LABORATORI DE COMUNICACIONS 2

se le dijo, ... y lo olvidó
lo vio, ... y lo creyó
lo hizo, ... y lo comprendió

Confucio, siglo VI a.C.

Octubre 2009

1. El Entorno de Trabajo

1.1 Objetivos

El objetivo de la primera práctica es el de introducir el entorno hardware y software de desarrollo de aplicaciones que se utilizará durante el curso.

1.2 El entorno de trabajo

En el campus digital puede encontrarse información sobre las herramientas de trabajo del LC 2. En concreto sobre los ítems siguientes: Tarjeta 6713DSK (`6713_dsk_techref.pdf`), el DSP TMS320C6713 (`tms320c6713.pdf`), Software CCS (`spru509f.pdf`), y Conversor A/D-D/A (`tlv320aic23b.pdf`). Los contenidos de más interés para el seguimiento de la asignatura son los siguientes:

1) Tarjeta de desarrollo 6713DSK
   - Descripción de la tarjeta, diagrama de bloques y mapa de memoria
   - El microprocesador TMS320C6713, arquitectura interna y buses
2) Software Code Composer Studio 3.1
   - Inicialización y creación de proyectos
   - Configuración y comandos de uso del compilador y linker
   - Comandos de uso del depurador
3) El conversor AIC23B
   - Diagrama de bloques, estructura interna y características
   - Conexión con el TMS320C6713
   - Inicialización y configuración
   - Proceso de lectura y escritura.

1.3 Trabajos propuestos

1.3.1 Introducción

El programa `PRAC1_INT.C` contiene en la subrutina principal `main()` un bucle infinito que recibe una muestra del conversor A/D y envía una muestra al conversor D/A en cada iteración. Esta transferencia de datos desde los conversores la realiza la subrutina `interrupt void c_int11()` mediante un acceso a los registros adecuados de la 6713DSK en cuanto los datos están disponibles. Interprete el funcionamiento del programa.

1.3.2 En el laboratorio

A. Acceda al sistema mediante el nombre de usuario `lc2xxy`, donde `xx` es el número de grupo del cuatrimestre 4A, e `y` es el número del puesto de trabajo. El password es `lc2`, y debe cambiarse la primera vez que se accede al sistema (con 5 caracteres como mínimo). Este será el login que identifique el grupo de estudiantes: el usuario es validado en el servidor, por lo que podrá utilizarse en cualquier PC del laboratorio. Cada usuario tiene acceso a los discos siguientes:

Disco C: Acceso local únicamente a las carpetas que grupo de estudiantes cree en el escritorio. El disco C: también contiene el entorno de trabajo software.

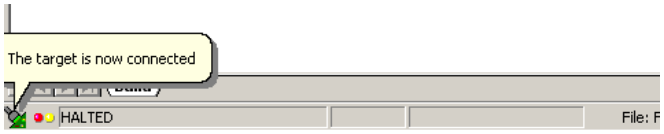
Disco H: Disco situado en el servidor en el que grupo de estudiantes puede almacenar su trabajo de forma permanente.

Disco P: Disco situado en el servidor del que pueden copiarse los archivos necesarios para la realización de las practicas.

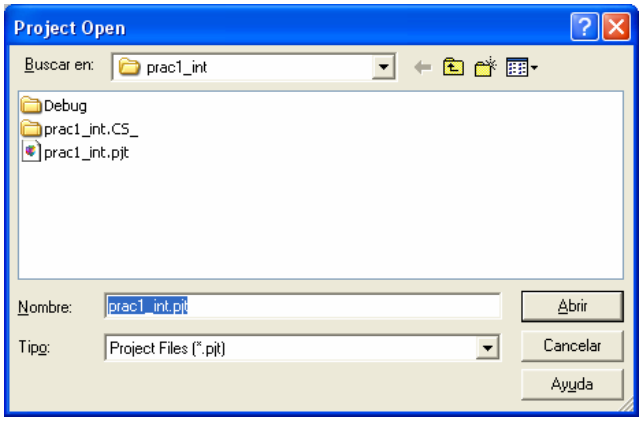
Disco S: Disco situado en el servidor que contiene manuales de laboratorio y documentación adicional.

IMPORTANTE: Los archivos de trabajo pueden estar en el escritorio del PC local, o en el disco H. Recuerde que, al final de cada sesión de laboratorio, el trabajo realizado ha de quedar grabado en el disco H (o bien en un dispositivo externo, como un pendrive). No se garantiza que los directorios creados en el escritorio se conserven de una sesión a otra.

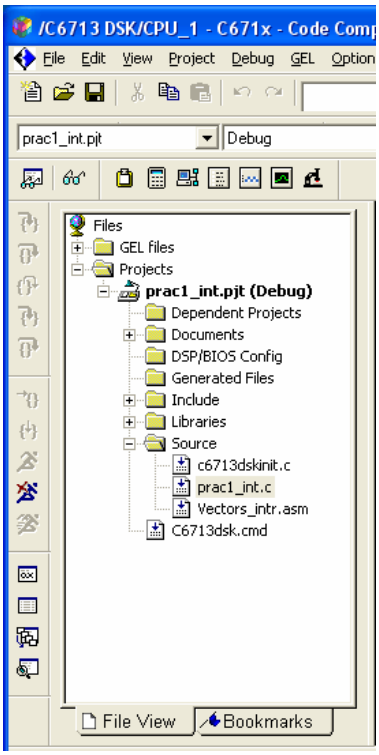
B. Asegúrese de que la tarjeta 6713DSK está conectada al PC mediante un cable con conector USB. Ponga en marcha la tarjeta y arranque el Code Composer Studio (CCS) usando el icono que aparece en el escritorio. Los 4 leds que están agrupados en la parte posterior de la placa deben parpadear y quedarse encendidos. Para tener el entorno activo y operativo, lo primero que tenemos que hacer es pulsar la combinación de teclas ALT+c, para conectar el DSP al PC. Si la tarjeta se ha inicializado correctamente los 4 leds deberán apagarse. La siguiente figura muestra como queda el entorno una vez conectado y preparado.



Cree en el escritorio del PC (o en el disco H) un directorio con el nombre LC2 y copie en este directorio el contenido del directorio `P:\LC2\PRACTICA1\` que contiene los archivos necesarios para realizar esta práctica. Este es el procedimiento habitual en todas las prácticas. Una vez copiados los archivos, abra desde el menú Project/Open del CCS el archivo con extensión .pjt existente en su directorio:

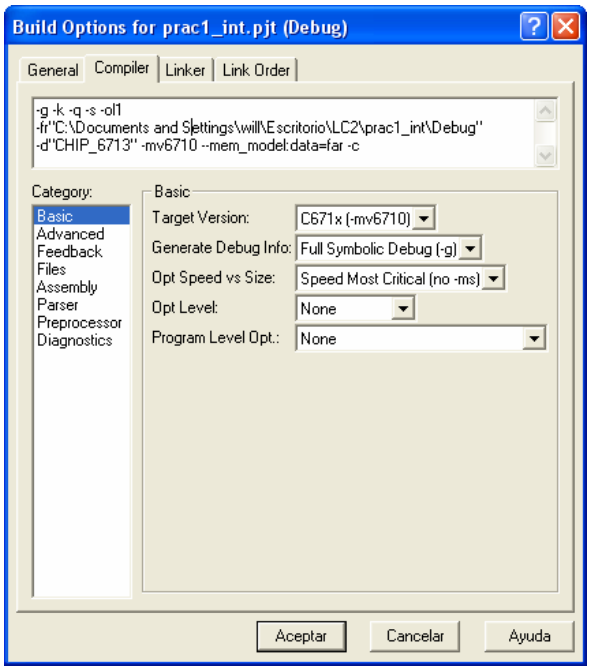


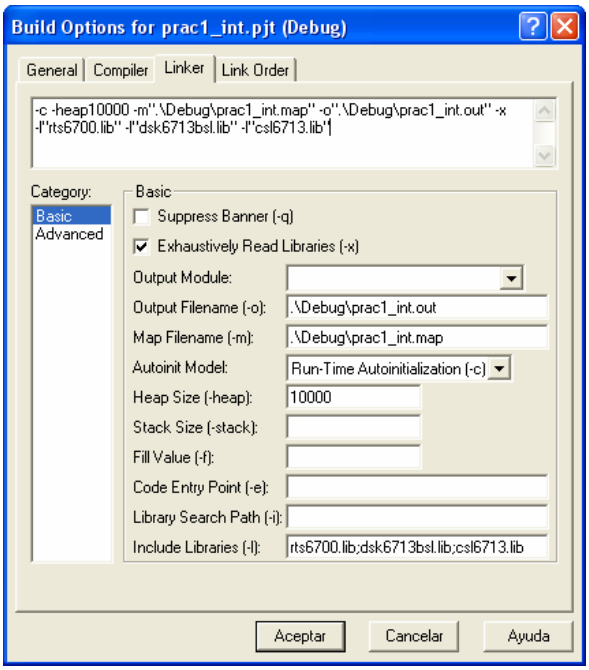
Los archivos con extensión .pjt contienen la información que usa el CCS, para realizar un proyecto. Incluyen la referencia a todos los archivos que van a ser usados por el compilador y el linker, y su contenido aparece en la ventana izquierda del CCS (véase la figura). Entre otros están: los archivos fuente (con extensión .c), el archivo de comandos para el linker (con extensión .cmd), los archivos incluidos (con extensión .h) y las librerias (con extensión .lib).



Abra el archivo `prac1_int.c` dentro de la carpeta `Source` que aparece en esta ventana. El programa `main()` (véase la sección 1.4 Apéndice A: Listado de los Programas utilizados) contiene un bucle infinito en el que se lee una muestra del conversor A/D y la misma se escribe en el conversor D/A. La lectura y escritura se realiza mediante la rutina de servicio de interrupción `c_int11()`.

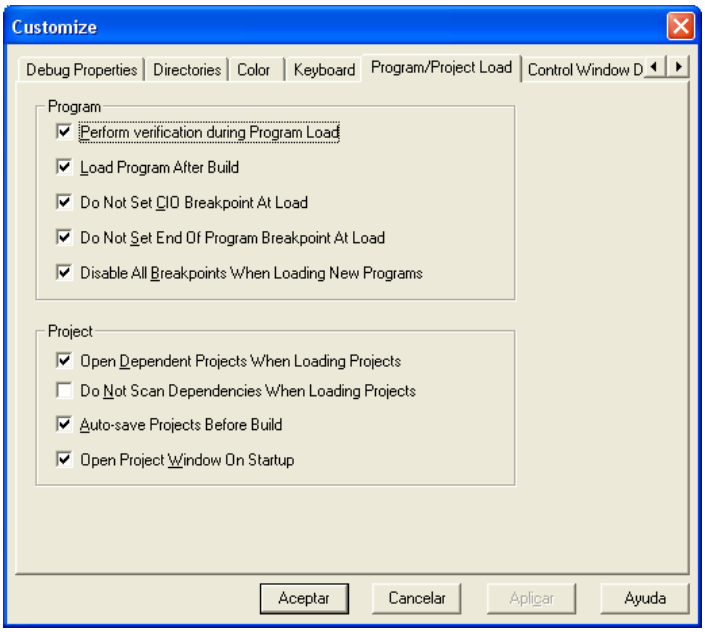
Para generar el archivo ejecutable utilice el comando Project/Build (F7), que compila y enlaza los archivos y librerías necesarios. Compruebe antes que los parámetros de compilación y enlace se ajustan a los especificados en la opción Project/Build options…, y que aparece en las siguientes figuras:





El compilador crea el código objeto `PRAC1_INT.OBJ`. El linker crea el código ejecutable en el archivo `PRAC1_INT.OUT` enlazando el código `PRAC1_INT.OBJ` con las rutinas de inicialización y las librerías

standard de C incluida en el proyecto. Por último el propio entorno del CCS tiene un asistente que carga el programa en la memoria de la placa de evaluación, si las opciones correctas en el menú Option/Customize…ProgramProject Load están seleccionadas:



Para ejecutar el programa seleccione Debug/Run (F5). Para parar la ejecución puede utilizar Debug/Halt (Mayúsculas+F5).

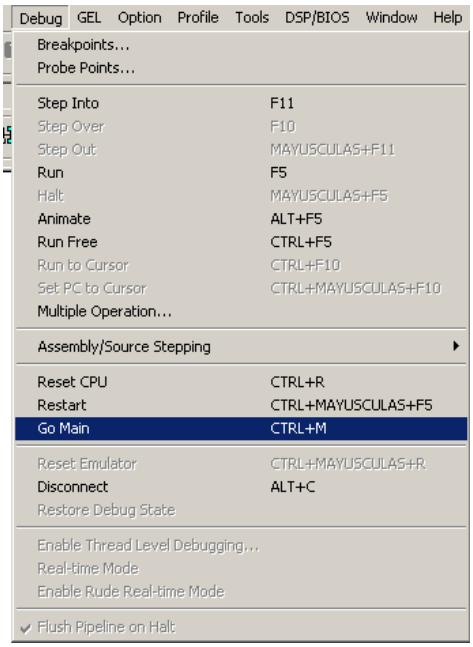
Utilice el generador de señales y el osciloscopio para comprobar que la salida coincide con la entrada para frecuencias inferiores a la frecuencia de corte de los filtros. Determine esta frecuencia de corte a 3 dB experimentalmente utilizando para ello uno de los frecuencímetros disponibles en el laboratorio (no utilice la base de tiempos del osciloscopio). ¿Puede identificar el aliasing?

C. Utilice el editor del entorno del CCS para modificar el programa `PRAC1_INT.C` y obtener una realización simple y eficiente del sistema

$$y[n] = x[n] (-1)^n$$

donde x[n] es la secuencia de entrada recogida del conversor A/D e y[n] es la secuencia de salida enviada al conversor D/A. Recuerde que la solución propuesta deber ser muy simple y no podrá contener ni llamadas a funciones exponenciales ni sinusoidales. Denomine al archivo modificado `PRAC1_INTM.C`, sustitúyalo por `PRAC1_INT.C` en el archivo de proyecto: pueden incluirse archivos a un proyecto haciendo clic con el botón derecho del ratón sobre las carpetas que aparecen en la ventana de proyectos. Cambie el nombre del programa ejecutable a `PRAC1_INTM.OUT` en Project/Build options…/Linker, y compile de nuevo.

D. Utilice el depurador de programas del entorno CCS para cargar el programa en la placa de evaluación y ejecutarlo. El sistema de depuración de aplicaciones contiene las opciones que aparecen a continuación:



Sitúese al inicio del programa principal usando *go main* (CRTL+m) y ejecute paso a paso el bucle infinito para comprobar que funciona tal como se desea.

Para ejecutar paso a paso puede utilizar las teclas de función:

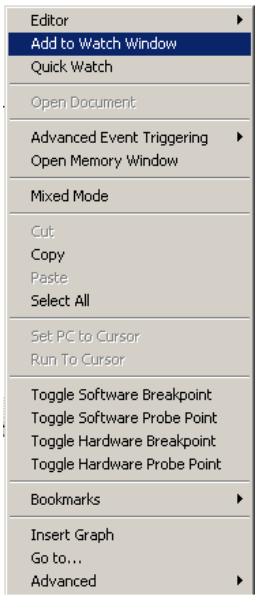
Step Into Ejecuta paso a paso entrando en las subrutinas (F11)

Step Over Ejecuta paso a paso saltando las subrutinas (F10)

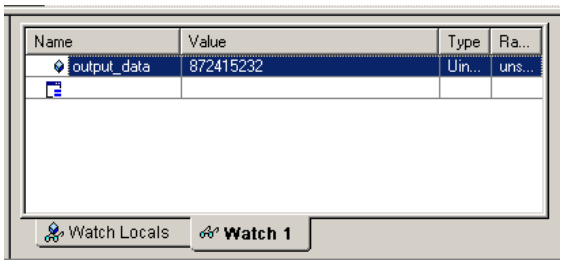
Step Out Sale de dentro de una subrutina la cual hemos entrado con un Step Into (Mayúsculas + F11)

También puede establecer puntos de ruptura (*breakpoints*) señalando con el ratón una línea del programa fuente y pasar a ejecutar el programa hasta ese punto mediante la tecla F5.

Para examinar el valor de variables de C realice un click con el botón derecho del ratón, encima de la ventana donde aparece el código, y se le desplegará el siguiente menú.



Para ver diferentes valores de la variables, haga click en *Add to Watch Window* tantas veces como necesite. Inserte el nombre de las siguientes variables `output_data, input_data`.



D. Anote en la libreta de prácticas la solución adoptada para obtener el sistema digital propuesto.

E. Compruebe que cuando la entrada es una sinusoide la salida es también otra sinusoide. Determine experimentalmente y de forma analítica la relación entre la frecuencia de entrada F_e y la frecuencia de salida F_S. Represente el resultado de forma gráfica (F_S en función de F_e). Utilice este resultado para determinar la frecuencia de muestreo de los conversores.

F. Modifique la frecuencia de trabajo de los conversores a 8 KHz, modificando el valor de la variable fs (compruebe cuales son los valores de frecuencia de muestreo posibles en el archivo `dsk6713_aic23.h`). Verifique la relación encontrada en E.

1.4 Apéndice A: Listado de los Programas utilizados

La subrutina de inicialización del interfaz analógico `comm_intr()` configura el `timer` del DSP, así como los registros del DSK que determinan la frecuencia de muestro y las demás opciones disponibles.

```
//=========================================================================//
//         PRAC1_INT.C: Archivo principal de la práctica 1
//=========================================================================//
/*Start of prac1_int.c  file*/

#include "dsk6713_aic23.h"              //archivo de soporte codec-DSK
#define IDLE  asm("  idle")
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;    //ajuste de la frecuencia de muestreo
int output_data;
int input_data;

interrupt void c_int11()                      //rutina de servicio de interrupcion
{
   input_data = input_left_sample();        // señal del canal izquierdo de line in
   output_sample(output_data);
   return;
}

void main()
{
  comm_intr();                                //inicio   DSK,   codec,   McBSP   usando
interrupciones
  while(1)                                    //bucle infinito
  {
   IDLE;
   output_data = input_data;
  }
}
```

Figura 1.1 Listado de `PRAC1_INT.C`. Contiene la función `main()` y la rutina de servicio de interrupción `c_int11()` para lectura y escritura de muestras

```
//=========================================================================//
//   C6713dskinit.c Includes functions from TI in the C6713 CSL and C6713DSK BSL
//=========================================================================//

#include "C6713dskinit.h"
extern Uint32 fs;                  //for sampling frequency

void c6713_dsk_init()              //dsp-peripheral initialization
{
   DSK6713_init();                    //call BSL to init DSK-EMIF,PLL)
   hAIC23_handle=DSK6713_AIC23_openCodec(0, &config);//handle(pointer) to codec
   DSK6713_AIC23_setFreq(hAIC23_handle, fs);  //set sample rate
   MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData);//interface 32 bits toAIC23
   MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START | MCBSP_RCV_START |
   MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);//start data channel again
}

void comm_poll()             //added for communication/init using polling
{
   poll=1;                           //1 if using polling
      c6713_dsk_init();              //init DSP and codec
}

void comm_intr()                //for communication/init using interrupt
{
   poll=0;                                 //0 since not polling
      IRQ_globalDisable();              //disable interrupts
   c6713_dsk_init();                //init DSP and codec
   CODECEventId=MCBSP_getXmtEventId(DSK6713_AIC23_codecdatahandle);//McBSP1 Xmit
   IRQ_setVecs(vectors);            //point to the IRQ vector table  //since    interrupt
vector handles this
   IRQ_map(CODECEventId, 11);    //map McBSP1 Xmit to INT11
   IRQ_reset(CODECEventId);         //reset codec INT 11
      IRQ_globalEnable();              //globally enable interrupts
   IRQ_nmiEnable();                 //enable NMI interrupt
      IRQ_enable(CODECEventId);     //enable CODEC eventXmit INT11
   output_sample(0);                   //start McBSP interrupt outputting a sample
}

void output_sample(int out_data)         //for out to Left and Right channels
{
   short CHANNEL_data;

   AIC_data.uint=0;                          //clear data structure
   AIC_data.uint=out_data;                //32-bit data -->data structure

//The existing interface defaults to right channel. To default instead to the
//left channel and use output_sample(short), left and right channels are swapped
//In main source program use LEFT 0 and RIGHT 1 (opposite of what is used here)

   CHANNEL_data=AIC_data.channel[RIGHT];       //swap left and right channels
   AIC_data.channel[RIGHT]=AIC_data.channel[LEFT];
   AIC_data.channel[LEFT]=CHANNEL_data;
      if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));//if ready to transmit
      MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);//write/output data
}

void output_left_sample(short out_data)         //for output from left channel
{
   AIC_data.uint=0;                             //clear data structure
   AIC_data.channel[LEFT]=out_data;   //data from Left channel -->data structure

   if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));//if ready to transmit
      MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);//output left channel
}

void output_right_sample(short out_data)    //for output from right channel
{
   AIC_data.uint=0;                                    //clear data structure
   AIC_data.channel[RIGHT]=out_data;   //data from Right channel -->data structure

   if (poll) while(!MCBSP_xrdy(DSK6713_AIC23_DATAHANDLE));//if ready to transmit
      MCBSP_write(DSK6713_AIC23_DATAHANDLE,AIC_data.uint);//output right channel
}

Uint32 input_sample()                          //for 32-bit input
{
   short CHANNEL_data;

   if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//if ready to receive
   AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);//read data

//Swapping left and right channels (see comments in output_sample())
```

```
   CHANNEL_data=AIC_data.channel[RIGHT];       //swap left and right channel
   AIC_data.channel[RIGHT]=AIC_data.channel[LEFT];
   AIC_data.channel[LEFT]=CHANNEL_data;

   return(AIC_data.uint);
}

short input_left_sample()                      //input to left channel
{
   if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//if ready to receive
   AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);//read into left channel
   return(AIC_data.channel[LEFT]);     //return left channel data
}

short input_right_sample()                     //input to right channel
{
   if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//if ready to receive
   AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);//read into right channel
   return(AIC_data.channel[RIGHT]);            //return right channel data
}
```

Figura 1.2. Funciones de gestión de interrupciones y de lectura/escritura de muestras

2. Estructura de un programa en C

2.1 Lenguaje C en el entorno de programación

Se explican algunas particularidades avanzadas en el uso del C.

2.2 Estructura de un programa en C en procesado digital de señal

El esquema del programa principal utilizado en el procesado de una señal en tiempo real depende de sí el procesado se realiza muestra a muestra o bloque a bloque cada N muestras. A continuación se comentan las dos opciones.

2.2.1 Procesado muestra a muestra

La lectura de la muestra a procesar y la escritura de la muestra procesada se pueden realizar dentro del bucle principal mediante interrupción:

```
int input, output;

void c_int11(void)
{
   input_data = input_left_sample();   // señal del canal izquierdo de line in
   output_sample(output_data);
   return;
}

void main(void)
{
   comm_intr();                        //inicio DSK, codec, McBSP usando interrupciones
   while(1)
   {
      IDLE;                            // espera a la interupción
      output = proceso(input);         // procesa una muestra
   }
}
```

El retardo introducido por este proceso es de una muestra. Por otro lado, para que el programa se pueda ejecutar en tiempo real, el tiempo de procesado de una muestra (función `proceso()`), debe ser inferior al periodo de muestreo.

2.2.2 Procesado por bloques

Cuando el procesado de una señal se realiza bloque a bloque cada N, muestras la lectura y escritura se deben realizar en paralelo con la subrutina principal de procesado de un bloque. En el siguiente ejemplo la subrutina `block_process` es interrumpida a la frecuencia de muestreo por la subrutina `c_int11` que deposita una nueva muestra en el buffer de entrada `ibuffer` y envía la siguiente muestra del buffer de salida `obuffer`.

```
#define LBUFF 512                          // tamaño del bloque
int ibuff1[LBUFF], ibuff2[LBUFF];
int *ibuffer = ibuff1, *iblock = ibuff2;
int obuff1[LBUFF], obuff2[LBUFF];
int *obuffer = obuff1, *oblock = obuff2;

interrupt void c_int11(void) {
   *opnt = obuffer[index];                    // envía muestra procesada
   ibuffer[index] = *ipnt;                    // lee siguiente muestra
   sample++;                                 // se incrementa el puntero
}

void wait_buffer(void) {
   int *pt;
   while (sample < LBUFF) ;

   // Intercambia los punteros del bloque y el buffer de entrada //
   pt = ibuffer;
   ibuffer = iblock;
   iblock = pt;

   // Intercambia los punteros del bloque y del buffer de salida //
   p = obuffer;
   obuffer = oblock;
   oblock = p;
   sample = 0;                                // se inicializa el puntero
}

void main(void) {
   init();
   while (1) {
      wait_buffer();                          // espera a que se lean N muestras
      block_process(iblock, oblock);
   }
}
```

Para reducir los requerimientos de memoria se puede dejar el resultado de procesar un bloque en el mismo *array* de entrada. Igualmente, y dado que normalmente no se necesita conservar las muestras enviadas al conversor D/A, se puede utilizar el mismo buffer para las muestras de entrada y salida, quedando el programa anterior como sigue:

```
#define LBUFF 512                  // longitud de los bloques
int buff1[LBUFF], buff2[LBUFF];
int *iobuffer = buff1, *ioblock = buff2;

void c_int11(void) {
   *opnt = iobuffer[sample];             // se envía muestra procesada
   iobuffer[sample] = *ipnt;             // se lee siguiente muestra
   sample++;                        // se incrementa puntero
}

void wait_buffer(void) {
   int *p;
   while (sample < LBUFF) ;           // espera a que se llene el buffer

   // se intercambian los punteros del bloque y buffer de entrada/salida
   p = iobuffer;
   iobuffer = ioblock;
   ioblock = p;
   sample = 0;                      // se inicializa el puntero
}

void main(void){
   init();
   while (1) {
      wait_buffer() ;                 // espera a que se lean las N muestras
      block_process(ioblock);         // se procesa el bloque
   }
}
```

2.3 Trabajos propuestos

2.3.1 Introducción

El programa `PRAC2.C` genera modulaciones en banda base a partir del vector `input_bits[]` definido e inicializado al inicio del archivo `PRAC2.C`. En primer lugar se realizan las inicializaciones de la placa DSK6713, los conversores, los vectores y algunas variables que configuran el sistema de modulación: amplitud de la señal de salida, muestras asignadas a cada símbolo, variables de estado de los moduladores, etc. A continuación, en un bucle infinito, se van generando bloques de muestras correspondientes a la modulación seleccionada. Específicamente, la función `read_bits()` es la encargada de proporcionar los bits (tantos como indique `usr.nbits`) en la variable `usr.bits`. Esta función controla que una vez agotados los bits del vector `input_bits[]`, estos se repitan de forma cíclica. A continuación se llama a la función de modulación deseada para generar las muestras de salida en el vector `oblock`. La subrutina `wait_buffer ()` junto con la subrutina de interrupción `c_int11()` gestionan el envío de las muestras al conversor D/A.

2.3.2 Trabajo previo

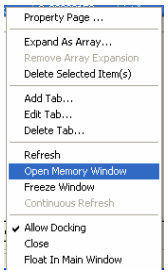
Escriba la funciones de codificación AMI (`cod_ami()`), CMI (`cod_cmi()`), HDB3 (`cod_hdb3()`), Manchester (`cod_manchester()`) y diferencial (`cod_dif()`) para obtener las señales codificadas correspondientes. Para ello complete el código proporcionado en el archivo `CODEC.C` o escriba su propia versión completa.

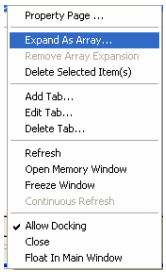
2.3.3 En el laboratorio

A. Compile y enlace el proyecto `prac2.pjt` . Compruebe el funcionamiento del codificador con retorno a cero (`RZ`). Modifique la secuencia de datos de entrada y substituya el codificador `RZ` (`cod_rz()`) por el bipolar (`cod_bipolar()`). Dibuje la señal obtenida en el osciloscopio y compruebe que corresponde con lo esperado. Tenga en cuenta que los amplificadores de salida del conversor DA invierten la señal generada en el DSP, por lo que deberá invertir el canal II del osciloscopio.

B. ¿Cuál es la velocidad de modulación? Modifique sus funciones para obtener una velocidad de modulación de 1,33 kbaudios.

C. Compruebe el funcionamiento de las funciones desarrolladas. Incluya en la libreta el algoritmo utilizado y dibuje con comentarios la salida del osciloscopio. Utilice el depurador de programas DSK6713 para observar el funcionamiento de las subrutinas desarrolladas. Las opciones de depuración están comentadas en la práctica 1, así como la forma de visualizar las variables con Quick Watch. Para visualizar el contenido de un array puede hacerlo de dos formas. Haga clic con el botón derecho del ratón sobre una variable de tipo puntero o array que ya esté en la ventana Quick Watch de CSS. Seleccione una de las dos opciones:





- Volcando el mapa de memoria sobre una ventana (figura izquierda)

- Expandiendo una variable como un array (figura derecha). Deberá introducir el número de elementos del array que desea visualizar.

Si desea transportar la señal generada a otra aplicación, es posible guardar el contenido de un bloque de memoria en un archivo de texto, utilizando el menú `File/Data/Save…`

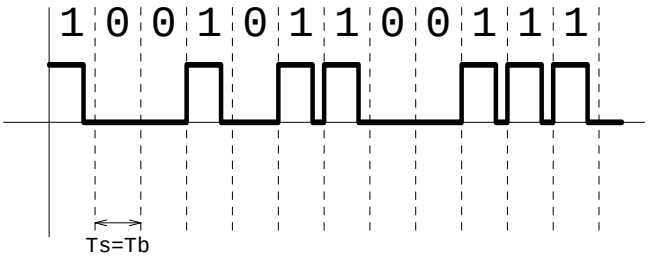
Apéndice A: Modulaciones Banda Base

A continuación se comentan los principios de la generación de las modulaciones banda-base que se emplean en la presente práctica.

Codificador con retorno a cero (RZ)

El codificador con retorno a cero transforma cualquier señal no retorno a cero en una señal con retorno a cero. Para ello el símbolo se divide en dos intervalos. En el primer intervalo del símbolo se transmite la información con el mismo nivel de tensión que tenía en la secuencia NRZ y en el segundo intervalo no hay transferencia de señal (0 V).

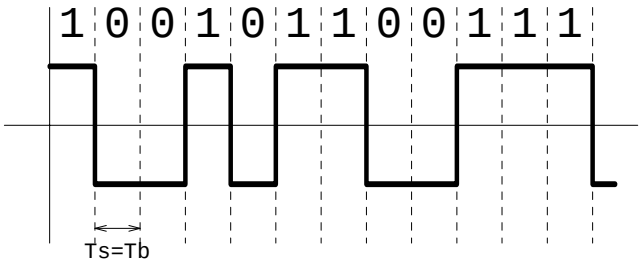
En la figura siguiente se muestra un ejemplo:



Codificador polar

El codificador polar codifica la señal del siguiente modo: Si el símbolo emitido por el generador es un "0", se representa con un valor negativo de tensión (-V). Si el símbolo emitido por el generador es un "1", se representa con un valor positivo de tensión (+V).

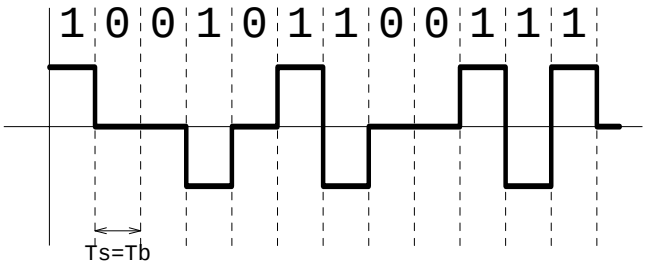
A continuación se puede ver un ejemplo de codificación polar:



Codificador bipolar

El codificador bipolar procede a codificar la señal del siguiente modo: El estado binario "0" se representa con un valor nulo de la señal. El estado binario "1" se representa con valores de tensión positiva (+V) y negativa (-V), alternativamente.

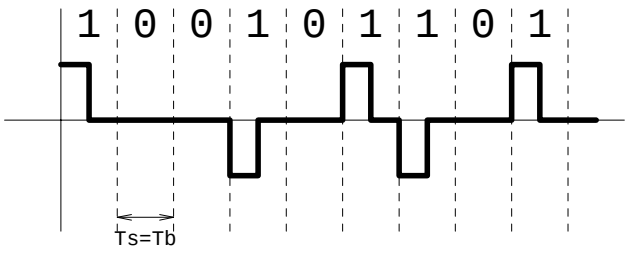
A continuación se muestra un ejemplo de codificación bipolar:



Codificador AMI

El codificador AMI codifica la señal según el procedimiento siguiente: El estado binario "0" se representa por un valor nulo de la señal. El estado binario "1" se representa con pulsos de tensión positiva (+V) y negativa (-V) alternadamente, y con una duración de la mitad del periodo de símbolo.

En la figura que sigue se puede ver un ejemplo de codificación AMI



Codificador HDB3

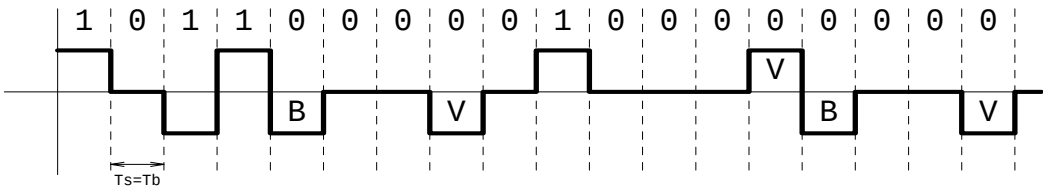
El codificador HDB3 codifica la señal según el siguiente algoritmo: El estado binario "0" se representa mediante un valor nulo de la señal. El estado binario "1" se representa con pulsos positivos (+V) y negativos (-V) alternativamente. Cuatro ceros seguidos se codifican como:

B00V: si el valor de la componente continua, hasta el pulso anterior no es nulo.

000V: si la componente continua hasta el último pulso es cero.

Donde la polaridad de V es la misma que la del último pulso, violando la alternancia de pulsos positivos y negativos. La polaridad de B sigue la alternancia de polaridad.

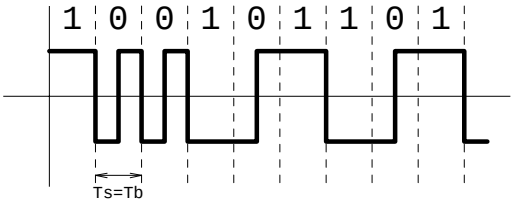
A continuación se muestra un ejemplo ilustrativo de codificación HDB3:



Codificador CMI

El codificador CMI sigue las siguientes reglas: El estado binario "0" se codifica del siguiente modo: El primer semiperiodo del símbolo se codifica con un valor negativo de tensión (-V); el segundo semiperiodo, con tensión positiva (+V). El estado binario "1" se codifica con valores positivos (+V) y negativos (-V) de tensión,

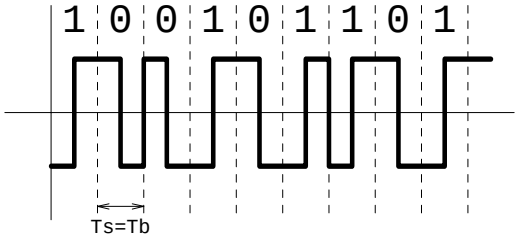
alternadamente, y con duración igual al periodo de símbolo. A continuación se muestra un ejemplo de codificación CMI:



2.4.7 Codificador Manchester

El codificador Manchester funciona según las siguientes características: El estado binario "0" se codifica con una tensión positiva (+V) durante el primer semiperiodo de símbolo y con una tensión negativa (-V) durante el segundo semiperiodo. El estado binario "1" se codifica con una tensión negativa (-V) durante el primer semiperiodo de símbolo y con una tensión positiva (+V) durante el segundo semiperiodo.

En la figura que sigue se muestra una secuencia codificada con código Manchester:



2.4.8 Codificador diferencial

El codificador diferencial permite conseguir la versión diferencial de cualquiera de los codificadores anteriores. Para ello, el codificador diferencial se conecta en cascada, como etapa previa, con el otro codificador. El algoritmo de funcionamiento es el siguiente: Si el símbolo actual es igual al símbolo anterior, a la salida se obtiene el estado binario `"0".` Si el símbolo actual es distinto al símbolo anterior, a la salida se obtiene el estado binario `"1".` Un ejemplo de codificación diferencial es el que sigue:

`1 0 0 1 0 1 1 0 0 1 1 1` → `1 1 0 1 1 1 0 1 0 1 0 0`

2.5 Apéndice B: Archivos utilizados en la práctica 2

```
//************************************************************************
//                                PRAC2.C
//                          Modulaciones banda base
//========================================================================
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "codec.h"
#include "dsk6713_aic23.h"               // Archivos de soporte del DSK6713
#define SIMB_BUFF            100         // simbolos a generar por bloque
#define AMPLITUD          16384
#define MUESTRAS_SIMBOLO 8
#define LBUFF          SIMB_BUFF*MUESTRAS_SIMBOLO
typedef struct {
    int *bits, nbits, index;
} SEQUENCE;

//  Se repiten los primeros ninput_bits de la secuencia input_bits
#define ninput_bits 4
int input_bits[] = {1,1,0,0};
int buffer1[LBUFF], buffer2[LBUFF];
int BITS[SIMB_BUFF + 3];               // +3 ya que el esquema hdb3 necesita 3 bits de
memoria
int *iobuffer = buffer1;
int *oblock = buffer2;
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; //ajuste de la frecuencia de muestreo
int sample=0;

void init_arrays() {
    int i;
    for (i=0;i<LBUFF;i++)
   buffer1[i] = buffer2[i] = 0;
    sample=0;
}
void init_user(USER *pUsr) {
    pUsr->lSymb = MUESTRAS_SIMBOLO;
    pUsr->V = AMPLITUD;
    pUsr->state.lastBit = 0;
    pUsr->state.sgnOne  = 1;
    pUsr->state.dc = 0;
    pUsr->state.zeroes  = 3;
    pUsr->nbits = SIMB_BUFF;
    pUsr->bits = &BITS[3];
    pUsr->bits[-3] = pUsr->bits[-2] = pUsr->bits[-1] = 0;
}

void wait_buffer (void) {
    int *pt;
    while (sample < LBUFF);
    pt = oblock;
    oblock = iobuffer;
    iobuffer = pt;
    sample = 0;
}

void read_bits(SEQUENCE *pSeq, int nbits, int *bits) {
    int i;
    for (i=0; i<nbits; i++) {
        bits[i] = pSeq->bits[pSeq->index++];
        if (pSeq->index == pSeq->nbits)
            pSeq->index = 0;
    }
}

interrupt void c_int11(){
   if (sample<LBUFF)
      output_sample(iobuffer[sample++]);
   return;
}

void main (void) {
   SEQUENCE seq = {input_bits, ninput_bits, 0};
   USER usr;
   init_arrays();
      init_user(&usr);              // Inicio de variable de estado
   comm_intr();              //inicio DSK, codec, McBSP usando interrupciones
    while(1)
    {
        wait_buffer();
        read_bits(&seq, usr.nbits, usr.bits);
```

```
        cod_rz(&usr, oblock, (usr.lSymb)/2);
        //cod_polar (&usr, oblock);
        //cod_bipolar (&usr, oblock);
    }
}
```

Funciones en PRAC2.C

```
//=========================================================================
//                                  Codec.h
//=========================================================================
typedef struct {
    int *bits;           /* If cod_hdb3 is used bits[-3],..,bits[-1]
                            store the value of previos symbols */
    int nbits;
    int lSymb;           /* Samples per symbol */
    int V;               /* Output amplitude   */
    struct {
        int sgnOne;      /* Used to alternate polarity */
        int dc;          /* Used in cod_hdb3 */
        int zeroes;      /* Used in cod_hdb3 */
        int lastBit;     /* Used in cod_dif  */
    } state;
} USER;

void cod_rz     (USER *, int *out, int rz);
void cod_polar  (USER *, int *out);
void cod_bipolar(USER *, int *out);
void cod_ami    (USER *, int *out);
void cod_hdb3   (USER *, int *out);
void cod_cmi    (USER *, int *out);
void cod_manchester (USER *, int *out);
void cod_dif    (USER *);
```

Funciones en CODEC.H

```
//=========================================================================
//         CODEC.C: Codificadores para transmision en banda base
//=========================================================================

#include "codec.h"

void cod_rz (USER *pUsr, int *out, int rz) {
   int *b, i, j, k;
   for (b=pUsr->bits, i=k=0; i < pUsr->nbits; i++) {
      if (b[i] == 0)
    for (j=0; j < pUsr->lSymb; j++,k++)
       out[k] = 0;
      else {
    for (j=0; j< rz; j++,k++)
       out[k] = b[i] * pUsr->V;
    for (;j< pUsr->lSymb;j++, k++)
       out[k] = 0;
      }
   }
}

void cod_polar (USER *pUsr, int *out) {
   int *b, i, j, k, x;
   for (b = pUsr->bits, i=k=0; i < pUsr->nbits; i++) {
      x = ( b[i]==1 ? pUsr->V : -pUsr->V);
      for (j=0; j< pUsr->lSymb; j++, k++)
    out[k] = x;
   }
}

void cod_bipolar (USER *pUsr, int *out) {
   int *b, i, j, k, x;
   for (b=pUsr->bits, i=k=0; i<pUsr->nbits; i++) {
      if (b[i]==1) {
    x = pUsr->V * pUsr->state.sgnOne;
    pUsr->state.sgnOne = -pUsr->state.sgnOne;
      } else
    x = 0;
      for (j=0; j<pUsr->lSymb; j++,k++)
    out[k] = x;
   }
}

void cod_ami (USER *pUsr, int *out) {
          // Completar codigo
}
```

```
void cod_cmi (USER *pUsr, int *out) {
          // Completar codigo
}

void cod_hdb3 (USER *pUsr, int *out) {
//
// PRECONDICION:  en pUsr->bits[-3], pUsr->bits[-2], pUsr->bits[-1],
//                están almacenados los 3 simbolos {+1,0,-1} previos
// POSTCONDICION: los bits en pUsr->bits son cambiados a simbolos {+1,0,-1}
//                El usuario deberia copiarlos si los quiere preservar

   int *b, i, j;
   b=pUsr->bits;
   for (i=0; i<pUsr->nbits; i++) {
      if (b[i]==1) {
    b[i] = pUsr->state.sgnOne;
    pUsr->state.sgnOne = -pUsr->state.sgnOne;
    pUsr->state.dc += b[i];
    pUsr->state.zeroes = 0;
      }
      else {
    pUsr->state.zeroes++;
    if (pUsr->state.zeroes == 4) {
       if (pUsr->state.dc) {
          b[i] = b[i-3] = pUsr->state.sgnOne;

          // Completar codigo para actualizar
        //pUsr->state.sgnOne y pUsr->state.dc

       } else {
          b[i] = -pUsr->state.sgnOne;

         // Completar codigo para actualizar
       //pUsr->state.sgnOne y pUsr->state.dc

       }
       pUsr->state.zeroes = 0;
    }
      }
   }
   //Conformacion de pulso de los 3 bits del tramo anterior y
   //los nuevos (nbits-3)
   // Modificacion de pUsr->bits para poder aprovechar cod_rz
   pUsr->bits = &(pUsr->bits[-3]);
   cod_rz(pUsr, out, pUsr->lSymb);
   // Recuperación de pUsr->bits
   pUsr->bits = b;

   // Guardar los 3 ultimos simbolos
   for (i= -3, j = pUsr->nbits-3; i<0; i++, j++)
       b[i] = b[j];
}

void cod_manchester (USER *pUsr, int *out) {
}

void cod_dif (USER *pUsr) {
}
```

Figura 2.1. Funciones en CODEC.C

3. Uso avanzado del software de desarrollo

3.1 El compilador C para el TMS320C6713

Las opciones de optimización del compilador pueden encontrarse en el documento Software CCS (`spru509f.pdf`) disponible en el servidor del laboratorio. Para modificar sus opciones, debe acceder al menú `Project -> Build Options -> Compiler.` Utilice en esta práctica la opción `Register –o0`.

3.2 El linkador

En documento mencionado en el apartado anterior se encuentra también la información referente al uso del *linkador*. Para modificar sus opciones, debe acceder al menú `Project -> Build Options -> Linker.` Se muestra a continuación un archivo de comandos de linkado. Se pueden observar las siguientes secciones:

- Ubicación de los distintos bloques de memoria y tamaño de ellas.
- Ubicación de las distintas secciones de programa, variables, tablas, vectores de interrupción, stack, etc.

```
/* C6713dsk.cmd  Linker command file   */

MEMORY
{
  IVECS: org=0h, len=0x220               /* interrupt vectors. in internal DSP memory */
  IRAM:  org=0x00000220, len=0x0002FDE0  /* internal DSP memory */
  SDRAM: org=0x80000000, len=0x00100000  /* external memory. Synchronous Dynamic RAM */
  FLASH: org=0x90000000, len=0x00020000  /* flash memory */
}

SECTIONS
{
  .EXT_RAM     :>       SDRAM
  .vectors     :>       IVECS       /* interrupt vectors in Vectors_intr.asm file*/
  .text        :>       IRAM        /* Created by C Compiler*/
  .bss         :>       IRAM
  .cinit       :>       IRAM
  .stack       :>       IRAM
  .sysmem      :>       IRAM
  .const       :>       IRAM
  .switch      :>       IRAM
  .far         :>       IRAM
  .cio         :>       IRAM
  .csldata     :>       IRAM
}
```

Figura 3.1. Archivo de comandos del linker

3.3 El debugger

El *debugger* o depurador de programas DSK6713 permite analizar el funcionamiento de un programa en la placa DSK. Es posible visualizar los contenidos de los registros del TMS320C6713, visualizar y modificar variables en C, ejecutar paso a paso, etc. En definitiva, es la herramienta adecuada para adentrarnos en la ejecución del programa y depurarlo hasta conseguir los objetivos buscados. Además de las funcionalidades comentadas en la práctica 1, se pretende aquí trabajar en los siguientes aspectos:

- Visualización de variables y modificación en tiempo de ejecución
- Display gráfico de valores de un array
- Evaluación del tiempo de ejecución

3.4 Modulaciones en banda trasladada

En esta práctica se generarán modulaciones en banda trasladada así como ruido Gausiano dentro de la banda de los filtros paso-bajo del DSK.

El programa principal se encuentra en el archivo `PRAC3.C`. En ese mismo archivo se encuentras las subrutinas necesarias para leer bits de entrada que se obtienen de forma cíclica a partir de la secuencia `input_bits[]`. También se encuentra la rutina `init_user()`, que inicializa las variables de la estructura `usr` conforme a la modulación deseada, la amplitud de la salida, la relación señal a ruido, el número de muestras por símbolo, etc.

Las subrutinas de modulación se encuentran en el archivo `MODULATE.C`. Las declaraciones de las funciones que ofrece dicho módulo se encuentran en `MODULATE.H` y son incluidas en `PRAC3.C` para que el compilador pueda verificar si son correctas las llamadas a las funciones. Las funciones que generan y añaden ruido a la señal se encuentran en `NOISE.C` y las declaraciones de estas funciones se encuentran en `NOISE.H`.

Las funciones que generan las modulaciones se encuentran en `MODULATE.C` y son las siguientes: `modulator_init(USR_BT *pUsr)`, `modulator_ask_psk(USER_BT *pUsr)` y `mpy_cos(int nSamples, double *phase, double *quad, int *out)`. La primera función es llamada desde `init_user()` e inicializa las variables necesarias dependiendo del tipo de modulación deseada. La segunda función realiza la modulación en banda base de los bits que se entregan en `pUsr->bits`, dejando las componentes en fase y en cuadratura en `pUsr->outp` y en `pUsr->outq`. Finalmente, la función `mpy_cos` crea la señal de salida, en el vector `out`, a partir de las componentes en fase y en cuadratura `phase` y `quad`

La función `add_noise(int nSamples, double stdev, double *xf, double *xq)` añade a la señal de entrada ruido Gaussiano blanco con componentes en fase y cuadratura, apropiado para trabajar con las modulaciones en banda trasladada. Al ejecutar la función se añade ruido de potencia $\texttt{sdev}^2$ a las `nSamples` muestras de las componentes en fase y en cuadratura de la señal que entran a la función en `xf` y `xq`.

El funcionamiento del algoritmo es el siguiente: para cada muestra la función `add_noise` llama a la función `tsgs2`, función que genera parejas de números aleatorios con distribución Gaussiana, media nula y varianza unitaria. La función calcula primero dos números aleatorios con distribución uniforme que se transforman para conseguir la distribución Gausiana. El modificador `inline` antes de la definición de la función hace que el compilador inserte el código de la función en el lugar de la llamada, evitando el coste que supondría realizar una llamada a la función por cada muestra.

3.5 Trabajos previos

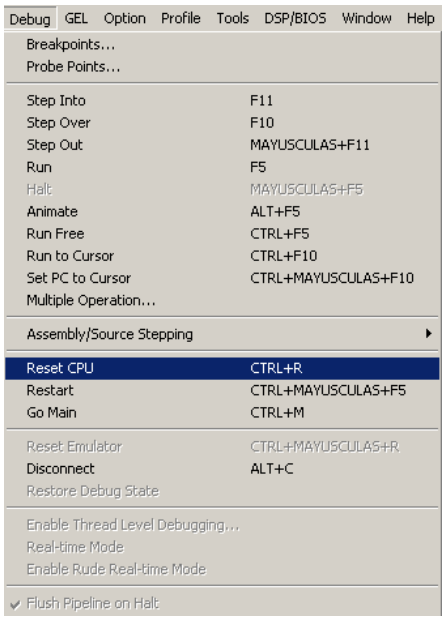
A. Comentar los programas utilizados en la práctica 3.

B. Justificar la expresión que se utiliza para calcula el valor `stdev`, en función de la amplitud y de la relación señal a ruido, para el caso `PSK` y `FSK` (función `modulate_init`). Calcule la potencia de ruido y complete el código para el caso `ASK`.

C. Efectuar una modulación FSK-2, de fase continua. Para ello complete la subrutina `modulator_fsk` de forma que genere la señal correspondiente a esta modulación en banda base. Identifique los valores de frecuencia resultantes asociados a los bits 0 y 1 tanto en Hz como normalizados respecto a la frecuencia de muestreo.

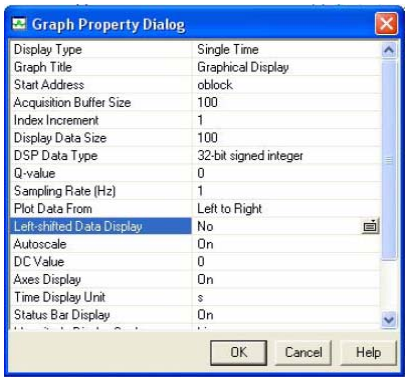
3.6 Trabajos a realizar en el laboratorio

A. Compile y *linke* los programas. Compruebe que se utiliza la opción -g, para permitir el depurado con símbolos. El programa `PRAC3.C` se inicia con una modulación PSK-2 y una secuencia repetitiva del tipo 10101…. Compruebe su funcionamiento con el *debugger*. Utilice los comandos `reset`, `run`, `halt`, `Step Into`, etc...



Comprobar el funcionamiento. Con el comando Quick Watch (haga click con el botón derecho del ratón, sobre la ventana que contiene el código en el entorno CCS), visualizar el contenido de la variable `oblock`. Ver los elementos del array haciendo clic con el botón derecho del ratón, y use las opciones `Expand as array`…, o bien `Open memory window.`

B. Observe gráficamente las muestras de señal generadas en una ventana de CSS. Para ello seleccione el menú `View/Graph/Time/Frequency`…, y en la ventana que se abra incluya la dirección a partir de la cual desea visualizar la señal. En este caso la señal de salida se encuentra en `oblock`, y el propio nombre de la variable es la dirección a la primera posición del array. Asegúrese de que las opciones son las siguientes



Pruebe las modulaciones PSK de 4 y 8 niveles. Para ello modifique el valor inicial de la variable `Lev`, y reconstruya el ejecutable con `Project/Build` y ejecute el programa. Visualice la señal así generada.

C. Utilice ahora la modulación ASK dando el valor `ASK` a la variable `Mod`. Realice las mismas observaciones y medidas que en el apartado anterior.

D. Disminuya la relación señal a ruido. Compruebe el resultado en el osciloscopio y observe los valores de salida utilizando la opción de `Quick Watch` (emplear la modulación PSK-2). Visualice la señal.

E. Calcule el tiempo de ejecución de las funciones descritas anteriormente. Para ello coloque *breakpoints* en el programa en aquellas líneas entre las que desee evaluar el tiempo de ejecución (marcando con el pulsador izquierdo del ratón sobre la línea escogida). Abra la ventana de comandos en el menú `Tools/Command Window` y ejecute el comando `runb` en esa ventana. Los ciclos de ejecución aparecen en la propia ventana. Observe también la duración en periodos de muestreo de la función, al terminar `mpy_cos()`, observando el valor de la variable `sample`.

F. Pruebe la modulación FSK de fase continua desarrollada en el estudio previo. Tenga en cuanta que en este caso debe editar la subrutina `main` del archivo `PRAC3.C` para modificar el valor de la variable `Mod` y recompilarlo. Compruebe que las frecuencias resultantes para cada bit se corresponden a las esperadas. Observe las muestras de señal generada mediante el menú `View/Graph/Time/Frequency`…

3.7 Apéndice: Listado de los programas

```
//*********************************************************************
//                                PRAC3.C
//
//                       Modulaciones banda trasladada
//=============================================
#include <stdlib.h>
#include <math.h>
#include "modulate.h"
#include "noise.h"
#include "dsk6713_aic23.h"          // Archivos de soporte del DSK6713
#define SIMB_BUFF       100         // simbolos a generar por bloque
#define AMPLITUD        16384       // amplitud maxima de componentes en fase y
cuadratura
#define MUESTRAS_SIMBOLO 12
#define LBUFFER         MUESTRAS_SIMBOLO*SIMB_BUFF

#define ILOG2(x)   ( (int) (0.5 + log((double) (x))/log(2.0)) )

typedef struct {
   int *bits;
   int nbits;
   int index;
} SEQUENCE;

#define ninput_bits 2               // Al empezar se tomara la secuencia 10101010...
int input_bits[]={1,0,0,0,0,1,1,1};

int  buffer1[LBUFFER];
int  buffer2[LBUFFER];
int  *iobuffer = buffer1;
int  *oblock   = buffer2;
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; //ajuste de la frecuencia de muestreo
int sample;

interrupt void c_int11() {
   if (sample<LBUFFER)
      output_sample(iobuffer[sample++]);
   return;
}

void init_arrays() {
   int i;
   for (i=0;i<LBUFFER;i++)
      buffer1[i] = buffer2[i] = 0;
   sample=0;
}

void init_user(USER_BT *pUsr, Modulation M, int Levels, double snr) {

   double l_snr;

   pUsr->modulation = M;         // Definicion de los niveles de la modulacion

   if (Levels > MAXLEVELS)
      Levels = MAXLEVELS;
   pUsr->nLevels = Levels;
   pUsr->snr = snr;

   // Inicializacion
   pUsr->lSymb = MUESTRAS_SIMBOLO;
   pUsr->lBuff = LBUFFER;
```

```
   pUsr->nbitsSymb = ILOG2(pUsr->nLevels);
   pUsr->nbits = SIMB_BUFF * pUsr->nbitsSymb;
   if (pUsr->bits != NULL) {
      free (pUsr->bits);
      free (pUsr->cos_fd);
      free (pUsr->sen_fd);
      free (pUsr->outp);
      free (pUsr->outq);
   }
   pUsr->bits  = (int *) malloc(sizeof(int) * pUsr->nbits);
   pUsr->outp  = (double *) malloc(sizeof(double) * pUsr->lBuff);
   pUsr->outq  = (double *) malloc(sizeof(double) * pUsr->lBuff);

   // Las siguientes variables pueden utilizarse para FSK
   // Dependiendo de la implementacion seran preferibles otras definiciones
   pUsr->cos_fd = (double *) malloc(sizeof(double) * pUsr->lSymb);
   pUsr->sen_fd = (double *) malloc(sizeof(double) * pUsr->lSymb);

   // ASK y PSK: Inicializa pUsr->coefP[] y pUsr->coefQ[]
   modulator_init(pUsr);

   pUsr->V = AMPLITUD;
   // Calcular sdev a partir de snr y del esquema de modulacion
   l_snr = pow(10,pUsr->snr/10);
   switch (pUsr->modulation) {
   case ASK:
   // Completar codigo
      break;
   case FSK:
      pUsr->sdev = pUsr->V / sqrt(2*l_snr);
      break;
   case PSK:
      pUsr->sdev = pUsr->V / sqrt(2*l_snr);
      break;
   }
}

void wait_buffer (void) {
   int *pt;
   while (sample < LBUFFER);
   pt = oblock;
   oblock = iobuffer;
   iobuffer = pt;
   sample = 0;
}

void read_bits(SEQUENCE *pSeq, int nbits, int *bits) {
   int i;
   for (i=0; i<nbits; i++) {
      bits[i] = pSeq->bits[pSeq->index++];
      if (pSeq->index == pSeq->nbits)
    pSeq->index = 0;
  }
}

void main (void){
   SEQUENCE seq = {input_bits, ninput_bits, 0};
   USER_BT usr;
   Modulation Mod = PSK;         // Modulacion tipo
   int Lev = 2;                  // Numero de simbolos
   double snr = 100.0;
   init_arrays();                   // Inicio de los arrays
   usr.bits = NULL;
   init_user(&usr, Mod, Lev, snr);  // Configuracion modulador
   comm_intr();                     // inicio del DSK, codec, MCBSP

   while (1)
   {
    wait_buffer();
    read_bits(&seq, usr.nbits, usr.bits);
    if (usr.modulation != FSK)
      modulator_ask_psk(&usr);
    else
      modulator_fsk(&usr);
    if (usr.snr < 90.0)
      add_noise(usr.lBuff, usr.sdev, usr.outp, usr.outq);
    mpy_cos  (usr.lBuff, usr.outp, usr.outq, oblock);
    }
}
```

Figura 3.2. Funciones en PRAC3.C

```
//*************************************************************************
//             MODULATE.C: Moduladores ASK, PSK y FSK
//*************************************************************************
#include "modulate.h"
#include "math.h"
#define SQRT2s2 0.70710678118655 // sin(pi/4)
#ifndef PI
#define PI 3.141592653
#endif

void mpy_cos(int nSamples, double * phase, double *quad, int *x)
{
   static const double Cos[4]= {1.0, 0.0, -1.0,  0.0};
   static const double Sin[4]= {0.0, 1.0,  0.0, -1.0};
   static int j=0;
   int i;
   for (i=0; i<nSamples; i++) {
      x[i]= (int) (phase[i]*Cos[j] - quad[i]*Sin[j]);
      j = (j == 3 ? 0 : j+1);
   }
}

int modulator_init(USER_BT *pUsr)
{
   // GRAY CODE
   int i, error = 0;
   double x;
   switch (pUsr->modulation) {
  case ASK:
      for (i=0;i<pUsr->nLevels;i++)
    pUsr->coefP[i] = pUsr->coefQ[i] = 0.0;

      switch (pUsr->nLevels) {
      case 2:
    pUsr->coefP[0] = 0.0;
    pUsr->coefP[1] = 1.0;
    break;
      case 4:
    x = 1.0/3.0;
    pUsr->coefP[0] = 0.0;
    pUsr->coefP[1] =   x;
    pUsr->coefP[3] = 2*x;
    pUsr->coefP[2] = 1.0;
    break;
      case 8:
    x = 1.0/7.0;
    pUsr->coefP[0] = 0.0;
    pUsr->coefP[1] =   x;
    pUsr->coefP[3] = 2*x;
    pUsr->coefP[2] = 3*x;
    pUsr->coefP[6] = 4*x;
    pUsr->coefP[7] = 5*x;
    pUsr->coefP[5] = 6*x;
    pUsr->coefP[4] = 1.0;
    break;
      default:
   // Error: el modulador no esta preparado
   //   para cualquier numero de niveles
      error = 1;
      }
      break;
   case PSK:
      for (i=0; i< pUsr->nLevels;i++)
     pUsr->coefP[i] = pUsr->coefQ[i] = 0.0;
      switch (pUsr->nLevels) {
      case 2:
    pUsr->coefP[0] =  1.0;
    pUsr->coefP[1] = -1.0;
    break;
      case 4:
    pUsr->coefP[0] =  SQRT2s2; pUsr->coefQ[0] =  SQRT2s2;
    pUsr->coefP[2] = -SQRT2s2; pUsr->coefQ[2] =  SQRT2s2;
    pUsr->coefP[3] = -SQRT2s2; pUsr->coefQ[3] = -SQRT2s2;
    pUsr->coefP[1] =  SQRT2s2; pUsr->coefQ[1] = -SQRT2s2;
    break;
      case 8:
    pUsr->coefP[0] =      1.0; pUsr->coefQ[0] =      0.0;
    pUsr->coefP[1] =  SQRT2s2; pUsr->coefQ[1] =  SQRT2s2;
    pUsr->coefP[3] =      0.0; pUsr->coefQ[3] =      1.0;
    pUsr->coefP[2] = -SQRT2s2; pUsr->coefQ[2] =  SQRT2s2;
    pUsr->coefP[6] =     -1.0; pUsr->coefQ[6] =      0.0;
```

```
     pUsr->coefP[7] = -SQRT2s2; pUsr->coefQ[7] = -SQRT2s2;
     pUsr->coefP[5] =      0.0; pUsr->coefQ[5] =     -1.0;
     pUsr->coefP[4] =  SQRT2s2; pUsr->coefQ[4] = -SQRT2s2;
     break;
       default:
    // Error: el modulador no esta preparado
    //   para cualquier numero de niveles
       error = 1;
       }
       break;
    case FSK:
       // Completar el codigo. Pueden utilizarse las variables
       //pUsr->cos_fd y pUsr->sen_fd
      break;
    }
    return error;
}

void modulator_ask_psk(USER_BT *pUsr) {

   double ampP, ampQ, *outp = pUsr->outp, *outq = pUsr->outq;
   int i, j, k, symb;

   for (i=k=0; i< pUsr->nbits;) {
      symb = pUsr->bits[i++];
      for (j=1; j<pUsr->nbitsSymb;j++)
     symb = 2*symb + pUsr->bits[i++];
      ampP= pUsr->V * pUsr->coefP[symb];
      ampQ= pUsr->V * pUsr->coefQ[symb];

      for (j=0; j < pUsr->lSymb; j++) {
    outp[k] = ampP;
    outq[k++] = ampQ;
      }
   }
}

void modulator_fsk(USER_BT *pUsr) {
//   double *outp = pUsr->outp, *outq = pUsr->outq;
// Completar el codigo
}
```

Figura 3.3. Funciones en MODULATE.C

```
//*********************************************************************
//      NOISE.C: Modulos que suman ruido blanco gaussiano a la
//               senyal de entrada
//*********************************************************************

#include <stdlib.h>
#include <math.h>
#include "noise.h"

inline void tsgs2 (double *g1, double *g2) {
   double u,v,x,env,ph;

   u = (double) rand() /RAND_MAX;
   v = (double) rand() /RAND_MAX;
   x=log(u);
   env=sqrt(-2*x);
   ph=2*PI*v;
   *g1=env*cos(ph);
   *g2=env*sin(ph);
}

void add_noise(int nSamples, double stdev, double *xf, double *xq) {
   int i;
   double g1,g2;

   for (i=0; i<nSamples; i++)  {
      tsgs2(&g1,&g2);
      xf[i] += stdev*g2;
      xq[i] += stdev*g1;
   }
}
```

Figura 3.4. Funciones en NOISE.C

4. Filtrado en tiempo real

4.1 Introducción

En el filtrado digital de señales se puede optar entre filtros con respuesta impulsional finita (FIR) o respuesta impulsional infinita (IIR). Dadas unas especificaciones o plantilla frecuencial a cumplir, la elección de utilizar filtros IIR suele resultar computacionalmente más barata, sobretodo para especificaciones exigentes en cuanto a selectividad. Sin embargo los filtros FIR presentan algunas ventajas frente a los filtros IIR y son utilizados también habitualmente en muchas aplicaciones.

Los filtros FIR permiten obtener una respuesta de fase lineal que es de gran importancia en aplicaciones tales como el filtrado de señales de datos o procesado de la imagen. En algoritmos adaptativos donde los coeficientes del filtro se modifican de forma continua se suelen utilizar también filtros FIR porque su estabilidad está asegurada y porque dan lugar a algoritmos de adaptación con mejor comportamiento.

En esta práctica nos centraremos en el diseño, realización y comparación de filtros FIR e IIR.

4.2 Filtrado por bloques

El procesado de una señal analógica mediante un filtro digital, tal como se muestra en la figura 4.1, puede realizarse muestra a muestra o por bloques. La primera opción requiere un programa principal como el mostrado en la figura 4.2

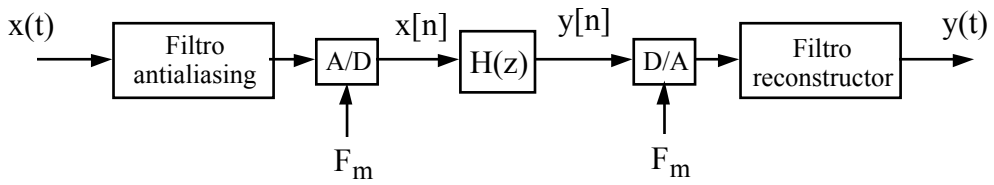


Figura 4.1 Filtrado digital de señales analógicas

```
void main(void)
{
   int input, output;
   init_DSK();                          // Inicialización
   while (1)                            // Bucle principal infinito
   {
      input = io_sample(output);   // Recibe y envía una muestra
      output = filtra(input);           // Procesa muestra
   }
}
```

Figura 4.2. Filtrado FIR muestra a muestra

Este esquema produce el mínimo retardo pero requiere una llamada a la subrutina de filtrado por cada muestra de entrada.

En esquemas más complejos donde el filtrado es sólo una parte del procesado de la señal suele resultar más adecuado realizar un procesado por bloques o tramas de varias muestras. El envío y la recepción de muestras se realiza en este caso en otra subrutina llamada mediante interrupción y debemos tener al menos dos *buffers*: el de entrada/salida de muestras, y el que contiene el bloque de muestras a procesar. El programa principal sigue, en este caso, el esquema mostrado en la figura 4.3.

```
float io1[IO_N], io2[IO_N];        // Reserva espacio para los buffers
float *io_block,*io_buffer;        // Puntero del bloque a procesar
int index=0;

interrupt void c_int11()           // Rutina de servicio de interrupcion
{
   if (index < IO_N) {
      output_sample((int)io_buffer[index]);
      io_buffer[index++]=(float)input_left_sample();
   }
   return;
}

void init_buffer_block(void)
{
    io_buffer = io1;
    io_block  = io2;
}

void wait_buffer_block(void)
{
    float *ptmp;                   // Puntero auxiliar

    while(index < IO_N);           // Espera a que se llene el buffer
    ptmp = io_buffer;              // Intercambia punteros:
    io_buffer = io_block;          // nuevo buffer de entrada/salida
    io_block = ptmp;               // nuevo bloque de datos
    index = 0;                     // Inicializa index
}

void main()
{
   float h[L];

   init_process(h);
   init_buffer_block();
   comm_intr();                    // inicio DSK, codec, McBSP usando interrupciones

   while(1) {                      // bucle infinito
      wait_buffer_block();         // espera a que se llene el buffer
      process(io_block,h);         // filtrado del bloque
   }
}
```

Figura 4.3. Esquema del programa principal de procesado por bloques

En el caso de realizar un filtrado FIR la función `processFIR()` debe obtener las muestras correspondientes a la salida del filtro implementando la ecuación de convolución:

$$y[i] = \sum_{j=0}^{L-1} h[j]x[i-j] \qquad i = 0,...,N-1$$

donde las muestras de la señal entrada, $x[i]$ (i = 0…N-1), se pasan a través del vector `ioblock` al llamar a la función, y las muestras de la señal de salida $y[i]$ se devuelven también en este vector. La figura 4.4 muestra la estructura del filtro FIR:

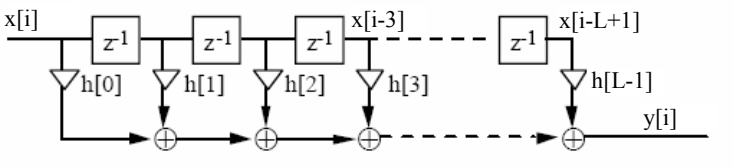


Figura 4.4. Estructura del filtro FIR

Obsérvese que, además de los L coeficientes del filtro $h[j]$ (j = 0…L-1), es necesario disponer de las L-1 muestras anteriores de la señal de entrada $x[i]$ (i = -(L-1)…-1) para poder realizar el cálculo. El programa de la figura 4.5 resuelve este problema incorporando estas muestras anteriores de $x[i]$ al vector `x` que será utilizado como entrada de la función `fir()`.

```
void processFIR(float *block, float *h)
{
    int i;
    static float x[(L-1)+IO_N];                // FIR Input buffer
    for (i=0; i<IO_N; i++) x[L-1+i] = block[i];
    fir(x, block, h, L, IO_N);
    for (i=0; i<(L-1); i++) x[i] = x[i+IO_N];
}
```

Figura 4.5 Subrutina de filtrado de un bloque (en PROCESS.C)

En el caso de realizar un filtrado IIR, la función `processIIR()` obtiene las muestras de salida a partir de muestras pasadas de la salida y de muestras actuales y pasadas de la entrada. La implementación de un filtro IIR que requiere el menor número de retardos es la que se representa en la figura 4.6

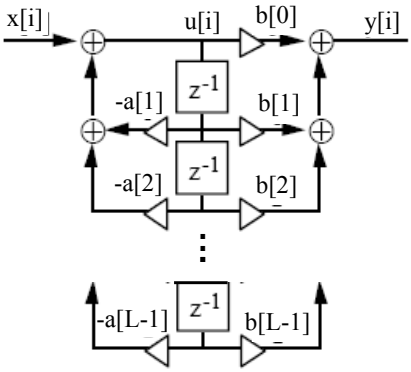


Figura 4.6 Estructura directa II de un filtro IIR

Las variables a_i (b_i) corresponden a los coeficientes del denominador (numerador) de la función de transferencia del filtro, de forma que la relación entre la entrada y la salida puede expresarse a través de un vector auxiliar *u*[]:

$$y[i] = \sum_{j=0}^{L-1} b[j]u[i-j] \qquad i = 0,...,N-1$$

$$u[i] = x[i] - \sum_{j=1}^{L-1} a[j]u[i-j] \tag{1}$$

$$H(z) = \frac{b[0] + b[1]z^{-1} + ... + b[L-1]z^{-L+1}}{1 + a[1]z^{-1} + ... + a[L-1]z^{-L+1}}$$

4.3 Ejercicios previos

1. Escriba en C la función

```
void fir(float *x, float *y, float *h, int k, int n)
```

de filtrado FIR de `n` muestras. Observe que para obtener las `n` muestras de salida *y*[*i*] (*i* = 0…*n*-1) el *array* de entrada `x` debe contener las `n+k-1` muestras *x*[*i*] (*i* = -(*k*-1)…*N*-1) donde `k` es la longitud de la respuesta impulsional *h*[*j*].

2. La función `init_processFIR1()` calcula los coeficientes del filtro FIR mediante el método de ventanas (enventanado de la respuesta impulsional ideal). Compruebe que la respuesta impulsional así obtenida corresponde a un filtro paso alto y determine de forma aproximada su respuesta frecuencial (frecuencia de corte y ancho de la banda de transición) [1].

3. Compruebe que la ecuación (1) responde a la relación entre entrada y salida del filtro IIR representado en la figura 4.6. Cree la función que implementa el filtro `processIIR()`

```
void processIIR(float *x, filter_iir2 *h, int k, int n)
```

en la que la dirección apuntada por x contiene el bloque de muestras a filtrar al entrar a la funcion, y el bloque de muestras filtradas a la salida. h es un apuntador a una estructura de arrays de tipo `filter_iir2`:

```
typedef struct {
   float b[L_IIR];              // Coeficientes b del numerador
   float a[L_IIR];              // Coeficientes a del denominador
   float u[L_IIR];              // Registros de desplazamiento del IIR de orden 2
} filter_iir2;
```

en la que se almacenan los coeficientes y los valores de los registros de desplazamiento del filtro IIR, de acuerdo con la figura 4.6.

4.4 En el laboratorio

A. En el subdirectorio `P:\LC2\PRAC4` encontrará los programas necesarios para realizar la práctica. Edite el archivo `PROCESS.C` y escriba la función `fir()` desarrollada. Compile y enlace el programa con F7. Utilice este programa para filtrar una señal sinusoidal analógica con el filtro FIR analizado en la sección 2 y compruebe que el sistema presenta la respuesta frecuencial determinada en el estudio previo.

B. Con la ayuda de MATLAB compararemos la complejidad en la realización de dos filtros paso-bajo de tipo FIR e IIR de mínimo orden que cumplan las restricciones de una cierta plantilla en frecuencia. Ejecute MATLAB y, una vez dentro, la aplicación FDAtool. Seleccione los parámetros que aparecen en la figura 4.7, para obtener los coeficientes de un filtro FIR paso bajo a las bandas de frecuencias y atenuaciones especificadas.

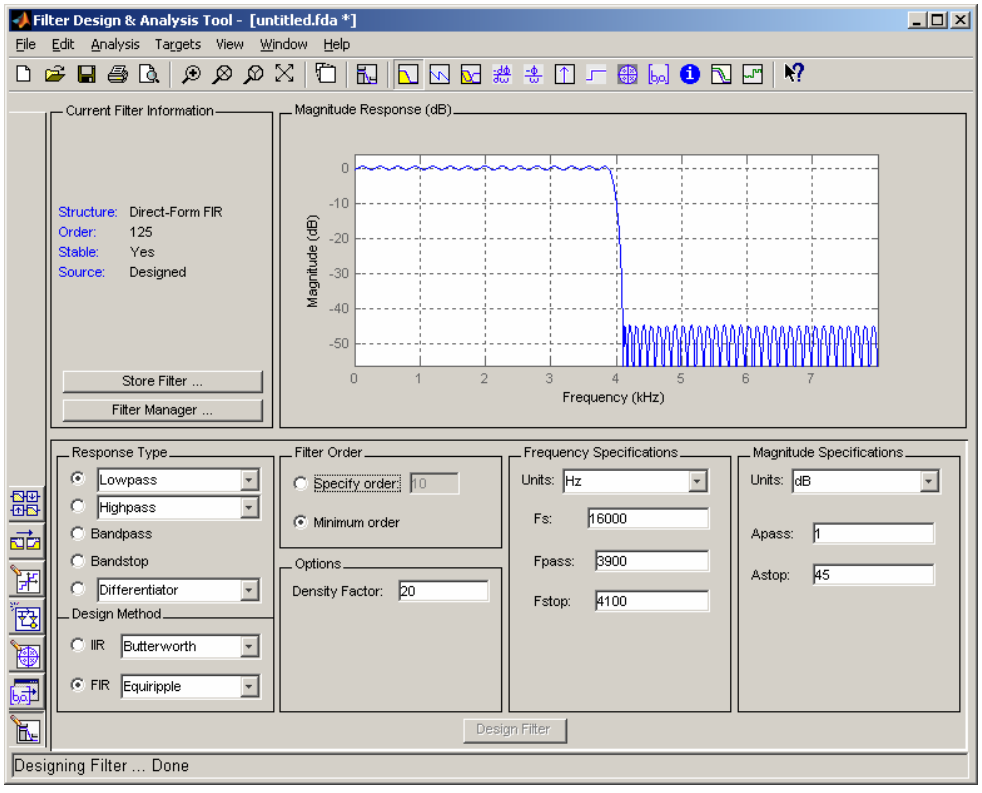


Figura 4.7. El entorno de diseño de filtros FDAtool, de MATLAB. Respuesta frecuencial del filtro FIR

Haga clic sobre el botón "Design Filter" y compruebe que la respuesta en frecuencia corresponde al filtro deseado. Exporte los coeficientes del filtro a un archivo de texto mediante la opción Targets/Generate C header… Desde ese archivo, copie los coeficientes y péguelos en la función `init_processFIR2()`. Modifique en `block.h` el valor de `L_FIR` con el número de coeficientes. Compruebe que el funcionamiento del filtro es el esperado variando la frecuencia de la señal sinusoidal del generador de funciones.

C. Repita el apartado anterior con el filtro IIR mediante la aproximación elíptica ("elliptic"), con los mismos valores de frecuencias y atenuaciones. Seleccione Edit/Convert to Single Section. Exporte los coeficientes a un archivo de texto mediante Targets/Generate C header… y péguelos en la función `init_processIIR()`. Modifique en `block.h` el valor de `L_IIR` con el número de coeficientes. Compruebe que el funcionamiento del filtro es el esperado.

D. Utilice el depurador de programas del `DSK6713` para obtener el coste computacional en ciclos de reloj de las funciones `processFIR()` y `processIIR()`. Calcule el coste computacional por muestra filtrada en cada caso y, desde este punto de vista, justifique el uso de un tipo de filtro u otro.

E. Compruebe que las variables `io1` e `io2` están situadas en la memoria interna del DSP, verificando su dirección de memoria. Modifique su posición desde la memoria interna a la SDRAM. Para ello utilice las directivas `pragma` en el archivo `PRAC4.c`:

```
#pragma DATA_SECTION(io1,".EXT_RAM")
#pragma DATA_SECTION(io2,".EXT_RAM")
```

justo después de declarar `io1` e `io2`. Evalúe el coste computacional de la forma que se hizo en el apartado D y compruebe que el acceso a la memoria SDRAM es más lento que el acceso a la memoria interna del DSP.

F. El uso de opciones de optimización del compilador permite reducir el coste computacional significativamente. Para medir esa reducción realice las siguientes operaciones (hágase sin las directivas pragma del apartado anterior):

1. Modifique el modo de optimización del compilador en Project/Build options…/Compiler. Seleccione Opt Level: -o3. De esta forma se generará código ensamblador con el nivel de optimización máximo.
2. Compile el archivo `process.c` y compruebe que en la carpeta `PRAC4` se ha generado el archivo `process.asm`.
3. Substituya en el proyecto el archivo `process.c` por el archivo `process.asm`. Compile y enlace todo el proyecto de nuevo con la opción habitual de optimización –o0. De esta forma es posible depurar el programa y a la vez utilizar una versión optimizada de las funciones de filtrado.
4. Calcule de nuevo los tiempos de ejecución de las funciones `processFIR()` y `processIIR()` desde la función `main()` .

4.5 Referencias

[1] Oppenheim, A. V., Schafer, R. W.; *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

4.5 Anexo

```
#include "dsk6713_aic23.h"         // Soporte codec-DSK
#include "block.h"
#include "process.h"

Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; // Ajuste de la frecuencia de muestreo

float io1[N], io2[N];              // Reserva espacio para los buffers

float *io_block,*io_buffer;        // Puntero del bloque a procesar
int index=0;

interrupt void c_int11()           // Rutina de servicio de interrupcion
{
   if (index < N) {
      output_sample((int)io_buffer[index]);
      io_buffer[index++]=(float)input_left_sample();
   }
   return;
}

void init_buffer_block(void)
{
    io_buffer = io1;
    io_block  = io2;
}

void wait_buffer_block(void)
{
    float *ptmp;                           // Puntero auxiliar

    while(index < N);                      // Espera a que se llene el buffer
    ptmp = io_buffer;                      // Intercambia punteros:
    io_buffer = io_block;                  // nuevo buffer de entrada/salida
    io_block = ptmp;                       // nuevo bloque de datos
    index = 0;                             // Inicializa index
}

void main()
{
   float hFIR[L_FIR];                      // Respuesta impulsional del filtro FIR
// filter_iir2 hIIR;                      // Coeficientes del filtro IIR

   init_processFIR1(hFIR);         // Filtro paso alto
// init_processFIR2(hFIR);         // Filtro generado con MATLAB
// init_processIIR(&hIIR);         // Filtro generado con MATLAB

   init_buffer_block();
   comm_intr();                    // inicio DSK, codec, McBSP usando interrupciones

   while(1)                        // bucle infinito
   {
      wait_buffer_block();                 // Espera a que se llene el buffer
      processFIR(io_block,hFIR);           // Procesa bloque
//    processIIR(io_block,&hIIR,L_IIR,N);  // Procesa bloque
   }
}
```

Figura 4.8. PRAC4.C (las líneas de la función main() correspondientes al filtrado IIR aparecen comentadas)

```
#include <math.h>
#include <stdlib.h>
#include "block.h"

#define PI 3.141592654
#define WC (PI/2)

float sinc(float s)
{
    if (s==0)
      return(1);
    else
      return(sin(s)/s);
}

void init_processFIR1(float *h)     // Filtrado FIR paso alto obtenido mediante
                                    // enventadado de Hamming
{
    int n;

    for (n=0; n<L_FIR; n++) {
      h[n] = -(WC/PI)*sinc(WC*(n-(L_FIR-1)/2));      // Highpass filter
      if (n==(L_FIR-1)/2) h[n] = h[n] + 1;
      h[n] *= (0.54 - 0.46*cos(2*PI*n/L_FIR));       // Hamming window
    }
}

void init_processFIR2(float *h)     // Filtrado FIR con coeficientes obtenidos en MATLAB
{

// Pegue entre corchetes en la definición de hh la lista de los coeficientes
//obtenidos con MATLAB
    float hh[]={};
    int n;

    for (n=0; n<L_FIR; n++)
      h[n] = hh[n];

}

void fir(float *x, float *y, float *h, int k, int n)
{
// Complete el código
}


void processFIR(float *block, float *h)
{
    int i;
    static float x[(L_FIR-1)+N];               // FIR Input buffer

    for (i=0; i<N; i++) x[L_FIR-1+i] = block[i];
    fir(x, block, h, L_FIR, N);
    for (i=0; i<(L_FIR-1); i++) x[i] = x[i+N];
}

void init_processIIR(filter_iir2 *h)
{
   int i;

// Pegue entre corchetes en la definición de a y de b la lista de los coeficientes
obtenidos con MATLAB
   float a[]={};
   float b[]={};

   for (i=0; i<L_IIR; i++){
      h->a[i]=a[i];
      h->b[i]=b[i];
      h->u[i]=0.0;
   }
}

void processIIR(float *x, filter_iir2 *h, int k, int n)
{
// Complete el código
}
```

Figura 4.9. PROCESS.C

```
#define N         1000
#define L_FIR     129     // Orden del filtro FIR + 1
#define L_IIR     1       // Orden del filtro IIR + 1

typedef struct {          // Estructura para filtrado IIR
   float b[L_IIR];        // Coeficientes b del numerador
   float a[L_IIR];        // Coeficientes a del denominador
   float u[L_IIR];        // Registros de desplazamiento del IIR de orden 2
} filter_iir2;
```

Figura 4.10. BLOCK.H

```
void init_processFIR(float *);
void init_processIIR(filter_iir2 *);
void processFIR(float *, float *);
void processIIR(float *, filter_iir2 *, int, int);
```

Figura 4.11. PROCESS.H

5. La Transformada Rápida de Fourier (FFT)

5.1 Filtrado FIR usando la FFT

En la práctica anterior se ha visto como realizar un filtrado digital. En esta práctica utilizaremos otra operación habitual en procesado de la señal, la Transformada Discreta de Fourier (DFT) [1,2]. En primer lugar estudiaremos su aplicación dentro del mismo contexto de la práctica anterior: el filtrado FIR.

Cuando en algunas aplicaciones se desea filtrar mediante filtros FIR de gran longitud la aplicación directa de la ecuación de convolución

$$y[i] = \sum_{j=0}^{L-1} h[j]x[i-j] \qquad i = 0,...,N-1 \tag{1}$$

puede representar una carga computacional excesiva. En esta práctica se muestra una forma alternativa de realizar este filtrado con un coste considerablemente inferior.

Un resultado básico de los sistemas lineales e invariantes es que un filtrado o convolución en el dominio del tiempo es equivalente a una multiplicación en el dominio transformado. Dado que los algoritmos FFT (*Fast Fourier Transform*) proporcionan una forma eficiente de cálculo de la DFT se puede pensar en obtener un filtrado mediante un producto en el dominio transformado en vez de una convolución en el dominio del tiempo.

En efecto, si dividimos la señal de entrada en bloques $x_i[n]$ de M muestras

$$x_i[n] = \begin{cases} x[n+iM] & n = 0,...,M-1 \\ 0 & \text{otro valor de } n \end{cases} \tag{2}$$

y el número de puntos N de la FFT es suficientemente grande el cálculo de la convolución circular de este bloque $x_i[n]$ con la respuesta impulsional del filtro $h[n]$

$$y_i[n] = x_i[n] \textcircled{N} h[n] = \text{IDFT}\{ X_i[k]\, H[k]\} \tag{3}$$

coincide con la convolución lineal:

$$y_i[n] = x_i[n] * h[n] \tag{4}$$

En este caso podemos obtener la señal de salida $y[n]$ como suma de la salidas $y_i[n]$ correspondiente a cada bloque. En efecto, dado que $x[n]$ se puede expresar como la suma de los bloques $x_i[n]$ de M muestras

$$x[n] = \sum_i x_i[n - iM] \tag{5}$$

la salida será la suma de la salida correspondiente a cada bloque

$$y[n] = x[n] * h[n] = \sum_i x_i[n - iM] * h[n] = \sum_i y_i[n - iM] \tag{6}$$

A la hora de implementar la ecuación (6) habrá que tener en cuenta que la longitud de $y_i[n]$ es mayor de M muestras por lo que habrá que sumar las primeras muestras del bloque actual con las últimas muestras del bloque anterior.

Para el cálculo del coste computacional debemos tener en cuenta que la realización de la convolución circular en el dominio de la frecuencia requiere el cálculo de $X_i[k]$ (DFT de N puntos de $x_i[n]$), la multiplicación de $X_i[k]$ con H[k] y el cálculo de la DFT inversa (IDFT) de este producto. Observe que el cálculo de H[k] (DFT de N puntos de h[n]) sólo es necesario realizarlo una vez y por tanto no interviene en el cálculo del coste computacional del filtrado. Una descripción más detallada de este procedimiento de filtrado por bloques y otra variante similar puede encontrarse en [1,2].

5.2 Ejercicios previos

1. Determine el valor adecuado de M en función de L y N para el que el procedimiento descrito anteriormente sea correcto.

2. Calcular el coste computacional en número de multiplicaciones por muestra c(v) que precisa este método si se hace uso del algoritmo FFT con $N = 2^v$. Para L = 126, obtener el valor de v que minimiza c(v). Comparar el coste mínimo obtenido con el que requiere el cálculo directo de la convolución. Considerar que tanto la FFT directa como la inversa necesitan $(N/2)\log_2 N$ multiplicaciones complejas.

3. Escriba una función en C

```
cc(float *x, float *H)
```

que calcule la convolución circular de `N` puntos de `x` con `h`. Suponga que el vector de entrada `H` contiene ya la DFT de `N` puntos de h, y almacene el resultado en el mismo vector `x` de entrada

$$x = x[n] \textcircled{N} h[n] = \text{IDFT}\{X[k]\, H[k]\}$$

Tenga en cuenta que el resultado de la FFT es un vector de complejos según el formato descrito en el apéndice. Complete también la función `process()` para realizar un filtrado FIR mediante la FFT. Utilice el archivo `process.c` mostrado en la figura 5.1 como punto de partida, así como el programa principal de procesado por bloques (`block.c`). Utilice la constante `M` para referirse a la longitud de cada bloque de muestras y `NFFT` para referirse a la longitud de la FFT.

5.3 En el laboratorio

A. En el subdirectorio `P:\LC2\PRAC5` encontrará los programas necesarios para realizar la práctica. Cree en su zona un subdirectorio con este nombre y copie los archivos correspondientes. Edite el archivo `process.c` y complete las funciones `process()` y `cc()` desarrolladas. Para ello debe entender de qué forma está representada la señal en la función `fftr()` y cómo devuelve las muestras de la DFT (ver figura 5.3). Utilice los coeficientes del filtro paso bajo FIR obtenido con MATLAB en la práctica 4. A continuación compile y enlace. Utilice este programa para filtrar una señal sinusoidal analógica y compruebe que el sistema se comporta adecuadamente.

B. Utilice el depurador `DSK6713` para calcular el coste computacional por muestra filtrada (en número de ciclos de reloj) y compárelo con el de la función `fir()` de la practica anterior.

C. Tal como se hizo en la práctica 4, utilice la opción –o3 del compilador para generar los archivos `process.asm` y `fftr.asm`. Modifique el proyecto para que sustituyan a los respectivos `process.c` y `fftr.c`. Compile y enlace todo el proyecto de nuevo con la opción habitual de optimización –o0. y compruebe los tiempos de ejecución desde la función `main()`.

5.4 Referencias

[1] Oppenheim, A. V.; Schafer, R. W.; *Discrete-Time Signal Processing*, Prentice-Hall, Englewwod Cliffs, NJ, 1989. Section 8.9.3 Implementing Linear Time-Invariant Systems using the DFT.

[2] Mariño, J.B.; Vallverdú, F.; Fonollosa, José A. R.; Moreno, A.; Procesado Digital de la Señal: Una introducción Experimental. Edicions UPC. Problema 2.17.

```
#include <math.h>
#include <stdlib.h>
#include "block.h"
#include "fftr.h"

float h[NFFT];
float x[NFFT];

void init_process()                          //   Filtrado   FIR   con   coeficientes
obtenidos en MATLAB
{

// Pegue entre los corchetes en la definición de hh la lista de los coeficientes
//obtenidos con MATLAB

   float hh[]={};
      int n;

      for (n=0; n<L_FIR; n++)
      h[n] = hh[n];
   for (; n<NFFT; n++)
      h[n] = 0.0;
   fftr(h, NFFT, 1);

}

void cc(float *y,float *H)
{
    // Complete el código

    fftr(y, NFFT, 1);                        //  FFT directa
    // Complete el código

    fftr(y, NFFT, -1);                       //  FFT inversa
}


void process(float *block)
{
    // Complete el código

    cc(x,h);
    // Complete el código

}
```

Figura 5.1. Filtrado FIR usando la FFT, en PROCESS.C

```
#define NFFT     1024
#define L_FIR    126              // Orden del filtro FIR + 1
#define M        (NFFT-L_FIR+1)
```

Figura 5.2. BLOCK.H

```
void fftr(float f[], int N, int forward)
/* Fourier transform of an array of N real numbers

PARAMETERS
   f:     At the input (output for the IDFT), the array contains N real samples
          At the output (input for the IDFT), it contains the samples of the DFT,
ordered as follows:
            f[0]                  DFT(0)
            f[1]                  DFT(N/2)
            f[2]                  Real(DFT(1))
            f[3]                  Imag(DFT(1))
            ...
            f[N-2]                Real(DFT(N/2-1))
            f[N-1]                Imag(DFT(N/2-1))
   N:     Number of real samples (size of f). Must be a power of 2
   forward: Determines whether to do a DFT (1) or an IDFT (-1)
*/

{
  int b;
  float pi2n = 4.*asin(1.)/N,cospi2n=cos(pi2n),sinpi2n=sin(pi2n); /* pi2n = 2 Pi/N */
  struct complex wb;               /* wb = W^b = e^(2 pi i b/N) in the Danielson-Lanczos
formula for a transform of length N */
  struct complex temp1,temp2;      /* Buffers for implementing recursive formulas */
  struct complex *c = (struct complex *)f;     /* Treat f as an array of N/2 complex
numbers */

  if(forward==1)
    fftc(f,N/2,1);                                                          /*   Do   a
transform of f as if it were N/2 complex points */

  wb.real = 1.;                                                             /*
Initialize W^b for b=0 */
  wb.imag = 0.;
  for(b=1;b<N/4;b++)                           /* Loop over elements of transform */
  {
    temp1 = wb;
    wb.real = cospi2n*temp1.real - sinpi2n*temp1.imag;                      /*      Real
part of e^(2 pi i b/N) used in D-L formula */
    wb.imag = cospi2n*temp1.imag + sinpi2n*temp1.real;                      /*
Imaginary part of e^(2 pi i b/N) used in D-L formula */
    temp1 = c[b];
    temp2 = c[N/2-b];
    c[b].real=.5*(temp1.real+temp2.real+forward*wb.real*(temp1.imag+temp2.imag)        +
wb.imag*(temp1.real-temp2.real));
    c[b].imag=.5*(temp1.imag-temp2.imag-forward*wb.real*(temp1.real-
temp2.real)+wb.imag*(temp1.imag+temp2.imag));
    c[N/2-b].real=.5*(temp1.real+temp2.real-forward*wb.real*(temp1.imag+temp2.imag)-
wb.imag*(temp1.real-temp2.real));
    c[N/2-b].imag=.5*(-temp1.imag+temp2.imag-forward*wb.real*(temp1.real-
temp2.real)+wb.imag*(temp1.imag+temp2.imag));
  }
  temp1 = c[0];
  c[0].real = temp1.real+temp1.imag;           /* Set b=0 term in transform */
  c[0].imag = temp1.real-temp1.imag;           /*  Put  b=N/2  term  in  imaginary  part  of
first term */

  if(forward==-1)
  {
    c[0].real *= .5;
    c[0].imag *= .5;
    fftc(f,N/2,-1);
  }
}
```

Figura 5.3. Función FFTR ()

6. Generación de tonos. Análisis espectral mediante la DFT

6.1 Objetivos

Esta práctica tiene un doble objetivo. En primer lugar se estudian diferentes maneras de generar una sinusoide de forma digital. A continuación se propone utilizar la Transformada Discreta de Fourier (DFT) para realizar un análisis espectral de la señal obtenida a través del conversor A/D.

6.2 Generación de tonos

La generación digital de una sinusoide puede realizarse de diferentes maneras:

1. **Aproximación polinómica**. La funciones trigonométricas disponibles en la mayoría de librerías matemáticas se basan en una aproximación polinómica adecuadamente escogida. *Texas Instruments* proporciona una versión en C que aparece recogida en la figura 6.1. El generar una sinusoide a partir de la función seno es inmediato aunque debemos evitar que el argumento tome un valor muy alto. Por ejemplo, dada la pulsación discreta ω y la fase inicial p, el código de generación en C de `ns` muestras de una sinusoide podría ser

```
for (i=0; i<ns; i++)
{
   x[i] = sin(p);
   if ((p += w) >= PI2) p -= PI2;
}
```

2. **De forma recurrente**. La muestra enésima de la señal exponencial compleja $x[n] = e^{j\omega n}$ se puede expresar fácilmente en función de la muestra anterior como

$$x[n] = e^{j\omega n} = e^{j\omega}\, e^{j\omega(n-1)} = e^{j\omega}\, x[n-1] \qquad (6.1)$$

   Esta ecuación nos permite generar un coseno y un seno de forma recurrente mediante una sola multiplicación compleja por muestra. Si únicamente nos interesa una componente podemos reducir aún más el coste computacional. En efecto, la señal $x[n] = A\,\mathrm{sen}(\omega n + \theta)$ puede generarse mediante la ecuación de recurrencia de segundo orden

$$x[n] = a\, x[n-1] - x[n-2] \qquad (6.2)$$

   donde a es un coeficiente que depende de la frecuencia.

3. **Mediante una tabla**. A partir de una tabla con N muestras equiespaciadas de un periodo de la función seno se puede generar cualquier frecuencia discreta f = k/N. Para ello sólo es necesario recorrer la tabla de forma cíclica de k en k muestras, por ejemplo, mediante el siguiente algoritmo:

```
p = 0;
for (i=0; i<ns; i++)
{
   x[i] = tabla[p];
   if ((p += k) >= N) p -= N;
}
```

   Este método está restringido a sinusoides de frecuencia discreta racional (f = k/N), y aunque el coste computacional es mínimo, su coste en posiciones de memoria es proporcional a N, el periodo de la secuencia discreta sinusoidal generada.

6.3 Ejercicios previos

1. La ecuación recurrente (6.2) puede interpretarse como la respuesta de un filtro IIR a una entrada nula. Obtenga la posición de los polos de este filtro sobre el plano z (módulo y argumento) y relacione este resultado con la frecuencia de la sinusoide generada. Obtenga también las condiciones iniciales adecuadas

(x[-1] y x[-2]) para generar la sinusoide x[n] = A sen(ωn + θ) de forma recurrente mediante la ecuación (6.2). ¿Cuál es la frecuencia, fase y amplitud de una sinusoide generada con a= -1, x[-1]=-1000 y x[-2]= 1000 ?

2. Escriba en C la función

```
rsin(int ns, float *x, float a, float state[2])
```

que genere una sinusoide de `ns` muestras de forma recurrente a partir de las condiciones iniciales `state[2]={x[-1], x[-2]}`. La función debe también actualizar el vector `state` para que contenga a la salida las dos últimas muestras generadas.

3. Escriba en C la función

```
spec (float *x, float *w, float *s, int ns)
```

que realice un análisis espectral en dB de la secuencia x[n] mediante un enventanado con la secuencia w[n]

$$y[n] = \begin{cases} x[n]w[n] & n = 0,\ldots, IO\_N-1 \\ 0 & n = IO\_N,..., ns-1 \end{cases}$$

y la aplicación posterior de la función FFT (en `fftr.c`) proporcionada en la práctica anterior:

$$Y[k] = DFT\{y[n]\}$$

$$s[k] = 10*\log10(|Y[k]|^2/ns) \qquad k = 0, \ldots, ns/2$$

4. Analice la función `process()` contenida en el archivo `PROCESS.C`.

5. Complete el código en C de la función `init_process` con el objeto de utilizar todas las ventanas que se muestran en la tabla 6.1.

Ventana	**w[n] 0≤n<M**	**NLPS (dB)**	**Ancho del lóbulo principal**
Rectangular	1	13	4π/(M+1)
Barlett	2n/M 0≤n≤M/2 2-2n/M M/2≤n<M	25	8π/M
Hanning	0.5-0.5cos(2πn/M)	31	8π/M
Hamming	0.54-0.46cos(2πn/M)	41	8π/M
Blackman	0.42-0.5cos(2πn/M) +0.08cos(4πn/M)	57	12π/M

Tabla 6.1. Características de las ventanas a utilizar

```
/*********************************************************************      */
/*  sin    v4.50                                                             */
/*  Copyright (c) 1992 Texas Instruments Incorporated                        */
/*********************************************************************      */
#include <math.h>
#include <values.h>
/*********************************************************************      */
/*  SIN() - sine                                                             */
/*                                          */
/*  Based on the algorithm from "Software Manual for the Elementary */
/*  Functions", Cody and Waite, Prentice Hall 1980, chapter 8.               */
/*                                          */
/*  N = round(x / PI)                                                        */
/*  f = x - N * PI                                                             */
/*  g = f * f                                                                */
/*  R = polynomial expansion                */
/*  result = f + f * R                                                       */
/*                                          */
/*  if x < 0, result = - result             */
/*  if N is even, result = - result         */
/*                                          */
/*  This will return the wrong result for x >= MAXINT * PI                   */
/*********************************************************************      */
double sin(double x)
{
   double d, y, xn, f, g, rg;
   float  sgn = (x < 0) ? -1.0 : 1.0;
   int n;

   x = fabs(x);
   n = (int) ((x * INVSPI) + 0.5);
   xn = (double) n;

   /* if n is odd, negate the sign             */
   if (n % 2) sgn = -sgn;

   /* f = x - xn * PI (but mathematically more stable)                      */
   f = (x - xn * C1) - xn * C2;

   /* determine polynomial expression                                       */
   g = f * f;

#if BITS<=24
   rg = (((R4 * g + R3) * g + R2) * g + R1) * g;
#elif BITS>=25 && BITS<=32
   rg = ((((R5 * g + R4) * g + R3) * g + R2) * g + R1) * g;
#elif BITS>=33 && BITS<=50
   rg = ((((((R7*g+R6)*g+R5)*g+R4)*g+R3)*g+R2)*g+R1)*g;
#else
   rg = (((((((R8*g+R7)*g+R6)*g+R5)*g+R4)*g+R3)*g+R2)*g+R1)*g;
#endif
   return (sgn * (f + f * rg));
}
```

Figura 6.1. Listado de la función seno en C

6.4 En el laboratorio

A. En el subdirectorio `P:\LC2\PRAC6` encontrará los programas necesarios para realizar la práctica. Cree en su zona o en el escritorio de su ordenador el subdirectorio `PRAC6` y copie los archivos correspondientes. Edite los archivos `RSIN.C` y `SPEC.C` para introducir el código desarrollado en el estudio previo. A continuación compile mediante la tecla `F7`. Ejecute el programa en la tarjeta de desarrollo `DSK6713`. Compruebe que obtiene la salida analógica correspondiente a la sinusoide generada digitalmente.

B. La función `process()` realiza también un análisis espectral de la suma de la señal de entrada más la sinusoide generada. Visualice el espectro generado mediante la utilidad `View/Graph`… de CCS:

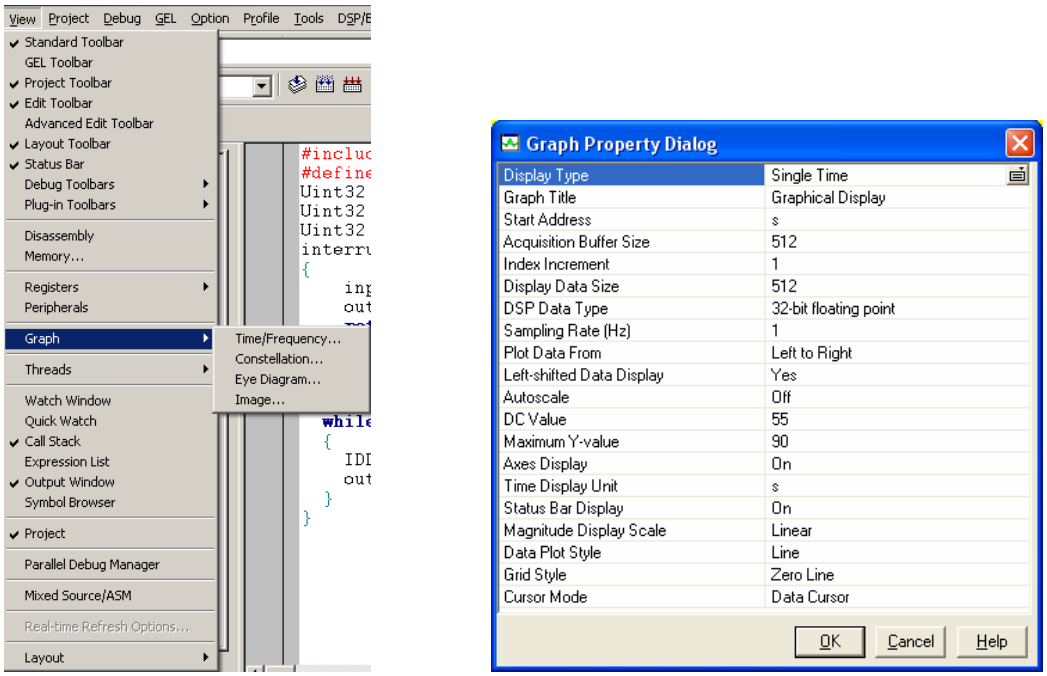


Figura 6.2. Parámetros de la representación gráfica del espectro

Para ello sitúe el ratón sobre la línea de código posterior a la función `spec` en `process.c` e inserte un gráfico que represente la variable `s`, pulsando el botón derecho. Compruebe que los parámetros del gráfico son los mostrados en la figura 6.2. Ejecute el programa y compruebe que el gráfico se va actualizando en el tiempo. Junto a la línea de código debe haber aparecido un rombo azul que indica un punto de sondeo (*probe point*). Utilice el generador de funciones para introducir una sinusoide por el conversor A/D y observe el resultado del análisis espectral. Varíe la amplitud de la señal de entrada hasta hacerla coincidir con la amplitud de la sinusoide generada internamente. Obtenga de forma aproximada la resolución del analizador espectral, midiendo con el multímetro las frecuencias de las dos sinusoides cuando éstas empiezan a ser distinguibles en el espectro. Para ello deshabilite el punto de sondeo usando el menú `Debug/Probe points`… Compruebe que los resultados obtenidos coinciden con los teóricos.

C. Pruebe el efecto de otras ventanas en el análisis espectral, generando para ello las correspondientes expresiones en `init_process`. Para cada una de las ventanas mida aproximadamente la resolución frecuencial.

D. Escoja dos ventanas de las propuestas en la tabla y mida por medio del depurador los parámetros NLPS y Ancho del lóbulo principal en la variable `s`. Para comprobar que coinciden con los teóricos anule totalmente la sinusoide de la entrada A/D (es decir, no sume la señal obtenida del conversor A/D a la señal generada `x` en `process.c`) y cambie el valor de la frecuencia de la sinusoide generada internamente a $f_m/4$, siendo f_m la frecuencia de muestreo. Comente porqué es conveniente el cambio de frecuencia propuesto para la sinusoide en este apartado.

6.5 Apéndice

```
#include "dsk6713_aic23.h"                // Soporte codec-DSK
#include "block.h"
#include "process.h"

Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; // Ajuste de la frecuencia de muestreo

float io1[IO_N], io2[IO_N];              // Reserva espacio para los buffers

float *io_block,*io_buffer;              // Puntero del bloque a procesar
int index=0;

interrupt void c_int11()                 // Rutina de servicio de interrupcion
{
   if (index < IO_N)
   {
      output_sample((int)io_buffer[index]);
      io_buffer[index++]=(float)input_left_sample();
   }
   return;
}

void init_buffer(void)
{
    io_buffer = io1;
    io_block  = io2;
}

void wait_buffer(void)
{
    float *ptmp;                            // Puntero auxiliar

    while(index < IO_N);                 // Espera a que se llene el buffer
    ptmp = io_buffer;                    // Intercambia punteros:
    io_buffer = io_block;                // nuevo buffer de entrada/salida
    io_block = ptmp;                     // nuevo bloque de datos
    index = 0;                           // Inicializa index
}

void main()
{
   window win=hamm;

   init_process(win);                    // inicializa la ventana
   init_buffer();                        // inicializa los buffers
   comm_intr();                          //    inicio    DSK,    codec,    McBSP    usando
interrupciones

   while(1)                              // bucle infinito
   {
      wait_buffer();                     // Espera a que se llene el buffer
      process(io_block);                 // Procesa bloque
   }
}
```

Figura 6.3. PRAC6.C

```
#include <math.h>
#include <stdlib.h>
#include "block.h"

float x[IO_N];                                  // Generated sinusoid
float y[NFFT];                                  // Sum of generated sinusoid and signal
from ADC
float w[NFFT];                                  // Window
float s[NFFT/2+1];                                 // Contains the estimated spectrum

void init_process(window win)               // Window initialisation
{
    int i;
    float theta;

    theta = (2*PI)/IO_N;

    // Crear tipo de ventana
    switch (win){
      case rect:
   // COMPLETAR CODIGO Rectangular window
       break;
      case barl:
   // COMPLETAR CODIGO Barlett window
       break;
      case hann:
   // COMPLETAR CODIGO  Hanning window
       break;
      case hamm:
         for (i=0; i<IO_N; i++)
            w[i] = 0.54 - 0.46*cos(i*theta);    // Hamming window
       break;
      case blac:
   // COMPLETAR CODIGO Blackman window
      break;
    }

    for (; i<NFFT; i++) w[i] = 0;

}

void rsin(int ns, float *x, float a, float state[2]) {
   // COMPLETAR CODIGO
}

void spec(float *x, float *w, float *s, int ns) {
   // COMPLETAR CODIGO
}

void process(float *block)
{
    int i;
    static float state[2] = {-10000,10000};
    static float a = -1.0;

    // Genera una sinusoide de forma recurrente
    rsin(IO_N, x, a, state);

    // Suma la señal generada a la recogida del conversor A/D
    for (i=0; i<IO_N; i++)
      y[i] = x[i] + block[i];
    for (; i<NFFT; i++)
      y[i] = 0.0;

    // Analisis espectral
    spec(y, w, s, NFFT);

   // Env¡a al conversor D/A la sinusoide generada
    for (i=0; i<IO_N; i++)
      block[i] = x[i];

}
```

Figura 6.4. PROCESS.C