DSP library:
La biblioteca DSP (libdsp-omf.a) proporciona un conjunto de operaciones de procesamiento de señal digital a un programa destinado a la ejecución en un controlador de señal digital (DSC) dsPIC30F.

Peripheral libraries:
Las bibliotecas de periféricos dsPIC (software y hardware) proporcionan funciones y macros para configurar y controlar los periféricos DSC dsPIC30F. También se proporcionan ejemplos de uso en cada capítulo relacionado de este libro.

STANDARD C LIBRARIES (WITH MATH FUNCTIONS)
Se proporciona un conjunto completo de bibliotecas conformes ANSI-89. Los archivos estándar de la biblioteca C son libc-omf.a (escrito por Dinkumware, un líder de la industria) y libm-omf.a (funciones matemáticas, escritas por Microchip).

MPLAB C30 BUILT-IN FUNCTIONS
The MPLAB C30 C compiler contains built-in functions that, to the developer, work like library functions.

DSP LIBRARY

2.1 INTRODUCTION

La Biblioteca DSP proporciona un conjunto de operaciones de procesamiento de señal digital a un programa dirigido a la ejecución en un controlador de señal digital (DSC) dsPIC30F. La biblioteca ha sido diseñada para proporcionarle a usted, el desarrollador de software C, una implementación eficiente de las funciones de procesamiento de señales más comunes. En total, 49 funciones son compatibles con la Biblioteca DSP.

Un objetivo principal de la biblioteca es minimizar el tiempo de ejecución de cada función. Para lograr este objetivo, la biblioteca DSP está escrita principalmente en lenguaje ensamblador optimizado.

2.1.2 C Code Applications

El directorio de instalación del compilador C MPLAB C30 (c: \ pic30_tools) contiene los siguientes subdirectorios con archivos relacionados con la biblioteca:

- lib: archivos de biblioteca / archivo DSP
- src \ dsp: código fuente para las funciones de la biblioteca y un archivo por lotes para reconstruir la biblioteca
- support \ h: archivo de encabezado para la biblioteca DSP

2.2 USING THE DSP LIBRARY

2.2.1 Building with the DSP Library

La creación de una aplicación que utiliza la Biblioteca DSP requiere solo dos archivos: dsp.h y libdsp-omf.a.

dsp.h es un archivo de encabezado que proporciona todos los prototipos de funciones, #defines y typedefs utilizados por la biblioteca.

libdsp-omf.a es el archivo de biblioteca archivado que contiene todos los archivos de objetos individuales para cada función de biblioteca.

Al compilar una aplicación, se debe hacer referencia a dsp.h (usando #include) por todos los archivos fuente que llaman a una función en la Biblioteca DSP o usan sus símbolos o typedefs. Al vincular una aplicación, libdsp-omf.a debe proporcionarse como una entrada al vinculador (utilizando el modificador --library o -l vinculador) de modo que las funciones utilizadas por la aplicación puedan vincularse a la aplicación.

El enlazador colocará las funciones de la biblioteca DSP en una sección de texto especial llamada .libdsp. Esto se puede ver mirando el archivo de mapa generado por el enlazador.

La creación de una aplicación que utiliza la Biblioteca DSP requiere solo dos archivos: dsp.h y libdsp-omf.a. dsp.h es un archivo de encabezado que proporciona todos los prototipos de funciones, #defines y typedefs utilizados por la biblioteca. libdsp-omf.a es el archivo de biblioteca archivado que contiene todos los archivos de objetos individuales para cada función de biblioteca. (Consulte la Sección 1.2 "Bibliotecas específicas de OMF / Módulos StarTup" para obtener más información sobre las bibliotecas específicas de OMF).

Al compilar una aplicación, se debe hacer referencia a dsp.h (usando #include) por todos los archivos fuente que llaman a una función en la Biblioteca DSP o usan sus símbolos o typedefs. Al vincular una aplicación, libdsp-omf.a debe proporcionarse como una entrada al vinculador (utilizando el modificador --library o -l vinculador) de modo que las funciones utilizadas por la aplicación puedan vincularse a la aplicación.

2.2.2 Modelos de memoria

La biblioteca DSP está construida con los modelos de memoria de "código pequeño" y "datos pequeños" para crear la biblioteca más pequeña posible. Dado que varias de las funciones de la biblioteca DSP están escritas en C y hacen uso de la biblioteca de punto flotante del compilador, los archivos de script del enlazador MPLAB C30 colocan las secciones de texto .libm y .libdsp una al lado de la otra. Esto garantiza que la biblioteca DSP pueda usar de manera segura la instrucción RCALL para llamar a las rutinas de punto flotante requeridas en la biblioteca de punto flotante.

2.2.4 Data Types

La biblioteca DSP define un tipo fraccional a partir de un tipo entero:

```
#ifndef fractional
        typedef int fractional;
        #endif
```

El tipo de datos fraccionales se usa para representar datos que tienen 1 bit de signo y 15 bits fraccionarios.

2.2.5 Data Memory Usage

La biblioteca DSP no realiza ninguna asignación de RAM y le deja esta tarea a usted. Si no asigna la cantidad adecuada de memoria y alinea los datos correctamente, se producirán resultados no deseados cuando se ejecute la función.

2.2.6 CORCON Register Usage

el registro CORCON se empuja a la pila. Luego se modifica para realizar correctamente la operación deseada, y finalmente el registro CORCON se saca de la pila para preservar su valor original. Este mecanismo permite que la biblioteca se ejecute lo más correctamente posible, sin interrumpir la configuración de CORCON.

2.2.8 Integrating with Interrupts and an RTOS

Para minimizar el tiempo de ejecución, la biblioteca DSP utiliza bucles DO, bucles REPEAT, direccionamiento de módulo y direccionamiento de bits invertidos.

2.2.9 Rebuilding the DSP Library

Se proporciona un archivo por lotes denominado makedsplib.bat para reconstruir la biblioteca DSP.

2.3.1 Fractional Vector Operations

Un vector fraccionario es una colección de valores numéricos, los elementos del vector, asignados contiguamente en la memoria, con el primer elemento en la dirección de memoria más baja. Se utiliza una palabra de memoria (dos bytes) para almacenar el valor de cada elemento, y esta cantidad debe interpretarse como un número fraccionario representado en el formato de datos 1.15.

Un puntero que se dirige al primer elemento del vector se utiliza como un controlador que proporciona acceso a cada uno de los valores del vector. La dirección del primer elemento se conoce como la dirección base del vector. Como cada elemento del vector tiene 16 bits, la dirección base debe estar alineada con una dirección par.

La disposición unidimensional de un vector se adapta al modelo de almacenamiento de memoria del dispositivo, de modo que se puede acceder al enésimo elemento de un vector de elemento N desde la dirección base BA del vector como:

$BA + 2 (n-1)$, para $1 \leq n \leq N$.

Todas las operaciones de vector fraccionario en esta biblioteca toman como argumento la cardinalidad (número de elementos) de los vectores operandos. Según el valor de este argumento, se hacen los siguientes supuestos:

a) La suma de tamaños de todos los vectores involucrados en una operación particular cae dentro del rango de memoria de datos disponible para el dispositivo objetivo.

b) En el caso de operaciones binarias, las cardinalidades de ambos vectores de operando deben obedecer las reglas del álgebra de vectores (en particular, ver comentarios para las funciones VectorConvolve y VectorCorrelate).

c) El vector de destino debe ser lo suficientemente grande como para aceptar los resultados de una operación.

2.3.3 Additional Remarks

La descripción de las funciones limita su alcance a lo que podría considerarse el uso regular de estas operaciones.

Por ejemplo, al calcular la función VectorMax, la longitud del vector de origen podría ser mayor que numElems. En este caso, la función se usaría para encontrar el valor máximo solo entre los primeros elementos numElems del vector fuente.

Como otro ejemplo, puede estar interesado en reemplazar elementos numElems de un vector de destino ubicado entre N y N + numElems-1, con elementos numElems de un vector fuente ubicado entre los elementos M y M + numElems-1. Entonces, la función VectorCopy podría usarse de la siguiente manera:

```
fractional* dstV[DST_ELEMS] = {...};
fractional* srcV[SRC_ELEMS] = {...};
int n = NUM_ELEMS;
int N = N_PLACE; /* NUM_ELEMS+N ≤ DST_ELEMS */
```

int M = M_PLACE; /* NUM_ELEMS+M ≤ SRC_ELEMS */

fractional* dstVector = dstV+N;

fractional* srcVector = srcV+M;

dstVector = VectorCopy (n, dstVector, srcVector);

También en este contexto, la función VectorZeroPad puede funcionar en su lugar, donde ahora dstV = srcV, numElems es el número de elementos al comienzo del vector de origen para preservar, y numZeros el número de elementos en la cola del vector para establecer en cero.

VECTOR ADD:

VectorAdd agrega el valor de cada elemento en el vector de origen uno con su contraparte en el vector de origen dos, y coloca el resultado en el vector de destino.

Retorna: Puntero a la dirección base del vector de destino.

VectorAdd

Description: `VectorAdd` adds the value of each element in the source one vector with its counterpart in the source two vector, and places the result in the destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorAdd (
   int numElems,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: If the absolute value of `srcV1[n] + srcV2[n]` is larger than $1\text{-}2^{-15}$, this operation results in saturation for the `n`-th element.
This function can be computed in place.
This function can be self applicable.

Source File: `vadd.asm`

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
17 + 3(*numElems*)

VECTOR CONVOLVE:
VectorConvolve calcula la convolución entre dos vectores de origen y almacena el resultado en un vector de destino.

VectorConvolve

Description: `VectorConvolve` computes the convolution between two source vectors, and stores the result in a destination vector. The result is computed as follows:

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k)\text{, for } 0 \le n < M$$

$$y(n) = \sum_{k=n-M+1}^{n} x(k)h(n-k)\text{, for } M \le n < N$$

$$y(n) = \sum_{k=n-M+1}^{N-1} x(k)h(n-k)\text{, for } N \le n < N+M-1$$

where x(k) = source one vector of size N, h(k) = source two vector of size M (with M ≤ N.)

Include: `dsp.h`

Prototype:

```
extern fractional* VectorConvolve (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems1	number of elements in source one vector
numElems2	number of elements in source two vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: The number of elements in the source two vector *must* be less than or equal to the number of elements in the source one vector.
The destination vector *must* already exist, with exactly *numElems1+numElems2-1* number of elements.
This function can be self applicable.

Source File: `vcon.asm`

VectorConvolve (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
58

Cycles (including C-function call and return overheads):
For N = *numElems1*, and M = *numElems2*,

$$28 + 13M + 6\sum_{m=1}^{M} m + (N-M)(7+3M)\text{, for } M < N$$

$$28 + 13M + 6\sum_{m=1}^{M} m\text{, for } M = N$$

VECTOR COPY:
VectorCopy copia los elementos del vector de origen al comienzo de un vector de destino (ya existente), de modo que: dstV [n] = srcV [n], 0 ≤ n <numElems

.

VectorCopy

Description: `VectorCopy` copies the elements of the source vector into the beginning of an (already existing) destination vector, so that: dstV[n] = srcV[n], 0 ≤ n < *numElems*

Include: `dsp.h`

Prototype:

```
extern fractional* VectorCopy (
   int numElems,
   fractional* dstV,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The destination vector *must* already exist. Destination vectors *must* have, at least, *numElems* elements, but could be longer.
This function can be computed in place. See Additional Remarks at the end of the section for comments on this mode of operation.

Source File: `vcopy.asm`

Function Profile: System resources usage:
W0..W3 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
6

Cycles (including C-function call and return overheads):
12 + *numElems*

VECTOR CORRELATE:
VectorCorrelate calcula la correlación entre dos vectores de origen y almacena el resultado en un vector de destino.

VectorCorrelate

Description: `VectorCorrelate` computes the correlation between two source vectors, and stores the result in a destination vector. The result is computed as follows:

$$r(n) = \sum_{k=0}^{N-1} x(k)y(k+n)\text{, for } 0 \le n < N+M-1$$

where x(k) = source one vector of size N, y(k) = source two vector of size M (with M ≤ N.)

Include: `dsp.h`

Prototype:

```
extern fractional* VectorCorrelate (
   int numElems1,
   int numElems2,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems1	number of elements in source one vector
numElems2	number of elements in source two vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: The number of elements in the source two vector *must* be less than or equal to the number of elements in the source one vector.
The destination vector *must* already exist, with exactly *numElems1+numElems2*-1 number of elements.
This function can be self applicable.
This function uses `VectorConvolve`.

Source File: `vcor.asm`

Function Profile: System resources usage:
W0..W7 used, not restored,
plus resources from `VectorConvolve`

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions,
plus `DO`/`REPEAT` instructions from `VectorConvolve`

Program words (24-bit instructions):
14,
plus program words from `VectorConvolve`

Cycles (including C-function call and return overheads):
19 + floor(M/2)*3, with M = *numElems2*,
plus cycles from `VectorConvolve`.

Note: In the description of `VectorConvolve` the number of cycles reported includes 4 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorConvolve` to add to `VectorCorrelate` is 4 less than whatever number is reported for a stand alone `VectorConvolve`.

VECTOR DOTPRODUCT:

VectorDotProduct calcula la suma de los productos entre los elementos correspondientes de los vectores fuente uno y fuente dos.

VectorDotProduct

Description: `VectorDotProduct` computes the sum of the products between corresponding elements of the source one and source two vectors.

Include: `dsp.h`

Prototype:

```
extern fractional VectorDotProduct (
    int numElems,
    fractional* srcV1,
    fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Value of the sum of products.

Remarks: If the absolute value of the sum of products is larger than $1\text{-}2^{-15}$, this operation results in saturation.
This function can be self applicable.

Source File: `vdot.asm`

Function Profile: System resources usage:

W0..W2	used, not restored
W4..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
17 + 3(*numElems*)

VECTOR MAX:

VectorMax encuentra el último elemento en el vector fuente cuyo valor es mayor o igual que cualquier elemento vectorial anterior. Luego, genera ese valor máximo y el índice del elemento máximo.

.

VectorMax

Description: `VectorMax` finds the last element in the source vector whose value is greater than or equal to any previous vector element. Then, it outputs that maximum value and the index of the maximum element.

Include: `dsp.h`

Prototype:
```
extern fractional VectorMax (
   int numElems,
   fractional* srcV,
   int* maxIndex
);
```

Arguments:

numElems number of elements in source vector
srcV pointer to source vector
maxIndex pointer to holder for index of (last) maximum element

Return Value: Maximum value in vector.

Remarks: If `srcV[i] = srcV[j] = maxVal`, and `i < j`, then `*maxIndex = j`.

Source File: `vmax.asm`

VectorMax (Continued)

Function Profile: System resources usage:
W0..W5 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
14
if *numElems* = 1
20 + 8(*numElems*-2)
if `srcV[n]` ≤ `srcV[n+1]`, 0 ≤ n < *numElems*-1
19 + 7(*numElems*-2)
if `srcV[n]` > `srcV[n+1]`, 0 ≤ n < *numElems*-1

VECTOR MIN:
VectorMin encuentra el último elemento en el vector fuente cuyo valor es menor o igual que cualquier elemento vectorial anterior. Luego, genera ese valor mínimo y el índice del elemento mínimo.

VectorMin

Description: `VectorMin` finds the last element in the source vector whose value is less than or equal to any previous vector element. Then, it outputs that minimum value and the index of the minimum element.

Include: `dsp.h`

Prototype:

```
extern fractional VectorMin (
    int numElems,
    fractional* srcV,
    int* minIndex
);
```

Arguments:

numElems	number of elements in source vector
srcV	pointer to source vector
minIndex	pointer to holder for index of (last) minimum element

Return Value: Minimum value in vector.

Remarks: If *srcV*[i] = *srcV*[j] = minVal, and i < j, then *minIndex = j.

Source File: `vmin.asm`

Function Profile:

System resources usage:
W0..W5 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
14
if *numElems* = 1
20 + 8(*numElems*-2)
if *srcV*[n] ≥ *srcV*[n+1], 0 ≤ n < *numElems*-1
19 + 7(*numElems*-2)
if *srcV*[n] < *srcV*[n+1], 0 ≤ n < *numElems*-1

VECTOR MULTIPLY:
VectorMultiply multiplica el valor de cada elemento en el vector de origen uno con su contraparte en el vector de origen dos, y coloca el resultado en el elemento correspondiente del vector de destino.

VectorMultiply

Description: `VectorMultiply` multiplies the value of each element in source one vector with its counterpart in source two vector, and places the result in the corresponding element of destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorMultiply (
    int numElems,
    fractional* dstV,
    fractional* srcV1,
    fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV1	pointer to source one vector
srcV2	pointer to source two vector

Return Value: Pointer to base address of destination vector.

Remarks: This operation is also known as vector element-by-element multiplication.
This function can be computed in place.
This function can be self applicable.

Source File: `vmul.asm`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
17 + 4(*numElems*)

VECTOR NEGATE:
VectorNegate niega (cambia el signo de) los valores de los elementos en el vector de origen y los coloca en el vector de destino.

VectorNegate

Description: `VectorNegate` negates (changes the sign of) the values of the elements in the source vector, and places them in the destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorNeg (
   int numElems,
   fractional* dstV,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The negated value of 0x8000 is set to 0x7FFF.
This function can be computed in place.

Source File: `vneg.asm`

VectorNegate (Continued)

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
16

Cycles (including C-function call and return overheads):
19 + 4(*numElems*)

VECTOR POWER:
VectorPower calcula la potencia de un vector fuente como la suma de los cuadrados de sus elementos.

VectorPower

Description: `VectorPower` computes the power of a source vector as the sum of the squares of its elements.

Include: `dsp.h`

Prototype:

```
extern fractional VectorPower (
   int numElems,
   fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
srcV	pointer to source vector

Return Value: Value of the vector's power (sum of squares).

Remarks: If the absolute value of the sum of squares is larger than $1\text{-}2^{-15}$, this operation results in saturation
This function can be self applicable.

Source File: `vpow.asm`

Function Profile: System resources usage:

W0..W2	used, not restored
W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
12

Cycles (including C-function call and return overheads):
16 + 2(*numElems*)

VECTOR SCALE:
VectorScale escala (multiplica) los valores de todos los elementos en el vector de origen por un valor de escala, y coloca el resultado en el vector de destino.

VectorScale

Description: `VectorScale` scales (multiplies) the values of all the elements in the source vector by a scale value, and places the result in the destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorScale (
   int numElems,
   fractional* dstV,
   fractional* srcV,
   fractional sclVal
);
```

Arguments:

numElems	number of elements in source vector
dstV	pointer to destination vector
srcV	pointer to source vector
sclVal	value by which to scale vector elements

Return Value: Pointer to base address of destination vector.

Remarks: `sclVal` must be a fractional number in 1.15 format.
This function can be computed in place.

Source File: `vscl.asm`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
18 + 3(*numElems*)

VECTOR SUBTRACT:
VectorSubtract resta el valor de cada elemento en el vector de origen dos de su contraparte en el vector de origen y coloca el resultado en el vector de destino.

VectorSubtract

Description: `VectorSubtract` subtracts the value of each element in the source two vector from its counterpart in the source one vector, and places the result in the destination vector.

Include: `dsp.h`

Prototype:

```
extern fractional* VectorSubtract (
   int numElems,
   fractional* dstV,
   fractional* srcV1,
   fractional* srcV2
);
```

Arguments:

numElems	number of elements in source vectors
dstV	pointer to destination vector
srcV1	pointer to source one vector (minuend)
srcV2	pointer to source two vector (subtrahend)

Return Value: Pointer to base address of destination vector.

Remarks: If the absolute value of `srcV1[n] - srcV2[n]` is larger than $1\text{-}2^{-15}$, this operation results in saturation for the `n`-th element.
This function can be computed in place.
This function can be self applicable.

VectorSubtract (Continued)

Source File: `vsub.asm`

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
17 + 4(*numElems*)

VECTOR ZERO PAD:
VectorZeroPad copia el vector de origen en el comienzo del vector de destino (ya existente) y luego llena con ceros los elementos numZeros restantes del vector de destino:

VectorZeroPad

Description: `VectorZeroPad` copies the source vector into the beginning of the (already existing) destination vector, and then fills with zeros the remaining `numZeros` elements of destination vector:
$dstV[n] = srcV[n]$, $0 \le n < numElems$
$dstV[n] = 0$, $numElems \le n < numElems + numZeros$

Include: `dsp.h`

Prototype:

```
extern fractional* VectorZeroPad (
    int numElems,
    int numZeros,
    fractional* dstV,
    fractional* srcV
);
```

Arguments:

numElems	number of elements in source vector
numZeros	number of elements to fill with zeros at the tail of destination vector
dstV	pointer to destination vector
srcV	pointer to source vector

Return Value: Pointer to base address of destination vector.

Remarks: The destination vector *must* already exist, with exactly *numElems+numZeros* number of elements.
This function can be computed in place. See Additional Remarks at the beginning of the section for comments on this mode of operation.
This function uses `VectorCopy`.

Source File: `vzpad.asm`

VectorZeroPad (Continued)

Function Profile: System resources usage:
W0..W6 used, not restored
plus resources from `VectorCopy`

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions
plus `DO`/`REPEAT` from `VectorCopy`

Program words (24-bit instructions):
13,
plus program words from `VectorCopy`

Cycles (including C-function call and return overheads):
18 + *numZeros*
plus cycles from `VectorCopy`.

Note: In the description of `VectorCopy`, the number of cycles reported includes 3 cycles of C-function call overhead. Thus, the number of actual cycles from `VectorCopy` to add to `VectorCorrelate` is 3 less than whatever number is reported for a stand alone `VectorCopy`.

MATRICES

MatrixAdd agrega el valor de cada elemento en la matriz de origen uno con su contraparte en la matriz de origen dos, y coloca el resultado en la matriz de destino.

MatrixAdd

Description: `MatrixAdd` adds the value of each element in the source one matrix with its counterpart in the source two matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern fractional* MatrixAdd (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM1,
    fractional* srcM2
);
```

Arguments:

numRows	number of rows in source matrices
numCols	number of columns in source matrices
dstM	pointer to destination matrix
srcM1	pointer to source one matrix
srcM2	pointer to source two matrix

Return Value: Pointer to base address of destination matrix.

Remarks: If the absolute value of *srcM1*[r][c]+*srcM2*[r][c] is larger than $1\text{-}2^{-15}$, this operation results in saturation for the (r,c)-th element.
This function can be computed in place.
This function can be self applicable.

Source File: `madd.asm`

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

DO and REPEAT instruction usage:
1 level DO instructions
no REPEAT instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
20 + 3(*numRows***numCols*)

MATRIXSCALE:
MatrixScale escala (multiplica) los valores de todos los elementos en la matriz fuente por un valor de escala, y coloca el resultado en la matriz de destino

MatrixScale

Description: `MatrixScale` scales (multiplies) the values of all elements in the source matrix by a scale value, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern fractional* MatrixScale (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM,
    fractional sclVal
);
```

Arguments:

numRows	number of rows in source matrix
numCols	number of columns in source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix
sclVal	value by which to scale matrix elements

Return Value: Pointer to base address of destination matrix.

Remarks: This function can be computed in place.

Source File: `mscl.asm`

Function Profile: System resources usage:

W0..W5	used, not restored
ACCA	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
20 + 3(*numRows***numCols*)

MATRIX SUBTRACT:
MatrixSubtract resta el valor de cada elemento en la matriz de origen dos de su contraparte en la matriz de origen uno, y coloca el resultado en la matriz de destino.

MatrixSubtract

Description: `MatrixSubtract` subtracts the value of each element in the source two matrix from its counterpart in the source one matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern fractional* MatrixSubtract (
   int numRows,
   int numCols,
   fractional* dstM,
   fractional* srcM1,
   fractional* srcM2
);
```

Arguments:

numRows	number of rows in source matrix(ces)
numCols	number of columns in source matrix(ces)
dstM	pointer to destination matrix
srcM1	pointer to source one matrix (minuend)
srcM2	pointer to source two matrix (subtrahend)

Return Value: Pointer to base address of destination matrix.

MatrixSubtract (Continued)

Remarks: If the absolute value of `srcM1[r][c]-srcM2[r][c]` is larger than $1\text{-}2^{-15}$, this operation results in saturation for the `(r,c)`-th element.
This function can be computed in place.
This function can be self applicable.

Source File: `msub.asm`

Function Profile: System resources usage:

W0..W4	used, not restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
15

Cycles (including C-function call and return overheads):
20 + 4(*numRows***numCols*)

MATRIXTRASPOSE:
MatrixTranspose transpone las filas por las columnas en la matriz de origen y coloca el resultado en la matriz de destino. En efecto: dstM [i] [j] = srcM [j] [i],
0 ≤ i <numRows, 0 ≤ j <numCols.

MatrixTranspose

Description: `MatrixTranspose` transposes the rows by the columns in the source matrix, and places the result in destination matrix. In effect:
dstM[i][j] = *srcM*[j][i],
0 ≤ i < *numRows*, 0 ≤ j < *numCols*.

Include: `dsp.h`

Prototype:

```
extern fractional* MatrixTranspose (
    int numRows,
    int numCols,
    fractional* dstM,
    fractional* srcM
);
```

Arguments:

numRows	number of rows in source matrix
numCols	number of columns in source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix

Return Value: Pointer to base address of destination matrix.

Remarks: If the source matrix is square, this function can be computed in place. See Additional Remarks at the beginning of the section for comments on this mode of operation.

Source File: `mtrp.asm`

Function Profile: System resources usage:
W0..W5 used, not restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
14

Cycles (including C-function call and return overheads):
16 + *numCols**(6 + (*numRows*-1)*3)

MATRIXINVERT:
MatrixInvert calcula el inverso de la matriz de origen y coloca el resultado en la matriz de destino.

MatrixInvert

Description: `MatrixInvert` computes the inverse of the source matrix, and places the result in the destination matrix.

Include: `dsp.h`

Prototype:

```
extern float* MatrixInvert (
    int numRowsCols,
    float* dstM,
    float* srcM,
    float* pivotFlag,
    int* swappedRows,
    int* swappedCols
);
```

Arguments:

numRowCols	number of rows and columns in (square) source matrix
dstM	pointer to destination matrix
srcM	pointer to source matrix

Required for internal use:

pivotFlag	pointer to a length numRowsCols vector
swappedRows	pointer to a length numRowsCols vector
swappedCols	pointer to a length numRowsCols vector

Return Value: Pointer to base address of destination matrix, or `NULL` if source matrix is singular.

Remarks: Even though the vectors *pivotFlag*, *swappedRows*, and *swappedCols*, are for internal use only, they must be allocated prior to calling this function.
If source matrix is singular (determinant equal to zero) the matrix does not have an inverse. In this case the function returns `NULL`.
This function can be computed in place.

Source File: `minv.asm` (assembled from C-code)

Function Profile: System resources usage:

W0..W7	used, not restored
W8, W14	saved, used, restored

`DO` and `REPEAT` instruction usage:
None

Program words (24-bit instructions):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

Cycles (including C-function call and return overheads):
See the file “readme.txt” in pic30_tools\src\dsp for this information.

FIRStruct:
FIRStruct describe la estructura del filtro para cualquiera de los filtros FIR.

FIRStruct

Structure: `FIRStruct` describes the filter structure for any of the FIR filters.

Include: `dsp.h`

Declaration:

```
typedef struct {
  int numCoeffs;
  fractional* coeffsBase;
  fractional* coeffsEnd;
  int coeffsPage;
  fractional* delayBase;
  fractional* delayEnd;
  fractional* delay;
} FIRStruct;
```

Parameters:

numCoeffs	number of coefficients in filter (also M)
coeffsBase	base address for filter coefficients (also h)
coeffsEnd	end address for filter coefficients
coeffsPage	coefficients buffer page number
delayBase	base address for delay buffer
delayEnd	end address for delay buffer
delay	current value of delay pointer (also d)

Remarks:

Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \leq m < M$, either within X-Data or program memory.

Delay buffer d[m], defined in $0 \leq m < M$, *only* in Y-Data.

If coefficients are stored in X-Data space, *coeffsBase* points to the actual address where coefficients are allocated. If coefficients are stored in program memory, *coeffsBase* is the offset from the program page boundary containing the coefficients to the address in the page where coefficients are allocated. This latter value can be calculated using the inline assembly operator `psvoffset()`.

coeffsEnd is the address in X-Data space (or offset if in program memory) of the last byte of the filter coefficients buffer.

If coefficients are stored in X-Data space, *coeffsPage* must be set to 0xFF00 (defined value `COEFFS_IN_DATA`). If coefficients are stored in program memory, it is the program page number containing the coefficients. This latter value can be calculated using the inline assembly operator `psvpage()`.

delayBase points to the actual address where the delay buffer is allocated.

delayEnd is the address of the last byte of the filter delay buffer.

FIRStruct (Continued)

When the coefficients and delay buffers are implemented as circular increasing modulo buffers, both *coeffsBase* and *delayBase* ***must*** be aligned to a 'zero' power of two address (*coeffsEnd* and *delayEnd* are odd addresses). Whether these buffers are implemented as circular increasing modulo buffers or not is indicated in the remarks section of each FIR filter function description.

When the coefficients and delay buffers are not implemented as circular (increasing) modulo buffers, *coeffsBase* and *delayBase* ***do not need to*** be aligned to a 'zero' power of two address, and the values of *coeffsEnd* and *delayEnd* are ignored within the particular FIR Filter function implementation.

FIR:
FIR aplica un filtro FIR a la secuencia de muestras de origen, coloca los resultados en la secuencia de muestras de destino y actualiza los valores de retraso.

FIR

Description: `FIR` applies an FIR filter to the sequence of source samples, places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:

```
extern fractional* FIR (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

Arguments:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \leq m < M$, implemented as a circular increasing modulo buffer.
Delay, d[m], defined in $0 \leq m < M$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \leq n < N$.
Destination samples, y[n], defined in $0 \leq n < N$.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `fir.asm`

FIR (Continued)

Function Profile: System resources usage:

W0..W6	used, not restored
W8, W10	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
55

Cycles (including C-function call and return overheads):
53 + N(4+M), or
56 + N(8+M) if coefficients in P memory.

FIRDecimate:
FIRDecimate diezma la secuencia de muestras de origen a una velocidad de R a 1; o equivalentemente, disminuye la señal por un factor de R.

FIRDecimate

Description: `FIRDecimate` decimates the sequence of source samples at a rate of R to 1; or equivalently, it downsamples the signal by a factor of R.
Effectively,
y[n] = x[Rn].
To diminish the effect of aliasing, the source samples are first filtered and then downsampled. The decimated results are stored in the sequence of destination samples, and the delay values updated.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRDecimate (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   int rate
);
```

Arguments:

numSamps	number of *output* samples (also N, N = Rp, p integer)
dstSamp	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
rate	rate of decimation (downsampling factor, also R)

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M, with M an integer multiple of R.
Coefficients, h[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \le n < NR$.
Destination samples, y[n], defined in $0 \le n < N$.
(See also `FIRStruct`, `FIRStructInit`, and `FIRDelayInit`.)

Source File: `firdecim.asm`

FIRDecimate (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
1 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
48

Cycles (including C-function call and return overheads):
45 + N(10 + 2M), or
48 + N(13 + 2M) if coefficients in P memory.

FIRDelayInit:
FIRDelayInit inicializa a cero los valores de retraso en una estructura de filtro FIRStruct.

FIRDelayInit

Description:	`FIRDelayInit` initializes to zero the delay values in an `FIRStruct` filter structure.
Include:	`dsp.h`

Prototype:	`extern void FIRDelayInit (`
	`FIRStruct* filter`
	`);`

Arguments:	*`filter`* pointer to `FIRStruct` filter structure.
Remarks:	See description of `FIRStruct` structure above. **Note:** FIR interpolator's delay is initialized by function `FIRInterpDelayInit`.
Source File:	`firdelay.asm`
Function Profile:	System resources usage: W0..W2 used, not restored `DO` and `REPEAT` instruction usage: no `DO` instructions 1 level `REPEAT` instructions Program words (24-bit instructions): 7 Cycles (including C-function call and return overheads): 11 + M

<u>FIRInterpolate:</u>

FIRInterpolate interpola la secuencia de muestras de origen a una velocidad de 1 a R; o equivalentemente, sube la muestra de la señal por un factor de R. Efectivamente,
y [n] = x [n / R].
Para disminuir el efecto del aliasing, las muestras fuente se muestrean primero y luego se filtran. Los resultados interpolados se almacenan en la secuencia de muestras de destino y se actualizan los valores de retraso.

FIRInterpolate

Description: `FIRInterpolate` interpolates the sequence of source samples at a rate of 1 to R; or equivalently, it upsamples the signal by a factor of R. Effectively,
y[n] = x[n/R].
To diminish the effect of aliasing, the source samples are first upsampled and then filtered. The interpolated results are stored in the sequence of destination samples, and the delay values updated.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRInterpolate (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   int rate
);
```

Arguments:

numSamps	number of input samples (also N, N = Rp, p integer)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
rate	rate of interpolation (upsampling factor, also R)

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M, with M an integer multiple of R.
Coefficients, h[m], defined in $0 \le m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \le m < M/R$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < NR$.
(See also `FIRStruct`, `FIRStructInit`, and `FIRInterpDelayInit`.)

Source File: `firinter.asm`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
63

Cycles (including C-function call and return overheads):
45 + 6(M/R) + N(14 + M/R + 3M + 5R), or
48 + 6(M/R) + N(14 + M/R + 4M + 5R) if coefficients in P memory.

FIRInterpDelayInit inicializa a cero los valores de retraso en una estructura de filtro FIRStruct, optimizado para su uso con un filtro de interpolación FIR.

FIRInterpDelayInit

Description: `FIRInterpDelayInit` initializes to zero the delay values in an `FIRStruct` filter structure, optimized for use with an FIR interpolating filter.

Include: `dsp.h`

Prototype:

```
extern void FIRDelayInit (
   FIRStruct* filter,
   int rate
);
```

Arguments:

filter	pointer to `FIRStruct` filter structure
rate	rate of interpolation (upsampling factor, also R)

Remarks: Delay, d[m], defined in $0 \leq m < M/R$, with M the number of filter coefficients in the interpolator.
See description of `FIRStruct` structure above.

Source File: `firintdl.asm`

Function Profile: System resources usage:
W0..W4 used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
13

Cycles (including C-function call and return overheads):
10 + 7M/R

FIRLattice

Description: `FIRLattice` uses a lattice structure implementation to apply an FIR filter to the sequence of source samples. It then places the results in the sequence of destination samples, and updates the delay values.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRLattice (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter
);
```

Arguments:

numSamps	number of input samples to filter (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure

Return Value: Pointer to base address of destination samples.

Remarks: Number of coefficients in filter is M.
Lattice coefficients, k[m], defined in $0 \leq m < M$, not implemented as a circular modulo buffer.
Delay, d[m], defined in $0 \leq m < M$, not implemented as a circular modulo buffer.
Source samples, x[n], defined in $0 \leq n < N$.
Destination samples, y[n], defined in $0 \leq n < N$.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlatt.asm`

FIRLattice:
FIRLattice utiliza una implementación de estructura reticular para aplicar un filtro FIR a la secuencia de muestras fuente. Luego coloca los resultados en la secuencia de muestras de destino y actualiza los valores de retraso.

FIRLattice (Continued)

Function Profile:

System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
PSVPAG	saved, used, restored (only if coefficients in P memory)

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
50

Cycles (including C-function call and return overheads):
41 + N(4 + 7M)
44 + N(4 + 8M) if coefficients in P memory

FIRLMS

FIRLMS aplica un filtro FIR adaptable a la secuencia de muestras de origen, almacena los resultados en la secuencia de muestras de destino y actualiza los valores de retraso.
Los coeficientes del filtro también se actualizan, muestra por muestra, utilizando un algoritmo de Mínimo Cuadrado Medio aplicado de acuerdo con los valores de las muestras de referencia.

FIRLMS

Description: `FIRLMS` applies an adaptive FIR filter to the sequence of source samples, stores the results in the sequence of destination samples, and updates the delay values.
The filter coefficients are also updated, at a sample-per-sample basis, using a Least Mean Square algorithm applied according to the values of the reference samples.

Include: `dsp.h`

Prototype:

```
extern fractional* FIRLMS (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   fractional* refSamps,
   fractional muVal
);
```

Arguments:

numSamps	number of input samples (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
refSamps	pointer to reference samples (also r)
muVal	adapting factor (also mu)

Return Value: Pointer to base address of destination samples.

FIRLMS (Continued)

Remarks: Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \le m < M$, implemented as a circular increasing modulo buffer.
delay, d[m], defined in $0 \le m < M-1$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \le n < N$.
Reference samples, r[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < N$.
Adaptation:
h_m[n] = h_m[n-1] + mu*(r[n] - y[n])*x[n-m],
for $0 \le n < N$, $0 \le m < M$.
The operation could result in saturation if the absolute value of (r[n] - y[n]) is greater than or equal to one.
Filter coefficients *must not* be allocated in program memory, because in that case their values could not be adapted. If filter coefficients are detected as allocated in program memory the function returns `NULL`.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlms.asm`

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W12	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
76

Cycles (including C-function call and return overheads):
61 + N(13 + 5M)

FIRLMSNorm aplica un filtro FIR adaptativo a la secuencia de muestras de origen, almacena los resultados en la secuencia de muestras de destino y actualiza los valores de retraso.
Los coeficientes del filtro también se actualizan, muestra por muestra, utilizando un algoritmo de mínimo cuadrado medio normalizado aplicado de acuerdo con los valores de las muestras de referencia.

FIRLMSNorm

Description: `FIRLMSNorm` applies an adaptive FIR filter to the sequence of source samples, stores the results in the sequence of destination samples, and updates the delay values.
The filter coefficients are also updated, at a sample-per-sample basis, using a Normalized Least Mean Square algorithm applied according to the values of the reference samples.

Include: `dsp.h`

FIRLMSNorm (Continued)

Prototype:

```
extern fractional* FIRLMSNorm (
   int numSamps,
   fractional* dstSamps,
   fractional* srcSamps,
   FIRStruct* filter,
   fractional* refSamps,
   fractional muVal,
   fractional* energyEstimate
);
```

Arguments:

numSamps	number of input samples (also N)
dstSamps	pointer to destination samples (also y)
srcSamps	pointer to source samples (also x)
filter	pointer to `FIRStruct` filter structure
refSamps	pointer to reference samples (also r)
muVal	adapting factor (also mu)
energyEstimate	estimated energy value for the last M input signal samples, with M the number of filter coefficients

Return Value: Pointer to base address of destination samples.

Remarks:

Number of coefficients in filter is M.
Coefficients, h[m], defined in $0 \le m < M$, implemented as a circular increasing modulo buffer.
delay, d[m], defined in $0 \le m < M$, implemented as a circular increasing modulo buffer.
Source samples, x[n], defined in $0 \le n < N$.
Reference samples, r[n], defined in $0 \le n < N$.
Destination samples, y[n], defined in $0 \le n < N$.
Adaptation:

h_m[n] = h_m[n-1] + nu[n]*(r[n] - y[n])*x[n-m],
for $0 \le n < N$, $0 \le m < M$,
where nu[n] = mu/(mu+E[n])
with $E[n]=E[n-1]+(x[n])^2-(x[n-M+1])^2$ an estimate of input signal energy.

On start up, `energyEstimate` should be initialized to the value of E[-1] (zero the first time the filter is invoked). Upon return, `energyEstimate` is updated to the value E[N-1] (which may be used as the start up value for a subsequent function call if filtering an extension of the input signal).
The operation could result in saturation if the absolute value of (r[n] - y[n]) is greater than or equal to one.
Note: Another expression for the energy estimate is:
$E[n] = (x[n])^2 + (x[n-1)^2 + \ldots + (x[n-M+2])^2$.
Thus, to avoid saturation while computing the estimate, the input sample values should be bound so that

$$\sum_{m=0}^{-M+2} (x[n+m])^2 < 1 \text{, for } 0 \le n < N.$$

Filter coefficients *must not* be allocated in program memory, because in that case their values could not be adapted. If filter coefficients are detected as allocated in program memory the function returns `NULL`.
(See also `FIRStruct`, `FIRStructInit` and `FIRDelayInit`.)

Source File: `firlmsn.asm`

FIRLMSNorm (Continued)

Function Profile: System resources usage:

W0..W7	used, not restored
W8..W13	saved, used, restored
ACCA	used, not restored
ACCB	used, not restored
CORCON	saved, used, restored
MODCON	saved, used, restored
XMODSTRT	saved, used, restored
XMODEND	saved, used, restored
YMODSTRT	saved, used, restored

`DO` and `REPEAT` instruction usage:
2 level `DO` instructions
1 level `REPEAT` instructions

Program words (24-bit instructions):
91

Cycles (including C-function call and return overheads):
66 + N(49 + 5M)

FIRStructInit

Description: `FIRStructInit` initializes the values of the parameters in an `FIRStruct` FIR Filter structure.

Include: `dsp.h`

Prototype:

```
extern void FIRStructInit (
  FIRStruct* filter,
  int numCoeffs,
  fractional* coeffsBase,
  int coeffsPage,
  fractional* delayBase
);
```

Arguments:

filter	pointer to `FIRStruct` filter structure
numCoeffs	number of coefficients in filter (also M)
coeffsBase	base address for filter coefficients (also h)
coeffsPage	coefficient buffer page number
delayBase	base address for delay buffer

Remarks: See description of `FIRStruct` structure above.
Upon completion, `FIRStructInit` initializes the `coeffsEnd` and `delayEnd` pointers accordingly. Also, `delay` is set equal to `delayBase`.

Source File: `firinit.asm`

Function Profile: System resources usage:

W0..W5	used, not restored

`DO` and `REPEAT` instruction usage:
no `DO` instructions
no `REPEAT` instructions

Program words (24-bit instructions):
10

Cycles (including C-function call and return overheads):
19

FIRStructInit inicializa los valores de los parámetros en una estructura FIRStruct FIR Filter.