;              /* Se busca la opción -p*/
   if (!d_options) return;       /* Si no se encuentra la variable D_OPTIONS se emplea el
                                    puerto por omisión */
   while (*++d_options)                /* Se inicia la búsqueda    */
   {

      if (d_options[-1] == '-' && toupper(d_options[0]) == 'P')
      {
         found_base = ON;
         break;
      }
   }
   if (!found_base) return;     /* Si no se encuentra –p se emplea el puerto por omisión */
   sscanf(++d_options, "%x", &iobase);/* En otro case se lee la dirección      */
   switch(iobase)
   {
      /*-------------------------------------------------------*/
      /* Se regresan sólo valores validos para el puerto          */
      /*-------------------------------------------------------*/
      case 0x240:
      case 0x280:
      case 0x320:
      case 0x340:
```

```
            break;
        /*-------------------------------------------------------*/
        /* Si se encuentra un valor no valido se envía un mensaje de error  y el programa
           termina    */
        /*-------------------------------------------------------*/
        default:
            puts("Valor no valido para el puerto encontrado en");
            puts("la variable de ambiente D_OPTIONS.");
            puts("Saliendo...");
            exit(EXIT_FAILURE);
    }
}

/*Función empleada para enviar la nueva configuración al módulo */
void send_comando(int i){
  while(READ_CMD); /* Se espera hasta que el DSP este listo para leer un nuevo
                      comando */
  WRITE_CMD(i);   /* Se le indica al DSP de que comando se trata */
  while(READ_CMD != i); /*Se espera hasta que el DSP confirme que ha leído el dato*/
  WRITE_CMD(NONE); /* Se le indica al DSP que se recibió la confirmación */
}

/* Función empleada para enviar la nueva configuración */
void send_conf(char command){
  int i=0;
  send_comando(command);  /* Se le indica al filtro que se requiere una nueva
                             configuración */
  send_comando(orden);  /* Se envía el orden y el tipo  */
  send_comando(tipo+1);
  while(READ_CMD); /*Esperamos para que el DSP este listo para recibir los
                     coeficientes */
  WRITE_CMD(TRANS); /* Esperamos para que este listo el canal DMA*/
  while(READ_CMD != TRANS);
  CLR_WRITE_ACK;
  for(i=0;i<=orden;i++){    /*Enviamos los valores empleando el canal DMA */
              WRITE_DATA((int)(coef[i]*10000));
    do{
        UPDATE_STATUSO;
      }
        while(!IS_WRITE_ACK); /* Esperamos para que el DSP lea el dato */
    CLR_WRITE_ACK;
  }
  WRITE_CMD(NONE); /*Notificamos fin de transmision*/
  send_comando(COMPLET);/*Indicamos al canal DMA que la transmisión fue
                          completa */
}
```

```
/*****************************************************************
REALIZADO POR: Miguel Angel Palomera Perez
 FECHA DE REALIZACION: 22/12/2001
 FECHA DE ULTIMA MODIFICACION: 22/12/2001
 DESCRIPCION:
        carga_wdf.h  Contiene la declaración de las funciones empleadas
        por carga_wdf.c además de varios macros importantes

        Esta cabecera esta basada en:
        PC_1.H
  TMS320C30 EVALUATION MODULE PC HOST SUPPORT FILE
        :MACROS, STRUCTURES, FUNCTION PROTOTYPES
  (C) 1990 TEXAS INSTRUMENTS, HOUSTON

*****************************************************************/
/* Definición de nuevos tipos */
#define OFF     0x00
#define ON      0x01
#define NONE    0x00

/*Declaración de las funciones */
void init_evm(void);
void get_iobase(void);
void send_comando(int i);
void send_conf(char command);
void split(char *out,char *in,char separador);
int car_val(char c);
char val_car(int val);
double strnum(char *s);

/*DEFINICION DE LOS MACROS PARA COMUNICACION ENTRE EL HOST Y EL */
/*MODULO
*/
/*-------------------------------------------------------------*/
/* SE DEFINE EL PUERTO POR OMISICIÓN                           */
/*-------------------------------------------------------------*/
#define IOBASE_DEFAULT 0x0240
/*A continuación se definen los registros que empleados por el módulo para la
 comunicación con el host */
#define CONTROL5        iobase + 0x000A    /* REGISTRO DE CONTROL     */
#define STATUS0         iobase + 0x0400    /* REGISTRO DE ESTADOS     */
#define COM_CMD         iobase + 0x0800    /* REGISTRO DE COMANDOS    */
#define COM_DATA        iobase + 0x0808    /* REGISTRO DE DATOS*/
#define SOFT_RESET      iobase + 0x0818    /* LOCALIZACION DEL RESET POR
                                              SOFTWARE   */

#define MINOR_CMD       iobase + 0x0014    /* REGISTRO DE COMANDO*/

/*-------------------------------------------------------------*/
/* MASCARAS PARA INDICAR QUE CONFIRMAR QUE LOS DATOS HAN SIDO
LEIDOS O ESCRITOS  */
/*-------------------------------------------------------------*/
#define MREAD_ACK       0x0002
#define MWRITE_ACK      0x0004
#define UPDATE_REQ      0x6044

#define WRITE_CMD(x)  outp(COM_CMD,x)  /*ESCRIBE UN COMANDO DE 8 BITs */
```

```
#define READ_CMD          inp   (COM_CMD)      /* LEE UN COMANDO DE 8 BITs */
#define WRITE_DATA(x) outport(COM_DATA, x)  /* ESCRIBE UN DATO DE 16 BITs*/
#define READ_DATA          inport (COM_DATA)     /* LEE UN DATO DE 16 BITs*/
#define RESET_EVM          outport(SOFT_RESET, 0) /* RESET AL EVM */
#define RESET_C30          outport(CONTROL5,0x808)/* PONE AL C30 EN RESET */
#define RUN_C30            outport(CONTROL5,0x800)/* COLOCA AL C30 FUERA DEL
                                                     RESET*/
#define READ_STATUS0 inport (STATUS0)  /* SE LEE EL ESTATUS DEL PUERTO */
#define WRITE_MINOR_CMD(x) outport(MINOR_CMD,x)  /* SE ESCRIBE EN EL
                                                     REGISTO DE ESTATUS   */
/*-------------------------------------------------------------*/
/* MACROS PARA CHECAR QUE LOS DATOS  HAYAN
   SIDO LEIDO O ESCRITRO CORRECTAMENTE                          */
/*-------------------------------------------------------------*/
#define IS_READ_ACK     (READ_STATUS0 & MREAD_ACK)
#define IS_WRITE_ACK    (READ_STATUS0 & MWRITE_ACK)
/*-------------------------------------------------------------*/
/* SE LIMPIAN LOS BITS DEL REGISTRO DE ESTADO  */
/*-------------------------------------------------------------*/
#define CLR_READ_ACK    WRITE_MINOR_CMD(MREAD_ACK)
#define CLR_WRITE_ACK   WRITE_MINOR_CMD(MWRITE_ACK)
#define UPDATE_STATUS0  WRITE_MINOR_CMD(UPDATE_REQ)
```

El programa car_wdf emplea varios macros para comunicarse con el módulo por lo que analizaremos a estos primero. La parte fuerte de la comunicación entre el host y el DSP se basan en varias funciones disponibles en el lenguaje C para interactuar directamente con el hardware, estas funciones son:

Función	Sintaxis	Descripción
Outp	#include <conio.h> int outp(unsigned portid, int value);	outp es un macro que escribe el menos significativo byte de value al puerto especificado por portid
Outport	#include <dos.h> void outport(int portid, int value);	outport trabaja de manera similar a outp, la parte baja del entero (el byte menos significativo) se envía al puerto especificado por portid la parte alta del entero se envía a la dirección portid +1
Inp	#include <conio.h> int inp(unsigned portid);	inp es un macro que lee un byte del puerto especificado por portid
Inport	#include <dos.h> int inport(int portid);	Inport trabaja de manera similar a inp, lee un byte del puerto especificado por portid, y lee un byte de la dirección portid +1

Tabla 4.2 Funciones empleadas para la comunicación con el módulo

Como observamos en la tabla las funciones descritas son más que suficientes para llevar a cabo la comunicación con el DSP, ya que como esta conectado directamente al host por un bus ISA sólo tenemos que enviar los datos a la dirección donde se encuentra y leerlos del mismo puerto. En el archivo de cabecera carga_wdf.h se definen varios macros que emplean estas funciones, la razón de emplear los macros es hacer un poco más legible al programa. Por ejemplo, se define al macro READ_CMD que será sustituido por el compilador por la instrucción inp(COM_CMD), de manera similar ocurre con el resto de los macros.

Ahora analizaremos al programa carga_wdf.c. Primero declaramos las variables globales como el puerto del DSP (iobase). A continuación empleamos la función init_evm(), la cual esta definida al final del archivo. Esta función se encarga de obtener la dirección del puerto donde se encuentra el módulo mediante la función get_iobase() y verifica que el DSP realmente se encuentre conectado a ese puerto, para ello envía una señal de petición de actualización del estado y espera la confirmación de lectura del dato, si esta se realiza antes de tres segundos se considera que la dirección es válida y por lo tanto se esta lista para transmitir datos al DSP, si por el contrario no se responde se envia un mensaje de error y se termina la ejecución del programa. Enseguida se leen los valores pasados al programa mediante la función split(), el primer argumento de esta función es un apuntador a caracter en el cual se regresa la cadena de salida, el segundo argumento es una cadena que contiene los valores que deseamos separar. Es necesario emplear esta función debido a que en C los argumentos pasados a los programas por la línea de comandos se almacenan en forma de cadena en la variable argv[], el formato de los datos es de la siguiente manera: orden,tipo,coef1,coef2..... A continuación de separar los campos se hace una conversión de tipo, de cadena a número, para poder trabajar con ello. Por último se envían los datos al microcontrolador mediante la función send_conf().

La función send_conf() tiene como argumento el comando a transmitir en este caso NUEVO. Lo primero que hace esta función, mediante la instrucción send_command(), es enviar una notificación al DSP sobre el comando que debe procesar, para que realice las tareas necesarias para recibir los datos que se le enviarán, haciendo uso del mismo comando se envía el orden y el tipo del filtro. Posteriormente se espera para que el canal DMA se configure apropiadamente y enseguida empieza la transmisión de los coeficientes.

En estos momentos es posible que nos hayan surgido algunas dudas como son: ¿Cómo se configura el canal de DMA?, ¿Cuántos comandos procesa el DSP?, ¿Cómo se prepara para recibir los datos?, etc. Todo ira quedando más claro conforme se analice las tareas que lleva a cabo el DSP. Empezaremos por conocer un poco más a detalle al canal de DMA.

El TMS320C30 cuenta con un canal de DMA capaz de acceder a cualquier localidad de memoria del DSP, incluyendo los registros asignados a los periféricos, para realizar operaciones de lectura y escritura. Una

transferencia DMA consiste en dos operaciones: leer de una localidad de memoria y escribir a otra localidad. Las operaciones del DMA están controladas por medio de cuatro registros:

- El registro de control global
- El registro de la dirección origen.
- El registro de la dirección destino.
- El registro para el contador de transferencia.

El **registro de control global** controla el estado en el que el DMA trabaja. Este registro también indica el estado del DMA que cambia cada ciclo de reloj. Las direcciones origen y destino pueden ser incremenetadas, decrementadas o sincronizadas usando bits específicos de este registro. Los **registros de dirección origen y destino** son registros de 24 bits que contienen la dirección de origen y de destino de los datos, respectivamente. De acuerdo al contenido de los bits DESRC, INSRC, DECDST y INDST del registro de control, estos registros son incrementados o decrementados al final del correspondiente acceso a memoria, es decir, el registro origen después de una lectura y el registro destino después de una escritura. El registro **contador de transferencia** es un registro de 24 bits, que cuenta de manera decreciente. El contador se decrementa al inicio de la operación de escritura. De esta manera se puede controlar el tamaño del bloque a transmitir. Cuando el bit TCINT del registro de control tiene un uno, el contador va generar una interrupción cuando llegue a cero.

Con estos conocimientos en la mente podemos hecharle un vistazo al código empleado para llevar a cabo la configuración del DMA.

```
/*-------------------------------------------------------------*/
/*Dirección del Registro de control global del canal DMA     */
/*-------------------------------------------------------------*/
volatile int *dma = (volatile int *) 0x808000;
/*-------------------------------------------------------------*/
/* Dirección donde se localiza el registro para la            */
/*     comunicación con el host.                              */
/*-------------------------------------------------------------*/
volatile int *host = (volatile int *) 0x804000;

/*-------------------------------------------------------------*/
/*Dirección de los registros de control del canal DMA        */
/*Para obtener la dirección completa al registro de control  */
/*global se le agrega la siguiente dirección                 */
/*-------------------------------------------------------------*/
#define SOURCE   4              /* Dirección origen    */
#define DEST     6              /* Dirección destino */
#define TRANSFER 8              /* Contador de Transferencia   */
```

```
/*-------------------------------------------------------------*/
/*Función empleada para configurar el canal DMA                */
/*-------------------------------------------------------------*/
void config_dma(int *dest,int comand, int n){
  asm(" AND 0FFF8h,IF"); /*Limpiamos cualquier interrupción pendiente del HOST*/
  asm(" OR  @_dma_int1,IE");/*Sincronizamos HOST con DMA*/
  dma[SOURCE]=(int)host;  /*La dirección origen es el puerto del host*/
  dma[DEST] =(int)dest;   /*La dirección destino se pasa como argumento a la función*/
  dma[TRANSFER]= n;     /*El número de datos a transmitir es también un argumento */
  dma[GLOBAL]=0x0D43;
  *host=comand; /*Confirmacion de la configuracion del DMA,listo para recibir*/
}
```

La función config_dma() recibe tres argumentos. El primero es un apuntador a la dirección donde se almacenan los datos, el segundo es un entero (comand) se emplea para confirmar que la configuración del DMA se llevó a cabo correctamente[7], y el último argumento, entero también, (n) indica el número de datos a transmitir. Esta función lee los datos colocados por el host en el puerto del módulo mediante la función send_conf(), y los escribe en una dirección de memoria accesible por el CPU. El CPU, de acuerdo a estos nuevos valores, llevará a cabo los cálculos para realizar el filtro solicitado.

Hasta el momento se han cubierto dos partes importantes del diseño del filtro: la adquisición de nuevas muestras y la obtención de los datos proporcionados por el usuario para especificar el filtro deseado; sólo nos resta ver como estas muestras son procesadas de acuerdo a los valores dados por el usuario. Esto se verá en la siguiente sección.

4.5 Programa principal.

Realmente el algoritmo para procesar los datos ya lo desarrollamos en el capítulo tres, por lo que sólo nos concretaremos a analizar el código del programa que lo implementa en el DSP.

```
**********************************************************************
REALIZADO POR: Miguel Angel Palomera Perez
FECHA DE REALIZACION: 21/01/2002
FECHA DE ULTIMA MODIFICACION: 21/01/2002
DESCRIPCION:
        wf_c30.c programa principal que realiza los filtros
        en el modulo de evaluación.
**********************************************************************/

#include "c30.h"
#include "wf_cmd.h"
#include "wf_c30.h"

void main(void){
```

[7] La forma de llevar a cabo la confirmación de transmisión recibida es realmente muy sencilla, el host transmite un valor entero (un comando) y se queda en espera hasta que el DSP regrese el mismo entero. Si el DSP no reenvía este valor se presume que ha ocurrido un error y el programa en el host termina.

```
  int i=0;
  init_evm(); /*Inicializamos al módulo*/
  init_aic();  /*Se inicializa el AIC*/
  init_host(); /* Se inicializa el puerto del host */
  *orden=DEFAULT_ORDEN;  /*Se establece un filtro por omisión de orden dos paso bajas*/
  *tipo=0;
  for(i=0;i<=DEFAULT_ORDEN;i++){ /*Inicialización de las variables */
        N[i]=0;
        B[i]=0;
  }
  /*Establecemos los coeficientes para N=2*/
  coef[0]=0.4142;
  coef[1]=0.2928;
  coef[2]=0.7071;
  for(;;){ /*Se crea un ciclo infinito. */
  procesa_comando(); /*Función empleada para obtener la nueva configuración*/
  if(!NEW)get_filtro(); /*Si existe una nueva configuración no se procesa la muestra hasta que
                          la nueva configuración se halla completado. */

  }
}

/*-------------------------------------------------------------------*/
/*Función empleada para obtener una nueva configuración    */
/*-------------------------------------------------------------------*/

void procesa_comando(void){
  int i,j=0;
  while(dma[TRANSFER]); /* Esperamos para que cualquier transferencia DMA se complete */
  i=*host & 0xFF; /*Leemos por si existe un nuevo comando */
  if(i ){ /*Verificamos si existe un nuevo comando */
       *host=i; /*Avisamos al host que se recibió el comando */
       switch(i){ /*Procesamos el comando. */
        case NUEVO:
                    while(*host & 0xFF);
                  *host = NONE;
                    while(!(*host & 0xFF)); /*Obtenemos el orden del filtro. */
                    *orden = *host & 0xFF;
                    *host = *orden;
                    while(*host & 0xFF);
                  *host = NONE;
                    while(*host & 0xFF);
                  *host = NONE;
                     while(!(*host & 0xFF));  /*Obtenemos el tipo */
                    *tipo = *host & 0xFF;
                    *host = *tipo;
                    *tipo = *tipo - 1;
                    NEW=1; /*Le indicamos al programa principal que existen nuevos datos */
                    while(*host & 0xFF);
                   *host = NONE;
                  break;
```

```
        case TRANS:
                config_dma(buffer,TRANS,orden[0]+1);  /*Configuramos el canal DMA*/
                 if(orden[0]>2)get_fil(*orden); /*Obtenemos la estructura del filtro */
                 for(j=0;j<=*orden;j++){ /*Limpiamos las variables intermedias */
                   N[j]=0;
                   B[j]=0;
                  }
                 while(dma[TRANSFER]); /*Esperamos que se complete la transmisión */
                  for(j=0;j<=*orden;j++) /*Ajustmos los coeficientes a los valores originales */
                coef[j]=(float)(buffer[j]*0.0001);
                 break;

        case COMPLET:          /*Confirmamos que la transmisión se completó */
                *host=i;
                 while(*host & 0xFF);
                *host = NONE;
                 NEW=0; /*Indicamos al programa principal que la nueva configuración se
                        realizó satisfactoriamente */
                 break;

      }

  }

/*-------------------------------------------------------------*/
/*Función empleada para procesar las muestras          */
/*-------------------------------------------------------------*/
void get_filtro(void){
  int i,r;
  asm(" IDLE");  /*Esperamos por una nueva muestra*/
  asm(" AND 1FFFh,ST"); /*Se desactivan las interrupciones para procesar la muestra */
  *x = input;  /*Se asigna a la variable x el valor de la nueva muestra */
  get_parI(-*x,0);
  switch(*orden){
     case 2:
      {
       get_parI(X[0],1);
       get_finparI(1);
       get_finparYN();
       get_parD(-*x,0);
       if(*tipo) /*Si es filtro paso altas se invierten los signos*/
       for(r=0;r<*orden;r++)N[r]=-N[r];
      break;
     }

   default:
    {
     for(i=0;i<*orden-2;i++){
      if(filtro[i] == -1 ) get_imparI(i+1);
      else get_parI(X[i],i+1);
      if(filtro[i] == -1 ){
        get_finimparI(*orden - 2);
        get_finimparYN(*orden - 1);
         }
      else{
        get_finparI(*orden-1);
        get_finparYN();
```

```
        }
      }
      for(i=*orden-3;i>=0;i--){
          if(filtro[i] == -1 ) get_imparD(i+1);
          else get_parD(X[i],i+1);
      }
      get_parD(-*x,0);
      if(*tipo) /*Si es filtro paso altas se invierten los signos */
      for(r=0;r<*orden;r++)N[r]=-N[r];
      break;
    }
  }
  output=*y; /*Colocamos el valor procesado y en la variable output para que el puerto serie la
procese */
  asm(" OR  2000h,ST"); /*Activamos las interrupciones para obtener otra muestra */
}

/*-------------------------------------------------------------*/
/*Funciones varias                                             */
/*-------------------------------------------------------------*/

/*-------------------------------------------------------------*/
/*Función empleada para obtener la potencia de un número       */
/*-------------------------------------------------------------*/


int pot(int i,int p){
        if(p==0)
                return 1;
        else
                return i*pot(i,p-1);
}

/*-------------------------------------------------------------*/
/*Función empleada para obtener la estructura del filtro       */
/*Si recordamos el capítulo tres la estructura del filtro esta */
/*formada por una secuencia de unos y ceros que indican las    */
/*ecuaciones a emplear.                                        */
/*-------------------------------------------------------------*/


void get_fil(int N){
        int i=0;
        for(i=0;i<N-2;i++){
                filtro[i]=pot(-1,i+1);
        }
        if( N%2 == 0)
                filtro[N-2]=1;
        else
                filtro[N-2]=-1;

}
```

```
/*-------------------------------------------------------------------*/
/*Las ecuaciones que vienen a continuación  realizan las */
/*ecuaciones que describen a los filtros, tal como se      */
/*obtuvieron en el capítulo anterior.                      */
/*-------------------------------------------------------------------*/


void get_parI(float z,int ind){ /*XN*/
  X[ind]=-(z+N[ind])*coef[ind]+N[ind];
}

void get_finparI(int ind){  /*BN*/
        B[ind]=X[ind-1]-(X[ind-1]+N[ind])*coef[ind];
}

void get_finparYN(void){  /*Y y N*/
        *y=-(X[*orden-2]+N[*orden-1])*coef[*orden];
  N[*orden-1]=(X[*orden-2]+B[*orden-1]+N[*orden-1])-(X[*orden-2]+N[*orden-1])*coef[*orden];
}

void get_parD(float z,int ind){ /*N y BN*/
  B[ind] = (z+B[ind+1]+N[ind])-(z+N[ind])*coef[ind];
        N[ind] = -(z+N[ind])*coef[ind]+B[ind+1];
}

void get_imparI(int ind){
        X[ind]=X[ind-1]+N[ind];
}


void get_imparD(int ind){ /*BN y N*/
  B[ind] = X[ind-1]-(X[ind]+B[ind+1])*coef[ind];
  N[ind]=B[ind]+B[ind+1];
}

void get_finimparI(int ind){ /*orden -2*/
        B[ind+1]= X[ind]-(X[ind]+N[ind+1])*coef[*orden-1]+(2-coef[*orden])*N[ind+1];
}


void get_finimparYN(int ind){ /* orden -1*/
        *y=-(X[ind-1]+N[ind])*coef[ind]+(2-coef[ind +1])*N[ind];
  N[ind]=N[ind]-(X[ind-1]+N[ind])*coef[ind]-N[ind]*coef[ind+1];
}
```

A continuación se presentan algunas funciones que se encuentran definidas en el archivo c30.c, pero que son empleadas por el programa que estamos analizando tal es el caso de init_evm(). Algunas otras funciones, tales como init_aic(), no se presentan debido a que ya fueran analizadas en la sección correspondiente.

```
/*=========================================================================*/
/* INIT_EVM(): Configuración del modulo de evaluación.                     */
/*=========================================================================*/
void init_evm(void)
{
   bus[EXPANSION] = 0x0;          /* Ceros al bus de expansión  */
   bus[PRIMARY]   = 0x0;          /* Ceros al bus primario */

   asm(" OR 800h,ST");            /* Se activa la memoria cache del DSP  */
}

/*=========================================================================*/
/*INIT_HOST(): Configuración del host                                      */
/*=========================================================================*/
void init_host(void)
{
   *host = NONE;                  /* Se limpia el puerto del host  */
}
```

Por último se muestra al archivo de cabecera wf_c30.h. En este archivo se definen, entre otras cosas, las direcciones de memoria donde se almacenarán los valores proporcionados por el usuario.

```
/*************************************************************************
 REALIZADO POR: Miguel Angel Palomera Perez
 FECHA DE REALIZACION: 21/01/2002
 FECHA DE ULTIMA MODIFICACION: 21/01/2002
 DESCRIPCION:
        wf_c30.h definicion de variable globales y funciones
        para el wf_c30.c
*************************************************************************/
/*Funciones varias */
void main(void);
void procesa_comando(void);
void get_filtro(void);
int pot(int i,int p);
void get_fil(int N);
void get_parI(float z,int ind);
void get_finparI(int ind);
void get_finparYN(void);
void get_parD(float z,int ind);


/*Variables  Globales locales */
int output=0;                  /* Muestra de entrada  */
int input=0;                   /* Muestra de salida  */
volatile int NEW=0;            /* Bandera para saber si existe una nueva configuración */
/*Variables globales definidas en c301.h*/
/*-------------------------------------------------------------------------*/
/* Variables de control para el AIC                                        */
/*-------------------------------------------------------------------------*/
extern AIC_COMMAND_0  aic_command_0;
extern AIC_COMMAND_1  aic_command_1;
extern AIC_COMMAND_2  aic_command_2;
extern AIC_COMMAND_3  aic_command_3;
extern volatile int send_command ;      /*Bandera para enviar una palabra de control al AIC */
```

```
extern volatile int secondary_transmit; /* Bandera para iniciar la transmisión de la palabra de
control. */
extern int aic_secondary;        /* Comando a transmitir. */
/*************************************************************************/
/*      Localidades de los registros de control para el TMS320C30                */
/*************************************************************************/
/*-----------------------------------------------------------------------*/
/* Localización del puerto serie                                         */
/*-----------------------------------------------------------------------*/
extern volatile int (*serial_port)[16];
/*-----------------------------------------------------------------------*/
/*Localización del registro de control del canal DMA                     */
/*-----------------------------------------------------------------------*/
extern volatile int *dma;
/*-----------------------------------------------------------------------*/
/* Registro de comunicación con el host                                  */
/*-----------------------------------------------------------------------*/
extern volatile int *host;

*-----------------------------------------------------------------------*/
/*Direcciones donde se almacenaran los datos pasados por el host*/
/*-----------------------------------------------------------------------*/
int (*orden) = (int *) 0x809c00;
int (*tipo) = (int *)  0x809c01;
int (*buffer) = (int *) 0x809c02; /*Dirección empleada por el DMA para almacenar los valores
                                    transmitidos por el host. */
float (*coef) = (float *) 0x809c0c; /*Coeficientes */
float (*B) = (float *) 0x809c2c; /*Valores intermedios */
volatile float (*N) = (volatile float *) 0x809c17;  /*Valores de los nodos intermedios */
volatile float (*X) = (volatile float *) 0x809c21; /*Valores de las salidas intermedias */
volatile int (*filtro) = (volatile int *) 0x809c36;
float (*x) = (float *) 0x809c40;
float (*y) = (float *) 0x809c41;
```

Básicamente el programa principal realiza dos operaciones: revisa si existe una nueva configuración y lleva a cabo las actividades correspondientes a esta; y procesa las muestras obtenidas por el puerto serie mediante el AIC. La primera actividad se lleva a cabo mediante la función procesa_comando(). Dicha función primero lee el registro *host por si existe algún nuevo comando, si este existe, este puede tomar uno de los siguientes tres valores NUEVO, TRANS, COMPLET. Si el comando es igual a NUEVO entonces se llevan a cabo las siguientes tareas: se obtiene el orden y el tipo del filtro y se le indica al programa principal que se esta llevando a cabo una nueva configuración para que este no procese la siguiente muestra ya que podría darse un error, esto se hace mediante la bandera NEW igual a 1. Si el comando es igual a TRANS se configura el canal DMA mediante la función config_dma(), vista anteriormente, se obtiene la estructura del nuevo filtro, se limpian las variables intermedias y se obtienen los coeficientes. Si el comando es COMPLET, este indica que el host a terminado de enviar los datos y esta en espera que el DSP termine, por lo que lo único que se hace es indicarle al host que la configuración ha terminado, para ello se reenvié el comando al host y se le indica al programa principal que la nueva configuración se ha realizado con éxito, para ello se establece la bandera NEW a cero.

Por su parte la segunda actividad, procesar las muestras, se realiza mediante la función get_filtro(). Esta función es una realización directa del algoritmo discutido en el capítulo anterior. Las ecuaciones que describen a los filtros digitales se llevan a cabo por funciones separadas que son llamadas por get_filtro(), de acuerdo a la estructura del filtro (secuencia de unos y ceros que nos indican el tipo de adaptador a realizar). Se remite al lector al capítulo tres para una mayor comprensión de esta función.

4.6 Características Adicionales.

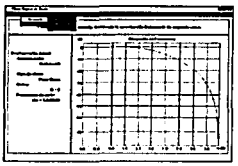
A la interfaz empleada por el host, además de las funciones empleadas para transmitir los coeficientes al host, se le agregaron funciones adicionales que pueden ser de interés, al llevar a cabo el análisis de los filtros. Estas características son:

- Poder ver la respuesta a un impulso unitario en el dominio de la frecuencia.
- Poder ver la respuesta Y[n].
- Poder ver la respuesta en fase.
- Obtener la estructura digital del filtro.
- Obtener la estructura analógica del filtro.

El código empleado para obtener dichas respuestas puede ser consultado en el apéndice A.

Referencias

[1] SPRS032A – APRIL 1996 – REVISED JUNE 1997, Texas Instrument, pp 1-2.
[2] TMS320C3X, User's Guide, Texas Instruments, pp 1-6,1-7.
[5] SLAS017F – MARCH 1988 – REVISED MAY 1995.
[6] TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT PROGRAMS (C) 1990 TEXAS INSTRUMENTS, HOUSTON.



Capítulo 5

RESULTADOS Y CONCLUSIONES.

En el presente capítulo se muestran una serie de capturas de pantallas del programa implementado, haciendo una breve descripción de cada una de ellas. Por último se describen las conclusiones a las que se llegó.

5.1 Capturas de pantalla.

En las siguientes figuras se muestra el programa empleado por el host para comunicarse con el módulo de evaluación del TMS320C30.

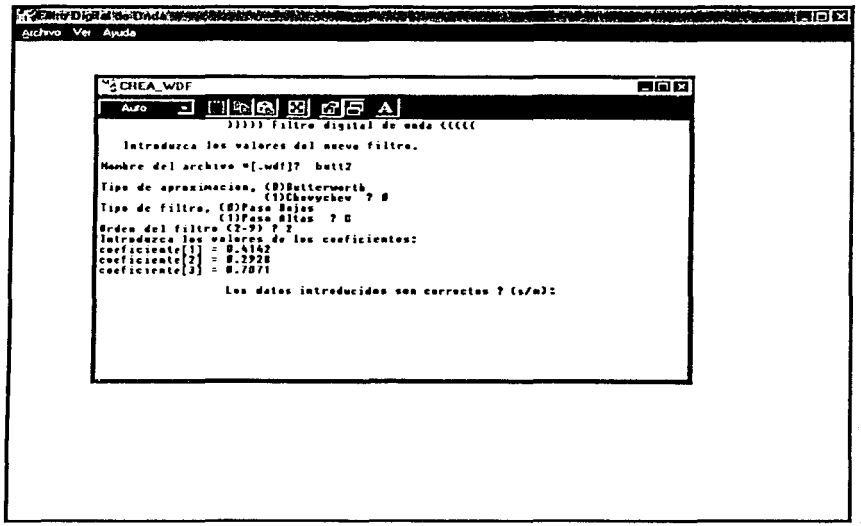


Figura 5.1. Función empleada para crear un nuevo filtro.

En la figura 5.1 se puede observar al programa empleado para crear un nuevo filtro. Se observa como se le piden los datos al usuario y como estos, posteriormente, son almacenados con el nombre de archivo proporcionado al inicio. En la siguiente imagen, figura 5.2, se muestra como se selecciona el archivo a abrir de una serie de configuraciones almacenadas con anterioridad.

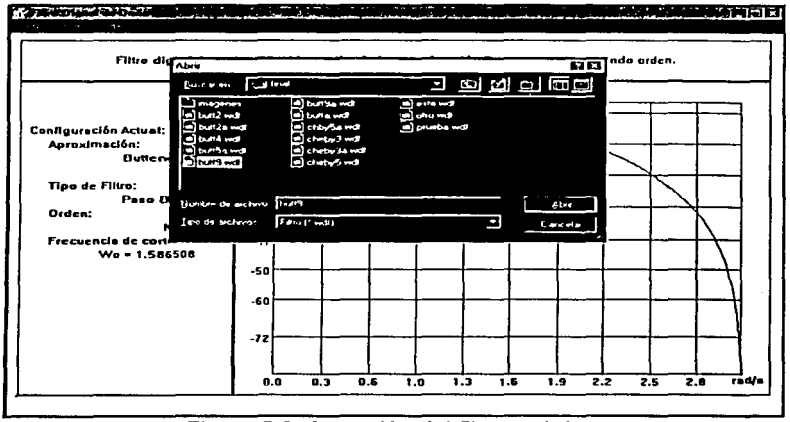


Figura 5.2. Selección del filtro a abrir.

Una vez que ya se ha seleccionado el filtro, el siguiente paso es cargar los datos en el módulo de evaluación, para ello se selecciona del menú principal la opción cargar tal como se muestra en la figura 5.3.

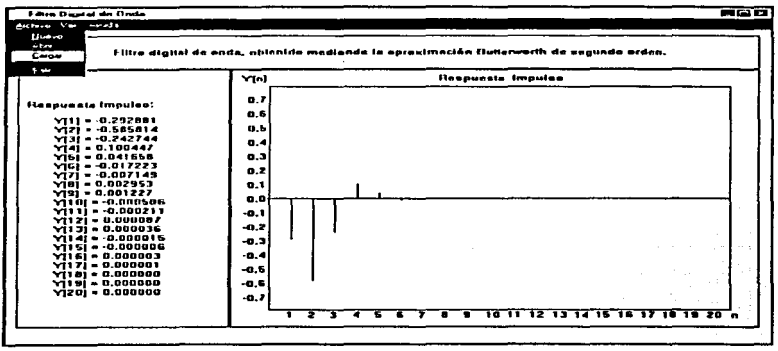


Figura 5.3. Opción cargar para pasar los datos al módulo.

En las siguientes capturas de pantalla se muestran algunas de las funciones adicionales que posee la interfaz del host, en este caso se observar la respuesta en frecuencia y en fase de un filtro paso bajas de orden 2.

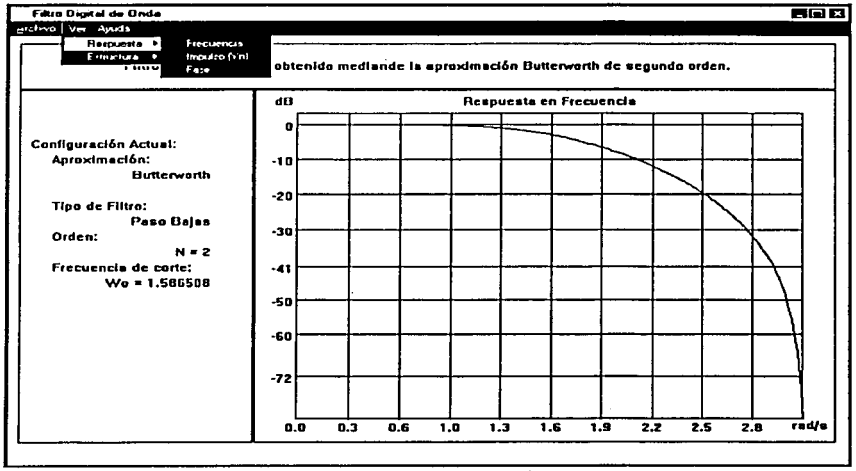
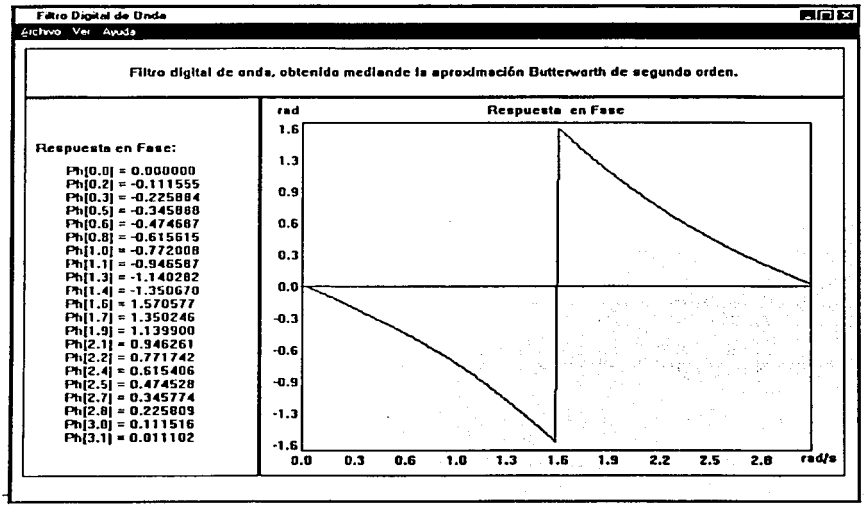


Figura 5.4. Funciones adicionales.



5.2 Conclusiones.

Al llevar a cabo el análisis del diseño de filtros digitales de onda, se pudo observar que existían elementos fundamentales que los constituían por lo que se podría suponer que existiera un algoritmo genérico capaz de realizarlos independientemente del orden o tipo de filtro, paso bajas o paso altas. La búsqueda de este fue el objetivo de la tesis.

Durante el desarrollo de los capítulos se observó como se llevó a cabo el análisis de las ecuaciones que describen a los filtros para buscar similitudes y diferencias que nos permitieran plantear los requerimientos que debe cumplir el programa para que pueda realizarlos y, al mismo tiempo, se obtuvieron los datos que debe proporcionar el usuario al programa. Todo esto se conjuntó para poder llevar a cabo el diseño del algoritmo, el cual podemos decir cumplió con las necesidades planteadas.

- Ser un algoritmo genérico.
- Tener la suficiente sencillez para poder implementarse en el TMS320C30.
- Poder cambiar las características del filtro en tiempo de ejecución, es decir, sin tener que volver a cargar el programa dentro del micro es posible modificar el orden, coeficientes y demás parámetros para realizar un filtro totalmente distinto.

Por lo antes dicho podemos decir que el objetivo planteado al inicio se cumplió cabalmente, quedando sólo algunas limitaciones que se plantean a continuación.

- Con la frecuencia del reloj interno del DSP sólo fue posible obtener una frecuencia de muestreo máxima de alrededor 8kHz, ideal para aplicaciones de audio, pero que limitó a que la frecuencia de corte de los filtros paso bajas diseñados estuviera por debajo a los 4kHz para evitar confundir la respuesta del filtro diseñado con la del filtro interno del DSP.
- El algoritmo resultó trabajar bien para filtros de hasta séptimo orden, pero para filtros mayores a este fue demasiado lento por lo que la señal obtenida a la salida se encontraba distorsionada, por lo que, para resolver este problema, se tuvieron que implementar los filtros de orden ocho y nueve de manera directa.

En base a las limitaciones planteadas se pueden indicar algunas mejoras que pueden realizarse en un futuro trabajo.

- Emplear un reloj externo para obtener frecuencias mayores.
- Optimizar el algoritmo empleando más el lenguaje ensamblador para poder usar operaciones en paralelo, logrando con esto disminuir el

tiempo usado para procesar las muestras, lo que permitiría llevar a cabo la realización de filtros de mayor orden.

- Llevar a cabo el análisis de los filtros paso bandas y ver si estos pueden ser implementados de manera similar o es necesario llevar a cabo el diseño de otro algoritmo.

Apéndice A

Código Fuente

A.1 Librería: adap.h

```
/**********************************************************
REALIZADO POR: Miguel Angel Palomera Perez
FECHA DE REALIZACION: 20/11/2001
DESCRIPCION:
adap.h contiene las funciones para cada uno de los
adaptadores ademas incluye la definicion de
la estructura vector y de sus funciones para llevar a
cabo el manejo dinamico de la memoria.

FECHA DE ULTIMA MODIFICACION: 22/04/2002
**********************************************************/

/*Declaracion de las Funciones*/
double get_ImFD(double XN, double A1, double A2, double N2);
void get_ImFI(double XN,double A1,double A2,double N2,double *YN,double *N1);
double get_PaFD(double XN, double B1, double B2, double N2);
void get_PaFI(double XN,double B1,double B2,double N2,double *YN,double *N1);
double get_PaD(double XN,double A,double N2);
void get_PaI(double XN,double A,double N2,double BN1,double *BN,double *N1);
double get_ImD(double XN,double B,double N2);
void get_ImI(double XN,double B,double N2,double BN1,double *BN,double *N1);
/*Reagrupamos las funciones para mas facil manejo */

/*Funciones para los elementos intermedios*/

/*dependientes*/
double get_EID(double XN,double A,double N2,int tipo);
/*Independientes*/
void get_EII(double XN,double A,double N2,double BN1,double *BN,double *N1,int tipo);
```

```
/*Funciones para los elementos finales*/

/*Dependientes*/
double get_FD(double XN, double A1, double A2, double N2,int tipo);
/*Independientes*/
void get_FI(double XN,double A1,double A2,double N2,double *YN,double *N1,int tipo);
/*Funciones miscelaneas empleadas*/
/*Funcion para obtener la potencia de un numero*/
int pot(int i,int p);
/*Fucion para obtener los elementos que componen el filtro*/
int *get_fil(int N);
```

A.2 Programa: adap.cpp

```
/*****************************************************
REALIADO POR: Miguel Angel Palomera Pérez
FECHA DE REALIZACION: 20/11/2001
DESCRIPCION:
                adap.cpp contine las declaraciones de las funciones
    definidas en adap.h
FECHA DE ULTIMA MODIFICACION: 22/04/2001
*****************************************************/
#include "adap.h"
#include <iostream.h>

double get_ImFD(double XN, double A1, double A2, double N2){
  return XN-XN*A1+2*N2-A1*N2-A2*N2;
}

void get_ImFI(double XN,double A1,double A2,double N2,double *YN,double *N1){
        *YN = -A1*XN+2*N2-A2*N2-A1*N2;
        *N1 = N2-A1*N2-A2*N2-XN*A1;
}

double get_PaFD(double XN, double B1, double B2, double N2){
        return XN-XN*B1-N2*B1;
}

void get_PaFI(double XN,double B1,double B2,double N2,double *YN,double *N1){
        *YN = -XN*B2-N2*B2;
        *N1 = XN-XN*B2+N2-N2*B2+XN-XN*B1-N2*B1;
}

double get_PaD(double XN,double A,double N2){
        return -XN*A+N2-N2*A;
}

void get_PaI(double XN,double A,double N2,double BN1,double *BN,double *N1){
        *BN = XN-XN*A+N2-N2*A+BN1;
        *N1 = -XN*A-N2*A+BN1;
}

double get_ImD(double XN,double B,double N2){
        return XN+N2;
}
```

```
void get_ImI(double XN,double B,double N2,double BN1,double *BN,double *N1){
        *BN = XN-B*(XN+N2)-B*BN1;
        *N1 = *BN+BN1;
}

double get_EID(double XN,double A,double N2,int tipo){
        if (tipo == 1)
                return get_PaD(XN,A,N2);
        else
                return get_ImD(XN,A,N2);

}

void get_EII(double XN,double A,double N2,double BN1,double *BN,double *N1,int tipo){
        if (tipo == 1)
                get_PaI(XN,A,N2,BN1,BN,N1);
        else
                get_ImI(XN,A,N2,BN1,BN,N1);
}

double get_FD(double XN, double A1, double A2, double N2,int tipo){
        if (tipo == 1)
                return get_PaFD(XN,A1,A2,N2);
        else
                return get_ImFD(XN,A1,A2,N2);
}
void get_FI(double XN,double A1,double A2,double N2,double *YN,double *N1,int tipo){
        if (tipo == 1)
                get_PaFI(XN,A1,A2,N2,YN,N1);
        else
                get_ImFI(XN,A1,A2,N2,YN,N1);
}
int pot(int i,int p){
        if(p==0)
                return 1;
        else
                return i*pot(i,p-1);
}
int *get_fil(int N){
        int *filtro;
        int i=0;
        filtro = new int[N];
        for(i=0;i<N-2;i++){
                filtro[i]=pot(-1,i+1);
        }
        if( N%2 == 0)
                filtro[N-2]=1;
        else
                filtro[N-2]=-1;

        return filtro;

}
```

A.3 Librería: complejo.h

```
/* REALIZADO POR: Miguel A. Palomera Perez
        FECHA DE REALIACION: 22-04-2002
        DESCRIPCION:
  archivo complejo.h que contiene la definicion de la clase complejo
  empleada para calcular la respuesta en frecuencia del filtro.
*/

#ifndef __COMPLEJO_H__
#define __COMPLEJO_H__
#include <iostream.h>
class complejo
        {
   private:
     double real;
          double imag;
   public:
   // Constructores
   complejo(void);
   // SetThings
   void SetReal(double);
   void SetImag(double);
   // GetThings
   double GetReal(void){return real;}
   double GetImag(void){return imag;}
   // Sobrecarga de operadores aritméticos
   complejo operator+ (const complejo&);
   complejo& operator= (const complejo&);
   // Funcion empleadas
   double modulo(void);
   void expon(double,double);
   double fase(void);
};
#endif
```

A.4 Programa: complejo.cpp

```
/* REALIZADO POR: Miguel A. Palomera Pérez
        FECHA DE REALIACION: 22-04-2002
        DESCRIPCION:
  archivo complejo.cpp que contiene la declaracion  de la clase complejo
  empleada para calcular la respuesta en frecuencia del filtro.
*/
#include "complejo.h"
#include "math.h"
// constructor por defecto
complejo::complejo(void)
{
        real = 0.0;
  imag = 0.0;
}
```

```
void complejo::SetReal(double re)
{
        real = re;
}
void complejo::SetImag(double im)
{
imag = im;
}
// operador miembro + sobrecargado
complejo complejo::operator+ (const complejo &a)
{
        complejo suma;
        suma.real = real + a.real;
  suma.imag = imag + a.imag;
        return suma;
}
// operador miembro de asignación sobrecargado
complejo& complejo::operator= (const complejo &a)
{
        real = a.real;
        imag = a.imag;
        return (*this);
}
// funcion modulo
double complejo::modulo(void)
{
        return sqrt(pow(real,2)+pow(imag,2));
}

// funcion exponencial compleja
void complejo::expon(double exp,double coef)
{
    real=cos(exp);
    imag=sin(exp);
    real=real*coef;
    imag=imag*coef;
}

// funcion para obtener la fase
double complejo::fase(void)
{
  if(real == 0.0 && imag > 0)return 3.1416/2;
  if(real == 0.0 && imag < 0)return -3.1416/2;
  if(real == 0.0 && imag == 0)return 0;
        return atan(imag/real);
}
```

A.5 Librería: freqz.h

```
/*********************************************************************
 REALIZADO POR: Miguel Angel Palomera Perez
 FECHA DE REALIZACION: 02/01/2002
 FECHA DE ULTIMA MODIFICACION: 19/01/2002
 DESCRIPCION:
        freqz.h definicion  de las funciones necesarias
        para calcular la respuesta en frecuencia de un filtro digital
        de la forma
                    jw          -jw           -jnbw
      jw  B(e)   b(1) + b(2)e + .... + b(nb+1)e
    H(e) = --- = ----------------------------
           jw          -jw           -jnaw
         A(e)   a(1) + a(2)e + .... + a(na+1)e
*********************************************************************/
#include "complejo.h"
complejo freqz(double*,double,int);
```

A.6 Programa: freqz.cpp

```
/*********************************************************************
 REALIZADO POR: Miguel Angel Palomera Perez
 FECHA DE REALIZACION: 2/04/2002

 DESCRIPCION:
        freqz.cpp declaracion de las funciones necesarias
        para calcular la respuesta en frecuencia de un filtro digital
        de la forma
                    jw          -jw           -jnbw
      jw  B(e)   b(1) + b(2)e + .... + b(nb+1)e
    H(e) = --- = ----------------------------
           jw          -jw           -jnaw
         A(e)   a(1) + a(2)e + .... + a(na+1)e
*********************************************************************/
#include "freqz.h"

complejo freqz(double *B,double w,int tamB)
{
  complejo bn,sum;
  //calculo del numerador
  for(int i=0;i<tamB;i++)
  {
    bn.expon(-i*w,B[i]);
    sum=sum+bn;
  }
  // calculo del valor
  return sum;
}
```

A.8 Librería: filtro.h

```
/*REALIZADO POR: Miguel Angel Palomera Pérez
  FECHA DE REALIZACION: 22/04/2002
  DESCRIPCION:
  filtro.h contine la definicion de la clase filtro, objeto principal
  del programa y de sus funciones miembro.

*/
#include <iostream.h>
#include "freqz.h"
#include "adap.h"
#define MAX_M 100 //Numero maximo de muestras para la respuesta en Frecuencia

class filtro
{
 //elementos de la clase
    private:
     int tipo;
     int orden;
     int *estructura;
     double *Yn;
     double *coef;
     complejo *Yf;

  //Metodos de la clase
  public:
  //Constructor
  filtro(void);
  filtro(int, int, double*);
  //Destructor
  ~filtro()
  {
    delete [] estructura;
    delete [] coef;
    delete [] Yn;
    delete [] Yf;
  }
  //Funciones varias
  void getFiltro(void);
  void setFiltro(int,int,double *);
  int getTipo(void){return tipo;};
  int getOrden(void){return orden;};
  double *getYn(void){return Yn;};
  double *getCoef(void){return coef;};
  complejo *getYf(void){return Yf;};
};
```

A.8 Programa: filtro.cpp

```
/*REALIZADO POR: Miguel Angel Palomera Perez
  FECHA DE REALIZACION: 22/04/2002
  DESCRIPCION:
  filtro.h contine la definicion de la clase filtro, objeto principal
  del programa y de sus funciones miembro.

*/
#include "filtro.h"

// Constructor por defecto
filtro::filtro(void){
  tipo=0;
  orden=2;
  coef = new double[3];
  coef[0]=0.4142;
  coef[1]=0.2928;
  coef[2]=0.7071;
  Yn = new double[MAX_M];
  Yf = new complejo[MAX_M*2];
}
// Constructor
filtro::filtro(int tip,int ord,double *c){
  tipo=tip;
  orden=ord;
  coef = new double[orden+1];
  for(int i=0;i<=ord;i++)
  coef[i]=c[i];
  Yn = new double[MAX_M];
  Yf = new complejo[MAX_M*2];
}

// Funcion para calcular el filtro
void  filtro::getFiltro(void){
//Definimos las variables intermedias para calcular el filtro
  double BN,XN=1;
  double *X,*B,*N;
  //Definicion de las variables para el calculo de la respuesta en frecuencia
  double inc = 3.1416/(MAX_M*2);
  double *W;

  X = new double[orden-1];
  B = new double[orden-1];
  N = new double[orden];
  for(int i=0;i<orden;i++)
  N[i]=0.0;
  //Calculo del filtro
  if(orden == 2){
  for(int i=0;i<MAX_M;i++){
      X[0] = get_EID(-XN,coef[0],N[0],1);
      B[0] = get_FD(X[0],coef[1],coef[2],N[1],1);
      get_FI(X[0],coef[1],coef[2],N[1],&Yn[i],&N[1],1);
```

```
      get_EII(-XN,coef[0],N[0],B[0],&BN,&N[0],1);
      XN=0;
      if(tipo == 1){ /*Si es filtro paso altas se invierten los signos*/
      for(int r=0;r<orden;r++)N[r]=-N[r];
      }
    }
  }
  else{
    structura = get_fil(orden);
    for(int i=0;i<MAX_M;i++){
     X[0] = get_EID(-XN,coef[0],N[0],1);
      for(int k=0;k<orden-2;k++)
      X[k+1] = get_EID(X[k],coef[k+1],N[k+1],estructura[k]);
      B[orden-2] = get_FD(X[orden-2],coef[orden-1],coef[orden],N[orden-1],estructura[orden-2]);
    get_FI(X[orden-2],coef[orden-1],coef[orden],N[orden-1],&Yn[i],&N[orden-1],estructura[orden-2]);
      for(int k=orden-2;k>0;k--)
      get_EII(X[k-1],coef[k],N[k],B[k],&B[k-1],&N[k],estructura[k-1]);
      get_EII(-XN,coef[0],N[0],B[0],&BN,&N[0],1);
      XN=0;
      if(tipo == 1){ /*Si es filtro paso altas se invierten los signos*/
      for(int r=0;r<orden;r++)N[r]=-N[r];
      }
    }
    }
  // Calculo de la respuesta en frecuencia
  W = new double[MAX_M*2];
  W[0]=0;
  for(int i=1;i<MAX_M*2;i++)W[i]=W[i-1]+inc;
  for(int i=0;i<MAX_M*2;i++)
  Yf[i]=freqz(Yn,W[i],MAX_M);
//Liberamos la memoria empleada
  delete [] X;
  delete [] B;
  delete [] N;
  delete [] W;
}

// Funcion para actualizar dinamicamente al filtro

void filtro::setFiltro(int tip,int ord,double *c){
  tipo=tip;
  orden=ord;
  //Liberamos la memoria anterior y solicitamos la necesaria para el nuevo
  delete [] coef;
  coef = new double[ord+1];
  for(int i=0;i<=ord;i++)
  coef[i]=c[i];
}
```

A.9 Librería: wdf.h

```
#define STRICT
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <commdlg.h>
#include <stdlib.h>
#include <math.h>
#include "ventana.h"
#include "filtro.h"
//Estructura para leer los datos de un archivo de acceso aleatorio
typedef struct{
int aprox; /* tipo de aproximacion: 0 Butterworth, 1 Chevichev*/
int orden; /* orden del filtro */
int tipo;  /* tipo de filtro: 0 paso bajas, 1 paso altas;*/
double coef[10];
}wdf_file;

typedef struct{
COLORREF Borde;
COLORREF Relleno;
RECT     Dimensiones;
}Rect;

/*********************************
****Funciones varias
*********************************/

//Funcion para obtener el nombre del archivo a abrir
BOOL Get_File( HWND hWnd );
//Funcion para obtener y dibujar la respuesta en frecuencia
void getResFrec(HWND hWnd,int aprox);
//Funcion para obtener y dibujar la respuesta impulso
void getResImp(HWND hWnd);
//Funcion para obtener y dibujar la respuesta en fase
void getResFase(HWND hWnd);
//Funcion para obtener y dibujar la estructura analogica
void getEstAna(HWND hWnd);
//Funcion para obtener y dibujar la estructura digital
void getEstDig(HWND hWnd);
//Funcion encargada de atender los eventos que ocurran en la ventana
LRESULT CALLBACK DoWMEvent (HWND, UINT, WPARAM, LPARAM);
//Funcion encargada de realizar las opciones del menu
int  DoWMCommand(WPARAM wParam, HWND hWnd);
//Funcion empleada para cargar una imagen desde un archivo bmp
BOOL CargaBMP( LPTSTR szFileName, HBITMAP *phBitmap,HPALETTE *phPalette );
//Funcion para dibujar rectangulos;
void DibujaRect(HDC hdc,Rect rect);
void DibujaTexto(HDC hdc,char *mensaje,COLORREF TextColor,COLORREF Fondo,RECT
Posicion,UINT format);
BOOL DibujaBMP(HDC hdc,LPTSTR bmpfile,int x,int y,int *width);
```

A.10 Programa: wdf.cpp

```
//*************************************************************//
//REALIZADO POR: Miguel Angel Palomera                         //
//FECHA DE REALIZACION: 04/06/2002                             //
//DESCRIPCION: wdf.cpp, es la interfaz principal de            //
// comunicacion entre el usuario y el modulo de                //
// evaluacion TMS320C30                                        //
//*************************************************************//


#include <windowsx.h>
#include "wdf.h"


/*************************************************************
 ** Variables globales
 *************************************************************/
 char szName[256]; //Nombre del archivo a abrir
 RECT rCliente;    //Dimensiones de la ventana a dibujar
 HDC hdcMem;       //Objeto DC empleado para dibujar en la ventana
 HBITMAP bmpMem;   //Mapa de Bits empleado para almacenar los cambios hechos en la ventana
 BOOL update=FALSE; //Bandera que indica si es necesario llevar a cabo un repaint.
 filtro wfiltro;   //Objeto Filtro
 int aprox;        //Aproximacion
 char *orden[]=    //Orden del filtro
 {
  "segundo",
   "tercer",
   "cuarto",
   "quinto",
   "sexto",
   "septimo",
   "octavo",
   "noveno",
   NULL
 };
 /*************************************************************
 * funcion WinMain
 *************************************************************/
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
{
   WNDCLASS   wndClass;
   MSG        msg;
   HWND       hWnd;

   /*
    * Se crea una nueva clase ventana
    * y se registra en la api
    */
```

```
    if (!hPrevInstance)
    {
       wndClass.style         = CS_VREDRAW | CS_HREDRAW;
       wndClass.lpfnWndProc   = DoWMEvent;
       wndClass.cbClsExtra    = 0;
       wndClass.cbWndExtra    = 0;
       wndClass.hInstance     = hInstance;
       wndClass.hIcon         = LoadIcon(hInstance, "ico_wdf");
       wndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
       wndClass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
       wndClass.lpszMenuName   = szAppName;
       wndClass.lpszClassName  = szAppName;

       if (!RegisterClass(&wndClass))
          return FALSE;
    }

    /*
     * Se crea y se muestra la ventana.
     */
    hWnd = CreateWindow(szAppName, "Filtro Digital de Onda",
           WS_OVERLAPPEDWINDOW,0,0,
           800, 600, NULL, NULL, hInstance, NULL);

    ShowWindow(hWnd, SW_SHOW);
    UpdateWindow(hWnd);

    //Se entra en un ciclo infinito en la espera que ocurra un evento
    while (GetMessage(&msg, NULL, 0, 0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }
    return 0;
}

/*****************************************************
***Definicion de las funciones declaradas en wdf.h
*****************************************************/

//Funcion encargada de manejar los eventos que ocurran en la ventana
LRESULT CALLBACK DoWMEvent (HWND hWnd, UINT Message,WPARAM wParam, LPARAM
lParam)
{
  HDC hdc;                // HDC para esta ventana
  PAINTSTRUCT ps;     // Paint Struct para las llamadas BeginPaint

   switch(Message)
   {
        case WM_CREATE: // Se inicializan las variables globales
        strcpy( szName, "" );          // Se limpia la cadena szName
         GetClientRect(hWnd, &rCliente); //Se obtienen las dimensiones de la ventana
         hdc = BeginPaint(hWnd, &ps);    // Se reserva memoria para
         hdcMem = CreateCompatibleDC(hdc);
```

```
            bmpMem = CreateCompatibleBitmap(hdc,
                     rCliente.right-rCliente.left,
                     rCliente.bottom-rCliente.top);
            EndPaint(hWnd, &ps);
          return 0L;
          case WM_PAINT:
          //Si es necesario se repinta la ventana, con la imagen guardada
          //en memoria
          hdc = BeginPaint(hWnd, &ps);
          SelectObject(hdcMem, bmpMem);
          BitBlt(hdc, rCliente.left, rCliente.top,
             rCliente.right-rCliente.left,
             rCliente.bottom-rCliente.top,
             hdcMem, 0, 0, SRCCOPY);
          EndPaint(hWnd, &ps);
          break;

        case WM_COMMAND:
        //si ocurre un evento del menu se llama al manegador correspondiente
        return DoWMCommand(wParam,hWnd);

        case WM_DESTROY:
        //Se libera la memoria empleada y se sale del ciclo infinito.
        DeleteDC(hdcMem);
        DeleteObject(bmpMem);
        PostQuitMessage(0);
        break;

        default:
           return DefWindowProc(hWnd, Message, wParam, lParam);
    }
    return 0;
}

//Manejador de las opciones del menu
int DoWMCommand(WPARAM wParam,HWND hWnd){
 switch(wParam)
        {
           case MID_NUEVO:
             //Se hace una llamada al comando crea_wdf, el cual crea
             //un archivo *.wdf con los datos del nuevo filtro
             system("crea_wdf");
           break;

           case MID_ABRIR:
             //Se obtiene el nombre del archivo
              if(Get_File(hWnd))
              {
               wdf_file wdfDatos; //Estructura para leer los datos del archivo
               FILE *in;
               in=fopen(szName,"r");
               fread(&wdfDatos,sizeof(wdfDatos),1,in);
               fclose(in);
               wfiltro.setFiltro(wdfDatos.tipo,wdfDatos.orden,wdfDatos.coef);
```

```
            wfiltro.getFiltro();
            getResFrec(hWnd,wdfDatos.aprox);
            aprox=wdfDatos.aprox;
          }
          break;
    case MID_CARGAR :
          if(update){
           char stream[256],aux[20];
           strcpy(stream,"car_wdf ");
           double *coef;
           coef=wfiltro.getCoef();
           sprintf(aux,"%d.0,%d.0",wfiltro.getOrden(),wfiltro.getTipo());
           strcat(stream,aux);
           for(int i=0;i<=wfiltro.getOrden();i++){
             sprintf(aux,",%f",coef[i]);
             strcat(stream,aux);
          }
          system(stream);
          }
        break;

    case MID_SALIR:
          //Damos un reset al modulo
          system("evmload Evminit.out");
          system("evmhost -p 340");
          //Destruimos la ventana
           DestroyWindow(hWnd);
           break;

       case MID_FRECUENCIA:
          if(update)
          getResFrec(hWnd,aprox);
       break;
       case MID_IMPULSO:
          if(update)
          getResImp(hWnd);
       break;

       case MID_FASE:
          if(update)
          getResFase(hWnd);
          break;
     case MID_ANALOGICO:
          if(update)
         getEstAna(hWnd);
         break;

       case MID_DIGITAL:
          if(update)
          getEstDig(hWnd);
          break;
```

```
        case MID_ACERCA:
          MessageBox(hWnd,szAcerca,"Filtro Digital de Onda",MB_OK|MB_ICONINFORMATION);
         break;

        default:
         break;
   }
  return 0;
}[1]
```

Nota aclaratoria: El resto del código fuente se omite, ya que este fue discutido en el capítulo 4 en el apartado correspondiente.

[1] El resto de las funciones se omiten ya que se tratan o de funciones genericas (como dibujar pixeles en una patalla) o que emplean algunas de las funciones definidas en los programas anteriores.

Apéndice B

Modos de Direccionamiento.

El TMS320C30 soporta cinco grupos de formas de direccionamiento. Seis tipos de acceso a memoria pueden emplearse dentro de estos grupos, lo cual nos permite acceder a datos dentro de la memoria, registros y palabras de código.

B.1 Tipos de direccionamiento.

Se tienen seis formas de acceder a datos, registros y palabras de código.

- Registros
- Directo
- Indirecto
- Inmediato-corto
- Inmediato-largo
- PC-relativo

Algunos tipos de direccionamiento se aplican algunas instrucciones otros no. Por esta razón, los tipos de direccionamiento se emplean en dentro de cinco grupos de acceso a memoria, los cuales son:

- Modo de direccionamiento general (G).
  - Registros
  - Directo
  - Indirecto
  - Inmediato-corto

- Modo de tres operadores (T):
  - Registros
  - Indirecto

- Modo de direccionamiento paralelo (P):
  - Registros
  - Indirecto
- Modo de direccionamiento de salto condicional (B):
  - Registros
  - PC-Relativo
- Direccionamiento Circular.

B.2 Direccionamiento por medio de Registros.

En este tipo, un registro del CPU contiene al operador, como se muestra en el ejemplo.

ABSF R1; R1=|R1|

B.3 Direccionamiento Directo.

En este tipo, la dirección del dato se forma por la concatenación de los ocho menos significativos bits del apuntador de página (DP) con los 16 menos significativos bits de la expresión (expr). Esto resulta en 256 páginas (64k palabra por página), dando al programador una gran cantidad de espacio para el direccionamiento sin tener que cambiar de página. La sintaxis y operación para el direccionamiento directo son:

Sintaxis: @expr
Operación: dirección = DP concatenado con expr.

Ejemplo:

ADDI @0BCDEh, R7

Antes de la instrucción.	Después de la instrucción.
DP=8Ah	DP=8Ah
R7=0h	R7=12345678h
Dato en 8ABCDEh=12345678h	Dato en 8ABCDEh=12345678h

B.4 Direccionamiento Indirecto

Este direccionamiento obtiene la dirección de un operador a través de un registro auxiliar, opcionalmente puede incrementar o decrementar estos registros. Las operaciones aritméticas requeridas son hechas por medio de las unidades aritméticas de los registros auxiliares (ARAUS) de manera paralela con el CPU. El

modo de direccionamiento indirecto es especificado por el quinto campo de la palabra de instrucción. El incremento o decremento se especifica por un entero de ocho bit incluido en la palabra de código o se emplea un desplazamiento por omisión igual a uno[1].

Ejemplos:

Sintaxis	Operación
*+ARn(disp)	Addr=ARn+disp
*ARn++(IR0)	Addr= ARn ARn=ARn+IR0
*--ARn(IR1)	Addr=ARn-IR1 ARn=ARn-IR1

B.5 Direccionamiento Inmediato corto.

En este tipo de direccionamiento, el operador es un valor inmediato de 16 bits contenido en los 16 bits menos significativos de la palabra de código. Dependiendo del tipo asumido por la palabra de código puede ser, un complemento a dos, un entero sin signo, un número de punto flotante. Ejemplo:

SUBI 1, R0

Antes	Después
R0=0h	R0=0FFFFFFFFh

B.6 Direccionamiento Inmediato largo.

El operador es un valor inmediato de 24 bits contenido en los 24 bits menos significativos de la palabra de código. Ejemplo:

BR 8000h

Antes	Después
PC=0h	R0=8000h

B.7 Direccionamieto PC-relativo.

Este direccionamiento se emplea para realizar saltos en el programa. Este agrega el contenido de los 16 o 24 menos significativos bits de la palabra de código al registro contador del programa (PC). El ensamblador toma la dirección origen (una etiqueta o dirección) dada por el usuario y genera el desplazamiento.

El desplazamiento se almacena como un entero sin signo en los últimos 16 o 24 bits menos significativos de la palabra de código.

[1] Consultar la Guía del Usuario del TMS320CX para poder observar la lista completa para este tipo de direccionamiento, pp 5-6,5-7

Apéndice C

Tablas.

Hasta el momento se tienen ya calculados los coeficientes para distintos tipos de filtros, que pueden ser realizados mediante filtros digitales de onda, los valores obtenidos se incluyen en esta sección.

C.1 Elementos del FDO[1] Butterworth n = 2, a_{max} [dB].

a_{max}	α_1	β_{21}	β_{22}
0.1	0.6444	0.5868	0.9106
0.2	0.6031	0.5334	0.8845
0.3	0.5779	0.5007	0.8664
0.4	0.5595	0.4768	0.8522
0.5	0.5450	0.4580	0.8404
0.6	0.5329	0.4424	0.8301
0.7	0.5226	0.4290	0.8210
0.8	0.5135	0.4174	0.8127
0.9	0.5054	0.4070	0.8052
1.0	0.4981	0.3976	0.7982

[1] FDO: Filtro Digital de Onda.

C.2 Elementos del FDO Butterworth n = 3, a_{max} [dB].

a_{max}	α_1	β_2	α_{31}	α_{32}
0.1	0.6519	0.3790	0.5497	0.9454
0.2	0.6248	0.3422	0.5099	0.9310
0.3	0.6083	0.3209	0.4858	0.9211
0.4	0.5964	0.3059	0.4685	0.9134
0.5	0.5864	0.2943	0.4548	0.9069
0.6	0.5791	0.2849	0.4434	0.9014
0.7	0.5723	0.2769	0.4337	0.8964
0.8	0.5664	0.2700	0.4252	0.8920
0.9	0.5611	0.2630	0.4176	0.8878
1.0	0.5562	0.2585	0.4208	0.8840

C.3 Elementos del FDO Butterworth n = 4, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_{41}	β_{42}
0.1	0.6765	0.3695	0.3212	0.5692	0.9680
0.2	0.6570	0.3426	0.2927	0.5387	0.9600
0.3	0.6451	0.3270	0.2762	0.5203	0.9546
0.4	0.6365	0.3159	0.2647	0.5069	0.9504
0.5	0.6297	0.3073	0.2528	0.4963	0.9465
0.6	0.6240	0.3002	0.2485	0.4876	0.9438
0.7	0.6191	0.2942	0.2424	0.4801	0.9411
0.8	0.6149	0.2891	0.2371	0.4735	0.9386
0.9	0.6110	0.2845	0.2325	0.4676	0.9363
1.0	0.6075	0.2803	0.2283	0.4622	0.9342

C.4 Elementos del FDO Butterworth n = 5, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_{51}	α_{52}
0.1	0.7022	0.3874	0.2867	0.3188	0.6021	0.9815
0.2	0.6872	0.3658	0.2655	0.2951	0.5781	0.9722
0.3	0.6782	0.3531	0.2532	0.2813	0.5636	0.9742
0.4	0.6716	0.3441	0.2446	0.2717	0.5530	0.9718
0.5	0.6664	0.3371	0.2380	0.2642	0.5446	0.9699
0.6	0.6621	0.3313	0.2325	0.2580	0.5376	0.9682
0.7	0.6584	0.3264	0.2280	0.2529	0.5317	0.9667
0.8	0.6551	0.3222	0.2240	0.2484	0.5264	0.9653
0.9	0.6522	0.3184	0.2205	0.2444	0.5217	0.9641
1.0	0.6495	0.3149	0.2173	0.2408	0.5174	0.9629

C.5 Elementos del FDO Butterworth n = 6, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_{61}	β_{62}
0.1	0.7256	0.4125	0.2871	0.2635	0.3355	0.6363	0.9895
0.2	0.7131	0.3944	0.2696	0.2458	0.3149	0.6170	0.9870
0.3	0.7066	0.3837	0.2595	0.2356	0.3029	0.6053	0.9854
0.4	0.7014	0.3761	0.2523	0.2284	0.2948	0.5968	0.9841
0.5	0.6072	0.3702	0.2468	0.2229	0.2879	0.5901	0.9830
0.6	0.6938	0.3653	0.2422	0.2184	0.2824	0.5845	0.9821
0.7	0.6909	0.3611	0.2384	0.2145	0.2779	0.5797	0.9813
0.8	0.6883	0.3574	0.2350	0.2112	0.2739	0.5754	0.9805
0.9	0.6859	0.3542	0.2321	0.2083	0.2703	0.5716	0.9798
1.0	0.6838	0.3512	0.2294	0.2056	0.2671	0.5681	0.9792

C.6 Elementos del FDO Butterworth n = 7, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_6	α_{71}	α_{72}
0.1	0.7462	0.4391	0.2994	0.2497	0.2628	0.3597	0.6679	0.9940
0.2	0.7635	0.4236	0.2842	0.2352	0.2473	0.3415	0.6522	0.9927
0.3	0.7307	0.4144	0.2754	0.2268	0.2384	0.3308	0.6428	0.9917
0.4	0.7265	0.4078	0.2692	0.2208	0.2320	0.3232	0.6358	0.9911
0.5	0.7231	0.4027	0.2643	0.2162	0.2271	0.3173	0.6304	0.9904
0.6	0.7204	0.3984	0.2603	0.2125	0.2231	0.3125	0.6258	0.9899
0.7	0.7280	0.3948	0.2569	0.2093	0.2197	0.3083	0.6218	0.9895
0.8	0.7159	0.3916	0.2539	0.2065	0.2168	0.3047	0.6184	0.9890
0.9	0.7140	0.3887	0.2513	0.2041	0.2142	0.3015	0.6153	0.9887
1.0	0.7123	0.3862	0.2490	0.2019	0.2118	0.2986	0.6124	0.9883

C.7 Elementos del FDO Butterworth n = 8, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_6	α_7	β_{81}	β_{82}
0.1	0.7643	0.4653	0.3166	0.2512	0.2375	0.2723	0.3867	0.6960	0.9966
0.2	0.7563	0.4518	0.3031	0.2385	0.2249	0.2583	0.3704	0.6832	0.9959
0.3	0.7514	0.4437	0.2952	0.2311	0.2117	0.2502	0.3608	0.6754	0.9954
0.4	0.7479	0.4379	0.2896	0.2259	0.2124	0.2445	0.3540	0.6697	0.9949
0.5	0.7451	0.4334	0.2852	0.2219	0.2083	0.2400	0.3487	0.6652	0.9946
0.6	0.7628	0.4297	0.2816	0.2186	0.2051	0.2364	0.3442	0.6614	0.9943
0.7	0.7408	0.4265	0.2786	0.2158	0.2023	0.2333	0.3405	0.6584	0.9941
0.8	0.7391	0.4237	0.2759	0.2133	0.1999	0.2306	0.3372	0.6553	0.9938
0.9	0.7375	0.4212	0.2735	0.2112	0.1977	0.2282	0.3343	0.6527	0.9936
1.0	0.7361	0.4190	0.2714	0.2092	0.1958	0.2260	0.3316	0.6504	0.9934

C.8 Elementos del FDO Butterworth n = 9, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_6	α_7	β_8	α_{91}	α_{92}
0.1	0.7802	0.4902	0.3585	0.2595	0.2303	0.2373	0.2870	0.4142	0.7207	0.9981
0.2	0.7734	0.4783	0.3236	0.2481	0.2194	0.2258	0.2741	0.3996	0.7989	0.9987
0.3	0.7693	0.4712	0.3164	0.2415	0.2130	0.2192	0.2667	0.3910	0.7035	0.9974
0.4	0.7663	0.4661	0.3114	0.2368	0.2085	0.2146	0.2614	0.3848	0.6987	0.9972
0.5	0.7640	0.4621	0.3074	0.2331	0.2050	0.2109	0.2573	0.3799	0.6950	0.9970
0.6	0.7620	0.4587	0.3041	0.2301	0.2021	0.2079	0.2539	0.3759	0.6920	0.9968
0.7	0.7604	0.4599	0.3013	0.2276	0.1997	0.2054	0.2510	0.3725	0.6892	0.9967
0.8	0.7589	0.4534	0.2989	0.2254	0.1976	0.2032	0.2485	0.3695	0.6868	0.9965
0.9	0.7576	0.4512	0.2967	0.2234	0.1957	0.2013	0.2463	0.3669	0.6847	0.9964
1.0	0.7564	0.4492	0.2947	0.2216	0.1940	0.1995	0.2442	0.3645	0.6827	0.9963

C.9 Elementos del FDO Chebychev n = 3, a_{max} [dB].

a_{max}	α_1	β_2	α_{31}	α_{32}
0.10	0.4922	0.3022	0.4620	0.7574
0.25	0.4341	0.2774	0.4310	0.6812
0.50	0.3852	0.2599	0.4126	0.6114
1.00	0.3307	0.2496	0.3995	0.5293
2.00	0.2695	0.2445	0.3929	0.4331
3.00	0.2300	0.2442	0.3925	0.3696

C.10 Elementos del FDO Chebychev n = 5, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_{51}	α_{52}
0.1	0.4658	0.2536	0.2161	0.2245	0.4179	0.7374
0.25	0.4197	0.2404	0.2059	0.2132	0.3987	0.6721
0.50	0.3696	0.2311	0.1975	0.2044	0.3869	0.5965
1.00	0.3190	0.2262	0.1911	0.1981	0.3798	0.5168
2.00	0.2610	0.2251	0.1857	0.1933	0.3797	0.4229
3.00	0.2231	0.2265	0.1828	0.1922	0.3830	0.3608

C.11 Elementos del FDO Chebychev n = 7, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_6	α_{71}	α_{72}
0.1	0.4585	0.2437	0.2022	0.1947	0.1962	0.2123	0.4049	0.7313
0.25	0.4087	0.2316	0.1944	0.1884	0.1903	0.2028	0.3875	0.6550
0.50	0.3653	0.2250	0.1893	0.1861	0.1867	0.1968	0.3782	0.5925
1.00	0.3158	0.2213	0.1847	0.1834	0.1836	0.1919	0.3735	0.4206
2.00	0.2587	0.2210	0.1805	0.1814	0.1812	0.1880	0.3745	0.4206
3.00	0.2213	0.2227	0.1783	0.1806	0.1802	0.1862	0.3782	0.3589

C.12 Elementos del FDO Chebychev n = 9, a_{max} [dB].

a_{max}	α_1	β_2	α_3	β_4	α_5	β_6	α_7	β_8	α_{91}	α_{92}
0.10	0.4554	0.2399	0.1979	0.1886	0.1854	0.1860	0.1908	0.2081	0.4000	0.7287
0.25	0.4064	0.2287	0.1912	0.1847	0.1821	0.1825	0.1863	0.1996	0.3836	0.6569
0.50	0.3636	0.2227	0.1867	0.1823	0.1800	0.1844	0.1834	0.1943	0.3751	0.5908
1.00	0.3145	0.2193	0.1826	0.1804	0.1783	0.1787	0.1811	0.1899	0.3708	0.5123
2.00	0.2578	0.2194	0.1789	0.1790	0.1769	0.1773	0.1793	0.1864	0.3723	0.4196
3.00	0.2205	0.2213	0.1769	0.1784	0.1763	0.1604	0.1647	0.1873	0.3753	0.3583

Apéndice D

Requerimientos de Hardware y Software.

El módulo de evaluación marca los siguientes requerimientos de hardware y software.

Hardware.

- **host** Una computadora compatible con los bus ISA/EISA.
- **memoria** mínimo 640k.
- **Requerimientos de alimentación del EVM** aproximadamente 1 ampere @ 5 volts (5 watts)
- **Material extra.** Discos 3 ½ formateados.

Software:

- **Sistema operativo.** MS-Dos (versión 3.0 o superior). Opcional MS-Windows.
- **Herramientas de Software.** Ensamblador y ligador para la familia de DSPs (C3x/C4x).
- **Archivo requerido.** evmrst.exe para reiniciar al EVM.

Para el desarrollo de este sistema se empleó como sistema operativo a MS-Windows 98 ® y como compilador de C a Borland C++ 5.0 ®