



Guía N° 6: *Procesamiento de señales con dsPIC*

Objetivo:

Utilizar el controlador de señal dsPIC33 en aplicaciones de procesamiento de señal

Fecha de entrega:

04/10/2019

Formato de entrega:

Esta guía se resuelve sobre un único proyecto base, en el cual cada ejercicio se corresponde con porciones de código, que serán comentadas/descomentadas para evaluar cada punto. Cada porción debe estar indicada con los comentarios necesarios. Se deberán incluir los scripts de Matlab/Octave necesarios.

Toda la carpeta del proyecto deberá comprimirse en un archivo .zip o .rar y enviarse por correo con título *Entrega Guía 6*.

Bibliografía recomendada:

Título: Programming dsPIC MCU in C

Authors: Zoran Milivojević, Djordje Šaponjić

Editorial: MikroElektronika

Disponible online en: <http://www.mikroe.com/products/view/266/programming-dspic-mcu-in-c/>

[The Scientist and Engineer's Guide to Digital Signal Processing](#)

[dsPIC Language Tools Libraries - Microchip Technology Inc](#)

[Manual de Referencia de la placa de desarrollo](#)

[Página web del dispositivo](#)

[Guía de uso para el bootloader de la placa nueva](#)



Configuración de la librería DSP y uso general

Librería

Ubicarse sobre la ventana de exploración de proyecto, hacer click derecho sobre la carpeta **Libraries** y seleccionar **Add Library/Object File**. Dirigirse a la carpeta de instalación del compilador XC16 y dentro de la carpeta *lib*, seleccionar el archivo *libdsp-elf.a*. En aquellos archivos donde se utilicen funciones de librería, declarar la librería mediante la inclusión de la línea `#include <dsp.h>` al comienzo. La descripción de uso de cada función de librería puede encontrarse en el manual [dsPIC Language Tool Libraries](#).

Tipos de datos

La librería dsp utiliza principalmente dos tipos de datos: **fractional** y **fractcomplex**.

El tipo de datos **fractional** se utiliza para representar **números en punto fijo** con definición S16.15, es decir **16 bits de precisión y 15 de parte fraccional**. Con este tipo de datos se pueden representar números entre -1 y $1-2^{-15}$. Es importante destacar que tanto los valores de los coeficientes como la señal de entrada debe normalizarse para adecuarse a este rango. Desde el punto de vista de la implementación, el tipo **fractional** no es más que **un entero de 16 bits**, por lo que una constante de este tipo puede escribirse como un número de 16 bits, en formato decimal o hexadecimal. Por ejemplo:

```
fractional cte = 0x8000; // Valor de la constante: -1
```

Desde MATLAB, se puede utilizar objetos **fi** para realizar la conversión de punto flotante a punto fijo. Para el formato S16.15, un ejemplo de uso es el siguiente:

```
x_fx = fi(x,1,16,15); % Convierte el número x en S16.15
s     = ['0x' x_fx.hex]; % Crea una cadena y antepone 0x
disp(s);                % Muestra la cadena por consola
```

El tipo de datos **fractcomplex** está compuesto **por dos elementos de tipo fractional**, el primero representando la **parte real y el segundo la parte imaginaria de una variable compleja, de acuerdo a la siguiente definición:**

```
typedef struct {
    fractional real;
    fractional imag;
} fractcomplex;
```

Para escribir una constante compleja, se deben escribir **dos valores de 16 bits**



consecutivos, el primero para la parte real y el segundo para la parte compleja:

```
fractcomplex cte_compleja = {0x4000,0x4000}; // 0.5 real, 0.5 imag
```

Uso de Memoria

Muchas de las funciones requieren que las variables estén ubicadas en porciones específicas de la memoria, y con algún alineamiento particular. El alineamiento implica que la dirección de inicio del vector sea múltiplo de una constante K especificada. Entonces si el algoritmo pide alineamiento K para un vector v , la dirección de inicio del vector solo puede tomar valores en el conjunto $K, 2K, 3K, \text{etc.}$

Por otro lado, la mayoría de los algoritmos DSP utilizan una instrucción particular denominada MAC (Multiply & Accumulate). Esta instrucción toma dos muestras, las multiplica, acumula el resultado e incrementa el puntero de memoria a las siguientes dos muestras, todo en un ciclo de instrucción. Para que esto sea posible, es necesario que ambas muestras provengan de secciones distintas de la memoria, conocidas como sección X y sección Y.

Se puede especificar la ubicación y el alineamiento de un vector con las siguientes directivas de compilador:

```
fractional coefs[10] __attribute__((space(xmemory), aligned (K))); // Memoria X,
                                                                    // Alineación K
fractional delay[10] __attribute__((space(ymemory), aligned (L))); // Memoria Y,
                                                                    // Alineación L
```

Los vectores pueden inicializarse en la misma declaración, luego de las directivas.

En algunas situaciones, el compilador puede intentar ubicar la sección en la porción de memoria baja (near) la cual no está disponible en el dspPIC33FJ, generando un error de enlace (link).

En dichas situaciones se puede indicar que ubique la sección en la memoria alta (far).

Esto se puede hacer agregando el atributo *far*. Como se muestra a continuación:

```
fractional delay[10] __attribute__((space(ymemory), far))
```

Procesamiento en tiempo real, uso de los buffers DMA en modo ping pong

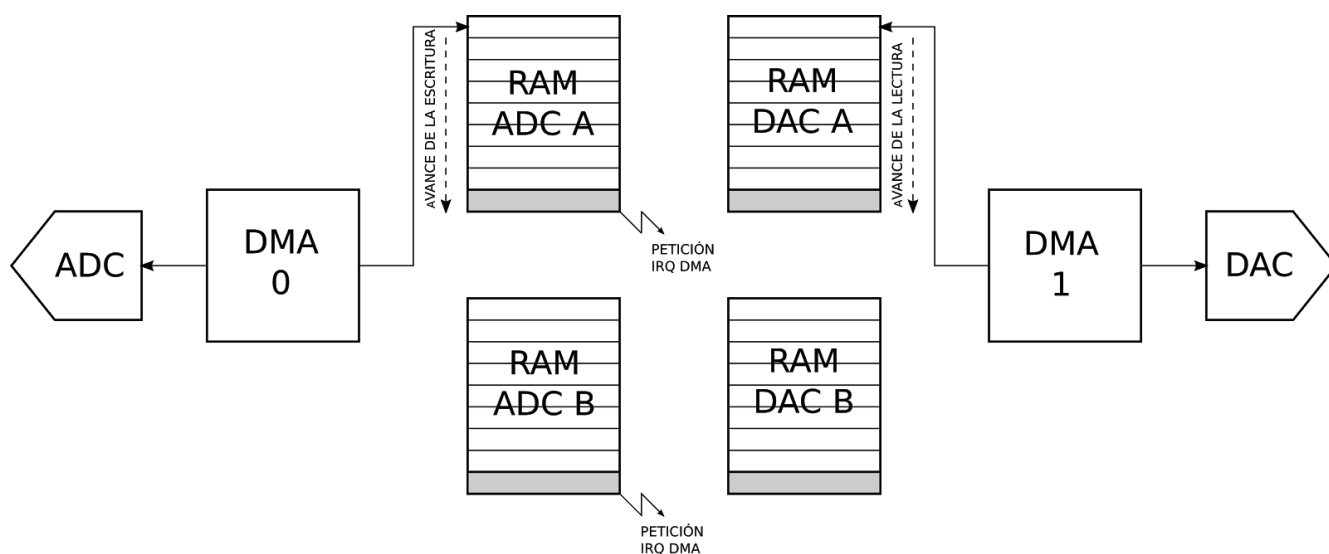
Un problema habitual en el procesamiento de tiempo real es la operación con muestras que necesitan ser actualizadas constantemente. El uso de un controlador DMA nos permite realizar la actualización automática de las muestras entre periféricos y la memoria pero cuando se necesitan procesar estas muestras mediante el CPU, ahora es necesario sincronizar el acceso del CPU con el controlador DMA para garantizar que las muestras usadas en el procesamiento provengan de un bloque continuo de datos. Una manera de resolver este problema es dividir la memoria en dos zonas, A y B, de



manera que el controlador DMA primero escribe/lee sobre la memoria A y luego lo hace sobre la memoria B, **indicando con una interrupción cuando completa cada bloque**. De este modo, el CPU puede trabajar sobre el bloque que **no está siendo accedido por DMA**, con la garantía de que todas las muestras son continuas.

Esta configuración de dos zonas es la que se denomina modo Ping-Pong. Resumiendo: **Se utiliza el modo Ping-Pong para poder leer y escribir sobre un bloque de muestras con la seguridad de que no va a ser leído ni escrito por el controlador DMA.**

Como se observa en la siguiente figura, el sistema está configurado con dos canales DMA, 0 y 1 y ambos en modo Ping-Pong. El canal DMA 0 toma una muestra con cada adquisición del ADC y la **almacena en el búffer A ADC** empezando desde la posición 0. Luego de completar la escritura de la última posición del búffer A del ADC, **genera una petición de interrupción**. Es en este punto donde el CPU puede procesar todas las muestras de la memoria A con la garantía de disponer de muestras continuas. Al mismo tiempo, el canal **DMA 1 toma una muestra del búffer A del DAC y se la envía al DAC**. Como ambos procesos **tiene la misma frecuencia de muestreo**, las direcciones de memoria se van a incrementar al mismo ritmo. Por lo tanto, el **CPU puede escribir el resultado del procesamiento sobre el búffer A del DAC con la seguridad de que el controlador DMA del DAC está accediendo a búffer B del DAC.**



1. Eliminación de interferencias con filtro FIR

- 1.1. Crear desde Octave/MATLAB un filtro FIR rechaza banda que permita eliminar una interferencia de 3KHz de una señal de audio capturada mediante el ADC con un mínimo de 45dB de atenuación y con una banda de rechazo de entre 200 y 400Hz de ancho y menos de 400 taps. La frecuencia de muestreo será igual a la utilizada en la configuración del ADC de la guía 5. Podrá utilizarse el método de filtro pasa-bajos y pasa alto en paralelo o inversión espectral de la



respuesta de un pasa-banda. Los coeficientes del filtro deberán cuantizarse con definición S16.15. Dibujar la respuesta antes y después de la cuantización.

- 1.2. Implementar el filtro anterior en un programa de dsPIC, de manera que los datos recibidos por el ADC sean filtrados mediante el FIR y luego se guarden en los búffer DMA del DAC, de manera de obtener a la salida del DAC la señal sin la interferencia de 3KHz.

Para utilizar el filtro FIR de la librería se deben seguir los siguientes pasos:

1- Declarar una variable global de tipo *FIRStruct*, la cual ya está definida en la librería DSP de la siguiente manera:

```
typedef struct {  
    int numCoeffs ;           // Número de coeficientes del filtro (M)  
    fractional* coeffsBase;   // Dirección inicial del vector de  
                               // coeficientes  
    fractional* coeffsEnd;    // Dirección del último byte del vector de  
                               // coeficientes (impar)  
    int coeffsPage;           // Página de ubicación de coeficientes  
                               // (0xFF00 para RAM)  
    fractional* delayBase;    // Dirección inicial del vector de retardos  
    fractional* delayEnd;     // Dirección del último byte del vector de  
                               // retardos (impar)  
    fractional* delay;        // Puntero a la posición actual del vector de  
                               // retardos  
} FIRStruct;
```

2- Declarar e inicializar un vector de coeficientes de M taps, global, de tipo *fractional*, ubicado en memoria X y alineado a la menor potencia de 2 mayor que $2 \cdot M$. Los valores de inicialización se obtendrán del punto anterior.

3- Declarar un vector de retardos de longitud M , global, de tipo *fractional*, ubicado en memoria Y y alineado a la menor potencia de 2 mayor que $2 \cdot M$.

4- Declarar un vector para almacenamiento temporal, global y de tipo *fractional*, con la misma cantidad de elementos que la utilizada en cada búfer DMA

5- Dentro del código de inicialización del programa, asignar cada campo de la estructura *FIRStruct*, utilizando el operador punto, donde:

- *numCoeffs* es la cantidad de coeficientes.
- *coeffsBase* es la dirección al inicio del vector de coeficientes, o directamente, el nombre del arreglo.
- *coeffsEnd* es la dirección al último byte del vector de coeficientes, que se puede obtener de la siguiente manera:

```
mifiltro.coeffsEnd = &((uint8_t *) FilterCoefs)[M*2-1]; // FilterCoefs es el vector  
                                                         // de coeficientes
```

- *coeffsPage* es la constante 0xFF00 porque los coeficientes están en RAM
- *delayBase* es la dirección al inicio del vector de retardos, o directamente, el



nombre del arreglo.

- *delayEnd* es la dirección al último byte del vector de retardos, que se puede obtener de la siguiente manera:

```
mifiltro.delayEnd = &((uint8_t *) DelayBuffer)[M*2-1]; // DelayBuffer es el vector de  
// retardos
```

- *delay* es igual a *delayBase*

6- Dentro del código de inicialización, inicializar el vector de retardos con la función *FIRDelayInit*

7- En la rutina de interrupción DMA procesar el búfer que haya sido completado (Llevar en una variable cual es el búfer disponible) mediante la función FIR, cuyos parámetros son

- Cantidad de muestras a procesar (Igual al tamaño del búfer DMA)
- Vector destino (Búfer temporal)
- Vector origen (Búfer DMA del ADC que esté completo, chequear en el registro DMACS1 el bit PPST correspondiente al canal DMA usado en el ADC, ver manual DMA (Pag. 11))
- Puntero a la estructura FIRStruct definida anteriormente.

Luego copiar el vector resultado al búfer DMA del DAC que esté libre (De nuevo chequear el bit PPST correspondiente al canal DMA usado en el DAC). Utilizar la función *VectorCopy*.

8- Probar el funcionamiento del filtro alimentando la entrada del ADC con una salida de audio contaminada con interferencia de 3KHz y conectando la salida del DAC a la entrada de audio de cualquier sistema amplificado (line in de la placa de sonido, entrada de audio de un parlante, etc)

2. Ecuador IIR

- 2.1. Crear desde Matlab 3 filtros IIR chebyshev tipo I de segundo orden, pasa-bajos, pasa-altos y pasa-banda, con frecuencias de corte en 500Hz, 3KHz y [700Hz, 2KHz] y ripple en la banda de paso de 0.1dB. El filtro pasa-banda deberá conformarse con una sección pasa-bajos de 2KHz, seguida de una sección pasa-altos de 700Hz. Graficar la respuesta individual para cada filtro. Graficar en otra figura la respuesta combinada de cuatro etapas en serie (Para el filtro pasa-banda esto implica 4 etapas pasa-bajos y 4 etapas pasa-altos) Por último normalizar los coeficientes dividiendo las constantes por 2, cuantificar con formato S16.15 y Declarar una cadena de caracteres con la secuencia B0,B1,-A1,B2,-A2., repetida 4 veces, puesto que cada filtro se compone de 4 secuencias iguales, como se muestra a continuación:

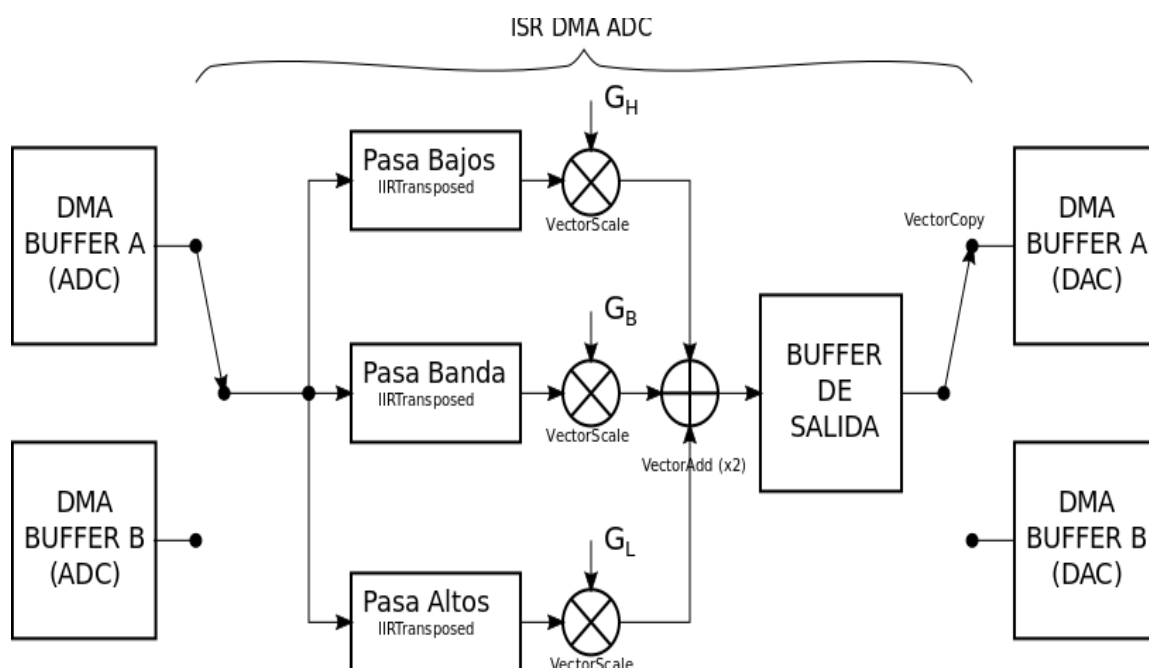
```
{0x8000,0x4000,0x1000,0x0000,0xA000, // B0,B1,A1,B2,A2 (Sección 0)  
0x8000,0x4000,0x1000,0x0000,0xA000, // B0,B1,A1,B2,A2 (Sección 1)
```



```
0x8000,0x4000,0x1000,0x0000,0xA000, // B0,B1,A1,B2,A2 (Sección 2)  
0x8000,0x4000,0x1000,0x0000,0xA000}; // B0,B1,A1,B2,A2 (Sección 3)
```

Para el Pasa-banda habrá un total de 8 secciones, cuatro secciones iguales con coeficientes pasa-bajos y cuatro secciones iguales con coeficientes pasa-altos

- 2.2. Implementar los 3 filtros anteriores siguiendo la estructura de la siguiente figura:



En cada interrupción DMA se calcularán los 3 filtros en paralelo, utilizando como origen de datos el buffer DMA que esté disponible para procesar (Chequear el bit PPST correspondiente en el registro DMACS1). El resultado de cada filtro se almacenará en un búfer intermedio. Luego cada búfer se multiplicará por un factor de escala de tipo fractional, definidos de manera global, y cuyo valor por defecto será 0.5 (0x4000). Estos valores deberán ser actualizado mediante la rutina de interrupción de la UARTRX al recibir un carácter determinado, de la siguiente manera:

	Factor de escala pasa-bajos	Factor de escala pasa-banda	Factor de escala pasa-altos
Aumentar 0.1 si se recibe	'Q'	'W'	'E'
Disminuir 0.1 si se recibe	'A'	'S'	'D'



Asegurarse de no exceder el rango de la variable con las sumas y restas. Comenzando en 0.5, sólo se puede sumar 5 veces y se debe tener cuidado de no excederse de $1-2^{-15}$.

Para realizar la multiplicación se utilizará la función *VectorScale*, que puede tener como destino el mismo vector de origen.

Por último se deberán sumar los 3 búfer intermedios mediante la función *VectorAdd*, que deberá utilizarse 2 veces. El resultado final se almacenará sobre el búfer DMA que esté disponible (De nuevo chequeando el bit PPST) mediante la función *VectorCopy*.

Para realizar el filtrado IIR se deben realizar los siguientes pasos

1- Declarar una variable global de tipo *IIRTransposedStruct*, la cual posee la siguiente definición:

```
typedef struct {  
    int numSectionsLess1;    // Número de secciones menos 1  
    fractional* coeffsBase;  // Dirección de inicio del vector de  
                             // coeficientes  
    int coeffsPage;          // Página del vector de coeficientes (0xFF00  
                             // para RAM)  
    fractional* delayBase1;  // Dirección del vector de retardos 1  
    fractional* delayBase2;  // Dirección del vector de retardos 2  
    int finalShift;          // Ganancia final del filtro, en potencia de 2  
} IIRTransposedStruct;
```

2- Declarar e inicializar el vector de coeficientes, global, de tipo fractional, que tendrá 5 taps por cada sección, ordenados como $B0_{S0}, B1_{S0}, -A1_{S0}, B2_{S0}, -A2_{S0}, B0_{S1}, B1_{S1}, -A1_{S1}, B2_{S1}, -A2_{S1}, \dots, B0_{S3}, B1_{S3}, -A1_{S3}, B2_{S3}, -A2_{S3}$. Este vector deberá estar en la memoria X.

Para los filtros pasa-bajos y pasa-altos se usarán 4 secciones, para el pasa-banda 8 secciones.

3- Declarar e inicializar dos vectores de retardo 1 y 2, globales y de tipo fractional, cada uno con tantos elementos como secciones tenga el filtro. Estos vectores deberán estar en la memoria Y.

4- Declarar una variable global para la ganancia del filtro, escalar y de tipo fractional, inicializada con el valor 0.5 (0x4000).

5- Declarar un vector temporal, global, de tipo fractional, con la misma cantidad de muestras que el búfer DMA.

6- Dentro del código de inicialización del programa, inicializar cada campo de la estructura *IIRTransposedStruct*

7- Dentro del código de inicialización del programa, inicializar los vectores de retardo llamando a la función *IIRTransposedInit*.

8- Dentro de la rutina de interrupción DMA calcular la salida del filtro mediante la función *IIRTransposed*, cuyos parámetros son

- Cantidad de muestras a procesar (Igual al tamaño del búfer DMA)
- Vector destino (Vector temporal del filtro)



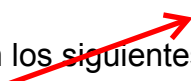
- Vector origen (Búfer DMA del ADC)
- Puntero a la estructura *IIRTransposedSrucl* definida anteriormente.

9- Repetir los puntos 1 a 8 para los filtros restantes

Por último probar el funcionamiento del filtro inyectando una señal de audio a la entrada del ADC conectando la salida del DAC a la entrada de audio de cualquier sistema amplificado (line in de la placa de sonido, entrada de audio de un parlante, etc). Mediante un terminal se deberán enviar caracteres a través del puerto serie y se deberá verificar el funcionamiento del ecualizador.

3. Analizador de Fourier

3.1. Crear un programa que calcule la transformada de Fourier de 32 Muestras del buffer de captura del ADC, con una frecuencia de actualización definida por el alumno entre 1 y 20 veces por segundo.

Los pasos para obtener la transformada de Fourier son los siguientes:  entero y imag

- 1- Declarar el vector origen de los datos, global, con 32 muestras de tipo *fractcomplex*, ubicado en memoria Y y alineado al tamaño de la FFT*4
- 2- Declarar el vector destino de los datos, global, con 32 muestras de tipo *fractcomplex*, ubicado en memoria Y y alineado al tamaño de la FFT*4
- 3- Declarar el vector de los factores de giro, global, con 16 muestras de tipo *fractcomplex*, ubicado en memoria X y alineado al tamaño de la FFT*2
- 4- En el código de inicialización del programa, inicializar el vector de factores de giro mediante una llamada a la función *TwidFactorInit*.
- 5- En la rutina de interrupción DMA del ADC, cada x ciclos (de acuerdo a la frecuencia de actualización), llamar a la función *FFTComplex*. Ver los argumentos y su uso en el manual de la librería. Las muestras del búfer DMA se tomarán como la parte real del vector de entrada a la FFT, mientras que la parte imaginaria se utilizará en cero. Este vector deberá cargarse antes de llamar a la función de cómputo de la FFT.

El resultado de la transformada deberá enviarse por puerto serie, con el siguiente formato:

0	1	2...15	16	17	18	19...32	33	34
0xAA	real[0]	...	real[15]	0x80	imag[0]	...	imag[15]	0x55

Donde para las muestras reales e imaginarias se enviarán los 8 bits más significativos (MSB). Los datos deberán visualizarse desde un terminal o guardados en un archivo.

(Opcional 1) Repetir el programa anterior agregando el enventanado del buffer de datos con alguna de las funciones ventana disponibles en librería.

(Opcional 2) Recibir los datos desde Labview y presentar el espectro de



manera gráfica.