Ecualizador paramétrico multibanda en paralelo tipo FIR Tratamiento Digital de Señales (TDS)

C. Serra Vendrell, J. Ferrando Vila

GIET, Universidad de Valencia, España, {carseven, jorfevi }@alumni.uv.es

Resumen

El objetivo de este proyecto es implementar una aplicación de procesado digital en tiempo real utilizando un DSP de bajo coste de reciente aparición. Se va a utilizar el C5515. Implementaremos un ecualizador de audio paramétrico multibanda basado en la conexión de varios filtros en paralelo para procesar señales de audios digitales muestreadas a 48 kHz. El sistema implementara varios modos de ecualización que podrán ser seleccionables mediante los pulsadores que nos proporciona la placa del DSP. Cada modo será especificado la ganancia (o atenuación) de cada banda. La implantación de las diferentes bandas se realizará mediante filtros tipo FIR.

1. Introducción

El proyecto desarrollado ha sido un problema de gran complejidad para nosotros, pero hemos aprendido muchísimo al poder aplicar en la práctica parte de la teoría aprendida en la asignatura y además al tener que solventar de forma autónoma todos los problemas que fueron surgiendo conforme desarrollábamos el proyecto. Hay que mencionar que el proyecto ha sido dividido en dos fases, una inicial de diseño del filtro y prototipado del algoritmo de convolución para implementar el filtro ante señales de entrada. Esta parte fue desarrollada usando MATLAB como herramienta principal por las comodidades que nos proporciona frente trabajar directamente en el DSP usando C. La otra parte en la que exportaríamos el filtro diseñado e implementaríamos la convolución en el DSP.

2. Desarrollo prototipado

2.1. Primera fase: Diseño en MATLAB

Para el diseño del filtro paramétrico hemos creado 5 ficheros en Matlab, 4 funciones que servirán para la implantación del filtro paramétrico y otro archivo en que realizaremos las diferentes pruebas.

2.2. Funciones LowPass.m, BandPass.m, HighPass.m

Para poder crear el filtro paramétrico multibanda necesitaremos 3 tipos de filtros: un filtro pasa baja que será la primera banda, otro pasa alta que se situará en la última banda y las demás bandas intermedias serán todas filtro pasa banda. Los filtros tendrán que cumplir que las respuestas en frecuencia se cortaran en -6 dB y que la atenuación de la banda pasante debe ser de unos 40 dB cuando las ganancias de todas las bandas sean de 0 dB (Filtro pasa todo).

Para el diseño de los filtros hemos utilizado la herramienta interna de Matlab, Fdatool (Figura 1).

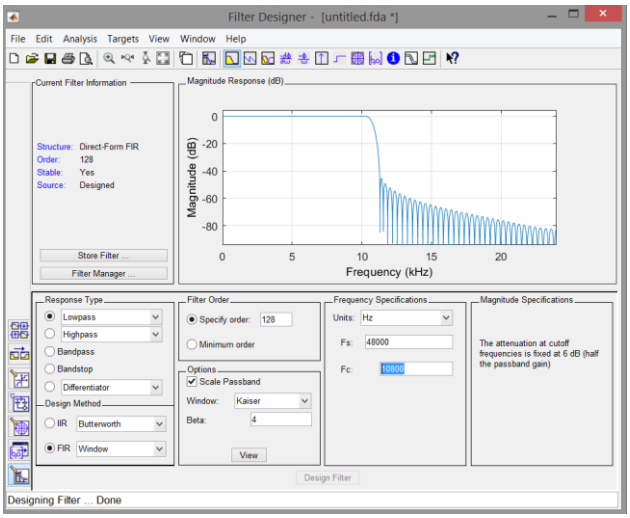


Figura 1. *Diseño filtro pasa baja con fdatool.*

Hemos utilizado el filtro de tipo FIR por enventanado de kaiser. Además, debemos utilizar la opción de especificar el orden para poder diseñar la frecuencia de corte con un valor fijo a 6 dB. Para obtener una atenuación en la banda pasante inferior a los 40 dB modificaremos el parámetro de la beta, con un valor de $\beta = 4$ en el cual logramos conseguir esta especificación. Cabe destacar, que todos los filtros serán diseñados con $\beta = 4$, porque de tener valores distintos tendrían diferentes pendientes y no se cruzarían las frecuencias de corte en 6 dB provocando que nuestro filtro paramétrico no sea pasa todo cuando las ganancias sean de 0 dB.

Con la opción Generate Matlab Code de la herramienta Fdatool, el programa creara el código equivalente para la creación del filtro. Esta función la modificaremos para poder introducir por parámetros de la función las frecuencias de corte, la frecuencia de muestreo y el orden del filtro. La función creada devolverá el vector de coeficientes del filtro diseñado. A continuación, tenemos el código de los 3 tipos de filtros:

LowPass.m

```
function b = LowPass(Fc,Fs,N)
flag = 'scale';  % Sampling Flag
Beta = 4;        % Window Parameter

% Create the window vector for the design algorithm.
win = kaiser(N+1, Beta);

% Calculate the coefficients using the FIR1 function.
b  = fir1(N, Fc/(Fs/2), 'low', win, flag);
```

HighPass.m

```
function b = HighPass(Fc,Fs,N)
flag = 'scale';  % Sampling Flag
Beta = 4;        % Window Parameter
% Create the window vector for the design algorithm.
win = kaiser(N+1, Beta);

% Calculate the coefficients using the FIR1 function.
b  = fir1(N, Fc/(Fs/2), 'high', win, flag);
```

BandPass.m

```
function b = BandPass(Fc1,Fc2,Fs,N)
flag = 'scale';  % Sampling Flag
Beta = 4;        % Window Parameter
% Create the window vector for the design algorithm.
win = kaiser(N+1, Beta);

% Calculate the coefficients using the FIR1 function.
b  = fir1(N, [Fc1 Fc2]/(Fs/2), 'bandpass', win, flag);
```

2.3. Fución ecualizaParalelo.m

La función creará un filtro paramétrico multibanda en paralelo de tipo FIR y filtrará la señal de entrada con el filtro creado (Figura 2).

```
Y = ecualizaParalelo(F,G,N,Fm,x)
```

Salida:

- Y: señal filtrada.

Entrada:

- F: Vector con las frecuencias corte que delimitaran cada banda del filtro.
- G: Vector con las ganancias en dB de cada banda del filtro.
- N: Orden del filtro.
- Fm: Frecuencia de muestreo.
- X: Señal de entrada, está será la que filtraremos con el filtro paramétrico.

Primeramente, pasaremos el vector de ganancias de dB a lineal para poder trabajar con él, para ello usaremos la función interna del Matlab db2mag.

```
G = db2mag(Ganancia);
```

Como hemos explicado antes, la primera banda de nuestro filtro paramétrico será un pasa baja. Usaremos la función explicada anteriormente LowPass.m para crearlo.

```
lp = LowPass(F(1),Fm,N)*G(1);
```

Al ser siempre la primera banda, le introducimos la frecuencia de corte de la primera posición del vector de frecuencias de corte "F(1)". Además, la frecuencia de muestreo "Fm" y el orden del filtro "N". Al llamar a la función LowPass, está nos devolverá los coeficientes del filtro que multiplicamos por la primera posición del vector de ganancias en lineal "G(1)". Finalmente guardaremos el vector de coeficientes con la ganancia ja aplicada en "lp".

La última banda se trata de un filtro pasa alta, para ello usamos la función HighPass para crearlo.

```
hp = HighPass(F(length(F)),Fm,N)*G(length(G));
```

Como se trata de la ultima banda, tendremos que introducir la ultima posición de los vectores de ganancia "G" y frecuencia de corte "F". Para ello, usaremos la función interna de Matlab length, que mide la longitud de un vector y de esta forma logramos que aunque las dimensiones de estos vectores varié, siempre introduciremos la última posición.

Una vez creados los filtros pasa baja y alta, solo tenemos que añadir tantos filtros pasa banda como bandas del filtro paramétrico nos quedan. Hemos utilizado un bucle for que ira de 1 hasta la longitud del vector de ganancias menos 2 ja que serán las bandas que quedarán restantes, ja que siempre tendremos 1 pasa baja y 1 pasa alta, por eso restamos por dos.

```
bp_total = 0;

for i = 1:(length(G)-2)
bp(i,:) = BandPass(F(i),F(i+1),Fm,N)*G(i+1);
bp_total = bp_total +bp(i,:);
end
```

Hay que resaltar que el filtro pasa banda tiene dos frecuencias de corte, una inferior y otra superior que serán dos consecutivas del vector "F". Es por esto que F(i) y F(i+1) como frecuencias de corte, para poder lograr esto. Para el vector de ganancia tenemos G(i+1) porque como el bucle empieza en 1, no podemos usar la ganancia G(1) ja que se trata de la ganancia del filtro pasa baja. Finalmente, usaremos bp_total como un vector para almacenar la suma de los coeficientes de los filtros que vamos creando, como consecuencia bp_total al final del bucle for almacenara los coeficientes del filtro pasa banda total.

Con todos los filtros creados, ja podremos sumar los coeficientes del pasa baja, pasa alta y el pasa banda total. Con esto, obtendremos el filtro paramétrico multibanda en paralelo equivalente.

```
parametric = lp + hp + bp_total;
```

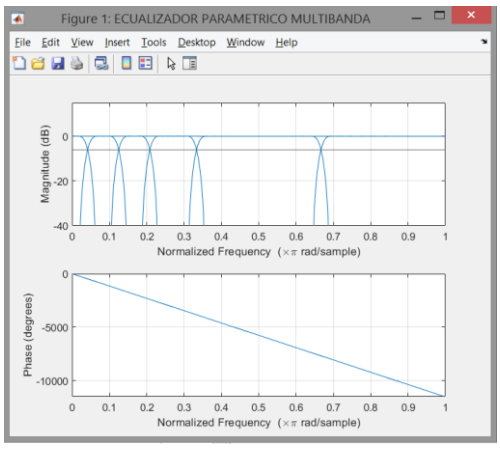


Figura 2. *Filtro creado por ecualizaParalelo,m .*

Como observamos, tenemos las diferentes bandas representadas. Al inicio tenemos un pasa baja, al final un pasa alta y todos los demás son pasa banda. La ganancia de las bandas es de 0 dB y se cumple que las frecuencias de corte se cruzan en -6 dB. Sin duda, la atenuación en la banda pasante es inferior a 40 dB. Sobre todo, destacar que el filtro equivalente tiene forma de pasa todo, vale constantemente 0 dB.

Una vez obtenemos la respuesta impulsional del filtro "parametric", lo aplicaremos sobre la señal de entrada "x". La implementación de un filtro FIR es la siguiente (Figura 3).

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k}$$

Figura 3. *Algoritmo implementación filtro FIR .*

En el fondo es como si realizáramos la convolución de la respuesta impulsional del filtro con la señal de entrada. Nosotros hemos implantado el algoritmo en Matlab de la siguiente forma:

```
for n = length(parametric)+1:length(x)
 y(n) = 0;
 for k = 1:length(parametric)
 y(n) = y(n) + parametric(k) * x(n-k);
 end
end
```

Hemos utilizado dos bubles for, el primero va de length(parametric+1) hasta length(x), porque de esta forma nos evitamos las x(n-k) negativas y como el vector de entrada tiene muchas muestras no supone un error muy grande. Nos ahorramos con esto crear un buffer con los retardos. El segundo bucle va de 1 hasta length(x), ja que en Matlab los vectores empiezan en 1, es la forma equivalente de ir de 0 a N-1. Finalmente, vamos almacenado las operaciones en la variable "y", esta será nuestra señal filtra.

Para comprobar que nuestro algoritmo funciona correctamente, hemos usado la función interna de Matlab filter y obtenemos los mismos resultados al filtrar la misma señal.

```
y = filter(parametric,1,x);
```

2.4. Principal.m

Este archivo nos servirá para realizar las pruebas de la función ecualizaParalelo.m.

Primeramente, cargaremos el archivo de audio que utilizaremos como señal de entrada. Usaremos la función interna de Matlab audioread. Esta función nos devuelve la frecuencia de muestreo de la señal de audio que utilizaremos más adelante en ecualizaParalelo.

```
[x,Fm] = audioread('Fisher - Losing It.wav');
```

A continuación, estableceremos las variables que necesitara ecualizaParalelo y llamaremos a esta. Añadir que solo proporcionaremos un canal de audio x(:,1) para poder trabajar con la función.

```
%Orden filtros
N = 128;
%Limites de las bandas
F = [1000,3000,5000,8000,16000];
%Modos de ecualización
Llano = [0 0 0 0 0 0];
y = ecualizaParalelo(F, Eq0, N, Fm,x(:,1));
```

Tras llamar a la función esta no devolverá la señal de entrada filtrada y ja solo tendremos que escucharla con la función interna de Matlab sound para obtener el resultado.

```
sound(x(:,1), Fm) % Original
sound(y,Fm) % Filtrada
```

Seguidamente vamos a comparar algunas señales de entrada frente a las señales filtradas con varios modos de ecualización.

Ejemplo 1:

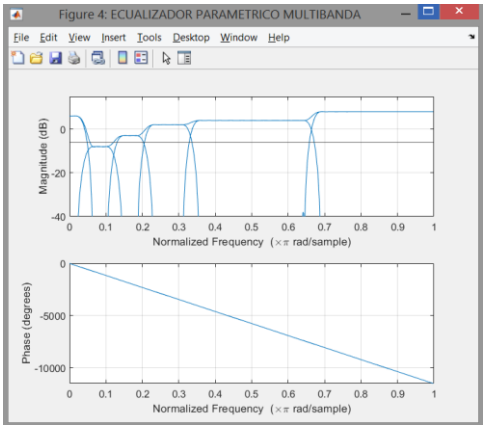


Figura 4. *Respuesta frecuencia filtro paramétrico con ROCK.*

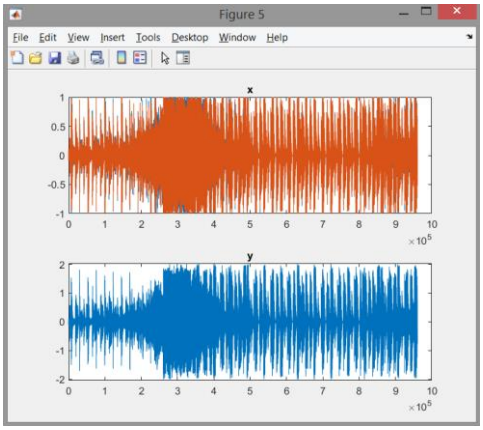


Figura 5. *X vs Y. Con filtro paramétrico con ROCK.*

Ejemplo 2:

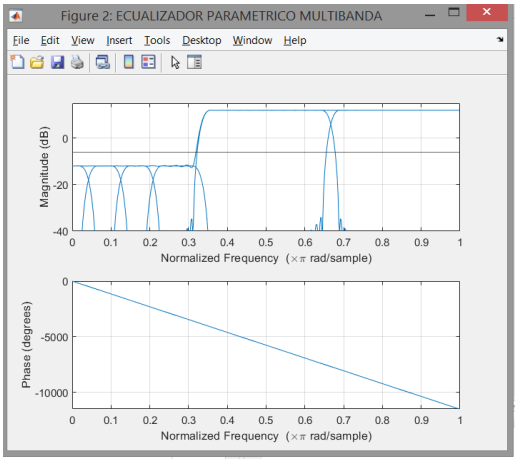


Figura 6. *Respuesta frecuencia filtro paramétrico con Eq0.*

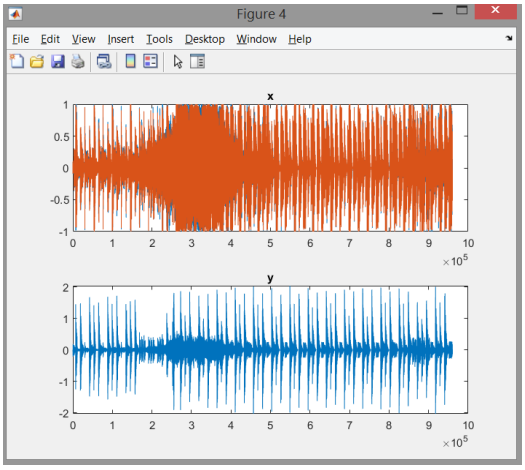


Figura 7. *X vs Y. Con filtro paramétrico con Eq0.*

3. Desarrollo en el DSP

3.1. Segunda fase: Code Composer Studio.

Antes de trabajar con el DSP con el Code Composer Studio, lo que tendremos que hacer es crear diferentes filtros paramétricos con diferentes ecualizaciones y exportar estos filtros a C con la función FIR_dump2cHPDS_INT.m que explicaremos a continuación. Y una vez tenemos los coeficientes en C, implementaremos el algoritmo de filtrado en el DSP y crearemos un menú en el que el usuario podrá elegir que ecualización será aplicada a la señal de entrada.

3.2. Segunda fase: Exporta coeficientes a C.

Para exportar los coeficientes de Matlab al DSP debemos transformar los coeficientes del filtro de no enteros a enteros, debido a que esta es la forma de trabajo óptima para el DSP. Usaremos la función FIR_dump2cHPDS_INT.m para hacer esta transformación.

```
coeffs = FIR_dump2cIHPDS_INT('Auriculares',
'Auriculares', parametric, length(parametric));
```

A la función le proporcionaremos el nombre de los ficheros .c y .h , los coeficientes que queremos exportar y la longitud del vector de coeficientes. Esta creara un fichero .c y .h que añadiremos al directorio de trabajo de nuestro proyecto en Code Composer Studio.

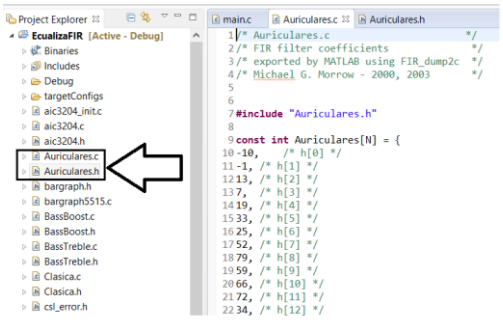


Figura 8. *Coeficientes exportados al DSP.*

Pero para que nuestro archivo main.c pueda encontrar los archivos, lo tenemos debemos incluir en la cabecera del fichero.

```
#include "Llano.h"
#include "Rock.h"
#include "Dance.h"
#include "Techno.h"
#include "Pop.h"
#include "Clasica.h"
#include "Vivo.h"
#include "BassBoost.h"
#include "Treble.h"
#include "BassTreble.h"
#include "Auriculares.h"
```

Figura 9. *Incluimos archivos .h en la cabecera del main.*

Mencionar que estas son todas las ecualizaciones que hemos incluido en el DSP.

3.3. Primera implementación.

En nuestro primer intento para realizar la implementación del filtro FIR realizamos el siguiente código:

Este fragmento de código solo mostramos la idea principal y no esta desarrollado por completo puesto que no es la implantación final y queríamos mostrar simplemente la idea. No queríamos extendernos mucho en esta parte.

Este for se trata del buffer delay en el que almacenaremos los retardos que son necesarios para la implantación del filtro FIR.

```
for(j=N-1;j>0;j--) buffer[j] = buffer[j-1];
```

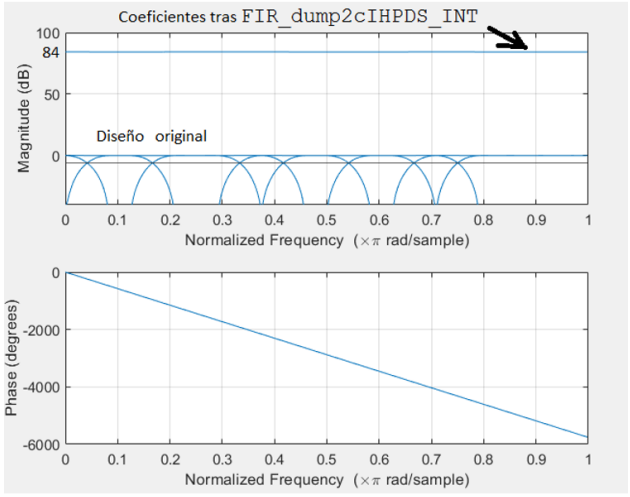
Después de actualizar el buffer delay almacenamos la muestra “actual” en la primera posición del vector.

```
buffer[0] = mono_input;
```

Por último, realizamos la convolución en este for.

```
for(j=0; j<N; j++) output += buffer[j]*h[j];
```

El resultado de nuestro código era una señal saturada y tras muchas comprobaciones no dimos cuenta que el algoritmo es correcto y el problema se encontraba cuando exportábamos los coeficientes.



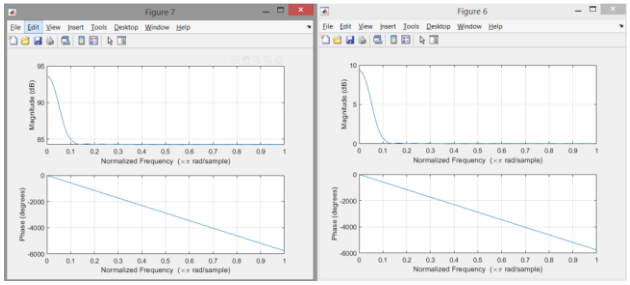


Figura 10 y 11. *Comparación coeficientes antes y después FIR_dump2cHPDS_INT.*

Como observamos, al convertir los coeficientes a enteros, obtenemos una ganancia mucho mayor a la que teníamos, esto es lo que provocaba que la señal saliera saturada. Para solucionarlo, solo tendremos que dividir la salida por un valor igual al de la ganancia demás que tenemos al convertir a enteros.

```
    }
//BUFFER DELAY

    for(j=N-1; j>0; j--) buffer_delay[j] = buffer_delay[j-1];   //Movemos a la derecha los retardos
    buffer_delay[0] = mono_input;
    //AÑadinos muestra

//CONVOLUCION

    aux =0;                                                    //Vaciamos variable auxiliar.
    for(j=0; j<N; j++) aux += buffer_delay[j] * (Int32)Llano[j] / 64;//h(n)*x(n-k)
    aux /= 30;

    // Con es de 84.44 db --> sqrt(db2mag(84.44)) = 129

//SALIDA AUDIO

    output = (Int16)aux;                                       // Convertimos salida de la conv a INT16

    right_output = output;                                     // Sacamos por la salida el resultado de la conv.
    left_output =  right_output;                               // Salida izquierda lo mismo que derecha. MONO
}
```

Figura 12. *Captura de la implementación algoritmo en C.*

Tras hacer esto y jugar con lo valores que dividimos, logramos escuchar la señal de salida de forma muy débil y distorsionada. Como no estábamos satisfechos con el resultado obtenido, decidimos buscar una solución alternativa que explicaremos en el siguiente apartado.

3.4. Implantación final.

Nuestra implementación final fue utilizando las funciones de la librería DSPlib, en concreto una llamada "fir" que implementa el algoritmo de un filtro FIR (Figura 13).

$$r[j] = \sum_{k=0}^{nh-1} h[k]\, x[j-k] \qquad 0 \le j \le nx$$

Figura 13. *Algoritmo interno de la función fir del DSPlib.*

La función "fir" es la siguiente y necesitara que le pasemos todos estos argumentos:

ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Salida:

- Oflag: Overflow erro flag.

Entrada:

- x: puntero hacia el vector de entrada.
- h: puntero hacia el vector de coeficientes.
- r: puntero hacia el vector de salida.
- dbuffer: puntero hacia el vector buffer delay.
- nx: número de muestras de entrada.
- nh: numero de coeficientes del filtro.

Así es como hemos implementado la función.

```
DATA y;
DATA dbuffer[N+2];
DATA x;
Int16 Parametric[N];

fir(&x, Parametric, &y, dbuffer, 1, N);
```

Pasamos la variable de entrada y salida como variables, puesto queremos hacerlo en tiempo real. Por eso nx = 1 también para que la función internamente lo sepa. Utilizamos &x ja que pasamos el puntero.

El vector dbuffer es de una longitud N+2 pues la función especifica que sea de esa longitud para poder trabajar internamente.

Pasamos directamente dbuffer pues en C los vectores son punteros.

Figure 4–16. dbuffer Array in Memory at Time j

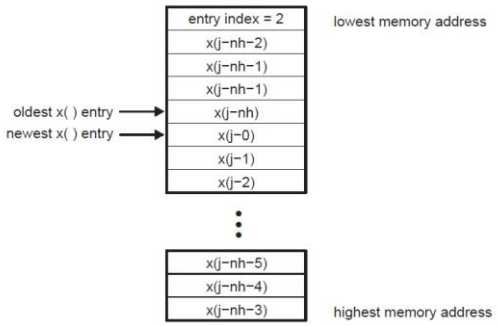


Figura 13. *Estructura del vector dbuffer en la función fir.*

También, le tenemos que proporcionar los coeficientes del filtro y pasaremos estos con el vector Parametric y la longitud de este que será mediante N.

Previamente hemos guardado los coeficientes en Parametric porque como Pop (y los coeficientes resto) es de tipo const int, la función "fir" no nos admitía este tipo de dato. Por eso, antes de usar "fir" guardamos los coeficientes en Parametric para guardarlos como INT16 que sí que lo admite.

```
for(j = 0; j < N; j++) Parametric[j] = Pop[j];
```

Y con este será el código que hemos implementado, solo quedará enviar la variable y a la salida de audio del DSP.

```
right_output = y;
left_output = right_output;
```

Para finalizar, solo nos queda mostrar la estructura de menús utilizada para que el usuario final pueda elegir los diferentes tipos de modos de ecualización que hemos creado. Para ello hemos usado la estructura de step de los ejemplos de los cursos de Texas Instrument.

```
Step = pushbuttons_read(12);
```

Hemos implementado 12 menus puesto que hemos implementado 12 modos:

- Modo Entrada = Salida: Cuando se seleccione este modo, no se realizará ninguna modificación a la señal de entrada del DSP.

```
right_output = right_input;
left_output = left_input;
```

Figura 14. *Modo Entrada = Salida.*

- Modo Llano: Aplicamos el algoritmo expuesto antes para aplicar el filtro Llano a la entrada y sacamos el audio por la salida del DSP.

```
// MODO 2: LLANO

    else if ( Step == 2 )
    {
        if ( Step != LastStep )
        {
            oled_display_message("TDS  SERRA/FERRANDO", "2. LLANO          ");
            LastStep = Step;
            for(j = 0; j < N; j++) Parametric[j] = Llano[j];          // Convertimos coefs a INT16 guardandolo en una variable.
        }
        x = mono_input;                                              // Muestra que entrara en el filtro.
        fir(&x, Parametric, &y, dbuffer, 1, N);                      // Aplicamos filtro FIR.
        right_output = y;                                            // Sacamos por la salida el resultado de la conv.
        left_output =  right_output;                                 // Salida izquierda lo mismo que derecha. MONO

    }
```

Figura 15. *Código implementado para el modo Llano.*

Figura 16. *Modo Llano.*

Los demás modos se implementarán de la misma forma, solo que ahora proporcionaremos el vector de coeficientes para realizar la convolución con el de cada modo.

3.5. Conclusiones.

Para concluir, hay que destacar que los resultados obtenidos son muy gratificantes. Hemos logrado los objetivos y los filtros funcionan correctamente. Este proyecto a supuesto todo un reto para nosotros al que hemos dedicado muchas horas, pero ha valido la pena, hemos aprendido mucho de forma práctica.

Referencias

[1] TMS320C55x DSP Library Programmer's Reference – Texas Instrumets.
[2] Material de la asignatura de Tratamiento Digital de Señales – GIET – ESTE – Universitat de València. Jordi Muñoz.
[3] Práctica 0 del Laboratorio de Señales y Sistemas- GIET – ESTE – Universitat de València.

[4] The scientist and Engineer's Guide to Digital Signal Processing – Steven W.Smith, Ph.D.