CD INCLUIDO

ERA REVISTA DE PROGRAMACIÓN EN CASTELLANO

SOPROGRAMADORES

AÑO VIII. 2.ª ÉPOCA • NÚMERO 81 • UNA PUBLICACIÓN DE: REVISTAS PROFESIONALES S.L. • Precio: 995 Ptas. 5,98 E (España) (IVA incluido)

GENERACIÓN DE TERRENOS

3D AL LÍMITE

web mail con POP3

LABORATORIO
SONDAS DE TEMPERATURA

RED
DIRECTPLAY 8

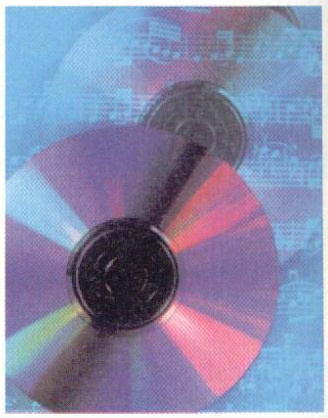
MUSICA EN PC
PROCESO DIGITAL DE SONIDO

LENGUAJES
JAVA PERFORMANCE

INTRODUCCIÓN A VISUALFOX PRO

PUNTEROS Y ARRAYS CON C - SOFTREFERENCES

Procesado digital de sonido en tiempo real



JOSÉ LUIS BLACO CLARACO.

Con el continuo aumento de velocidad en los ordenadores, ya se pueden realizar con sistemas digitales las tradicionales técnicas basadas en circuitos electrónicos analógicos. En este artículo vamos a ver algunas formas elementales para procesar sonido con un *PC*.

El objetivo final de este artículo es describir la creación un programa bajo *Windows* con *C++ Builder* que codifique y descodifique sonido digitalmente. Como paso intermedio, será necesario el uso de filtros digitales selectivos en frecuencia, por lo que primero se estudiarán éstos, y posteriormente se pasará al codificador y descodificador. El tipo de codificación que se usará es una muy sencillo, conocido como *scrambling*, porque lo que hace es invertir el espectro en un determinado rango. Este tipo de codificación se ha usado, por ejemplo, para codificar vídeo y sonido de algunos canales de televisión.

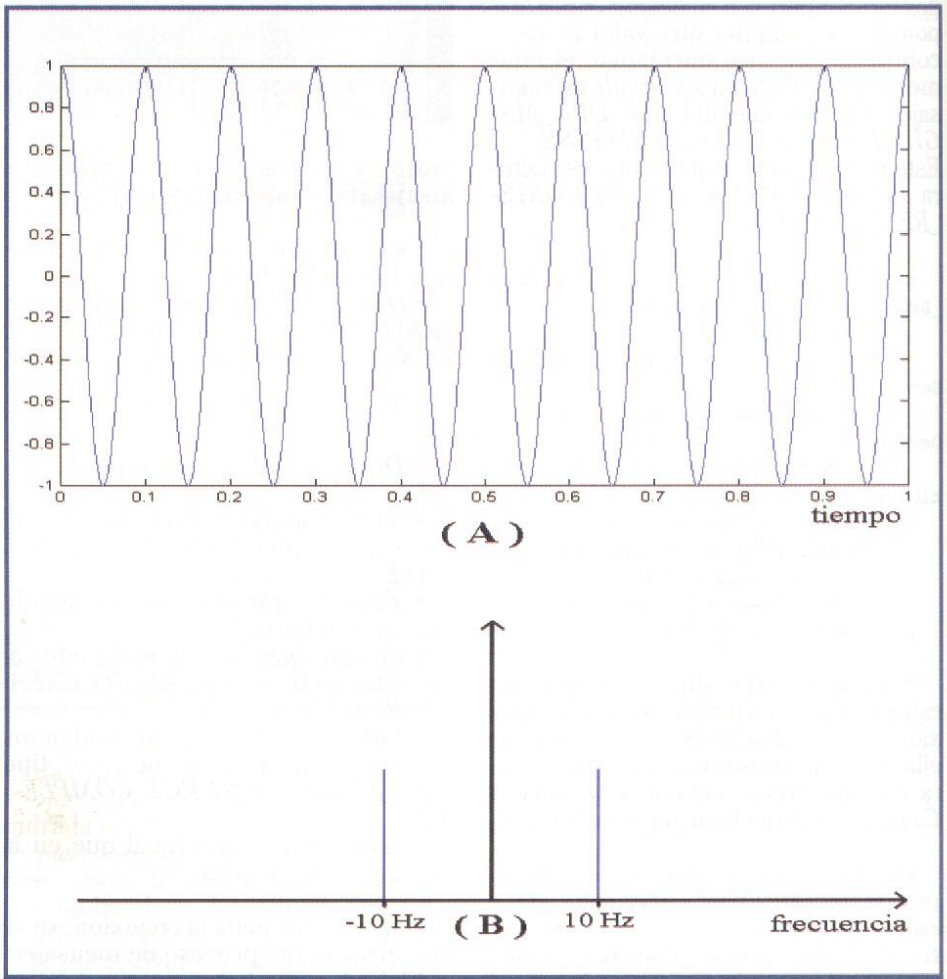
Aunque las operaciones matemáticas que se usarán en el programa son sencillas, vamos a ver antes el funcionamiento teórico de estos sistemas, sin entrar en los detalles matemáticos.

TEORÍA

Un sistema *DSP* sustituye un circuito electrónico analógico por el muestreo de la señal analógica, digitalizarla, procesarla y posteriormente, reconstruirla de nuevo a señal analógica. En nuestro caso, usaremos la tarjeta de sonido del *PC* para el muestreo y la reconstrucción de la señal. Ventajas importantes de este tipo de sistemas respecto al uso de circuitos analógicos, son su exactitud y su flexibilidad. Como veremos en el programa de ejemplo, con una misma función diseñada para filtrar una señal, se pueden crear multitud de filtros diferentes, cambiando sólo unos parámetros. El concepto más importante para entender todo el proceso, es el considerar a las señales en el dominio de la frecuencia.

ESPECTRO

Como se ve en la Figura 1, a partir de ahora no vamos a dibujar las señales respecto al tiempo, sino con respecto a las frecuencias (espectro). El uso de frecuencias negativas no es relevante, y sólo tiene valor matemático. Una frecuencia negativa equivale a la positiva correspondiente, por lo que a efectos prácticos, la parte izquierda es un reflejo de la parte derecha. En el gráfico, en el dominio de la frecuencia, lo que representamos con la altura del gráfico es



1 Un tono representado en el tiempo y en la frecuencia.

la "cantidad" de esa frecuencia que contiene la señal. Como se ve, un tono de una frecuencia sólo tiene componentes en el espectro en esa frecuencia.

Cerca del eje central están las bajas frecuencias, los *bajos*, y lejos del eje, las altas frecuencias, los *agudos*. La frecuencia máxima que se puede tener viene limitada por la frecuencia de muestreo de la tarjeta de sonido, siendo la mitad de esta. Si usamos una frecuencia de muestreo de 44100 *Hz*, la máxima frecuencia de sonido que se puede procesar es de 22050 *Hz*.

FILTROS DIGITALES

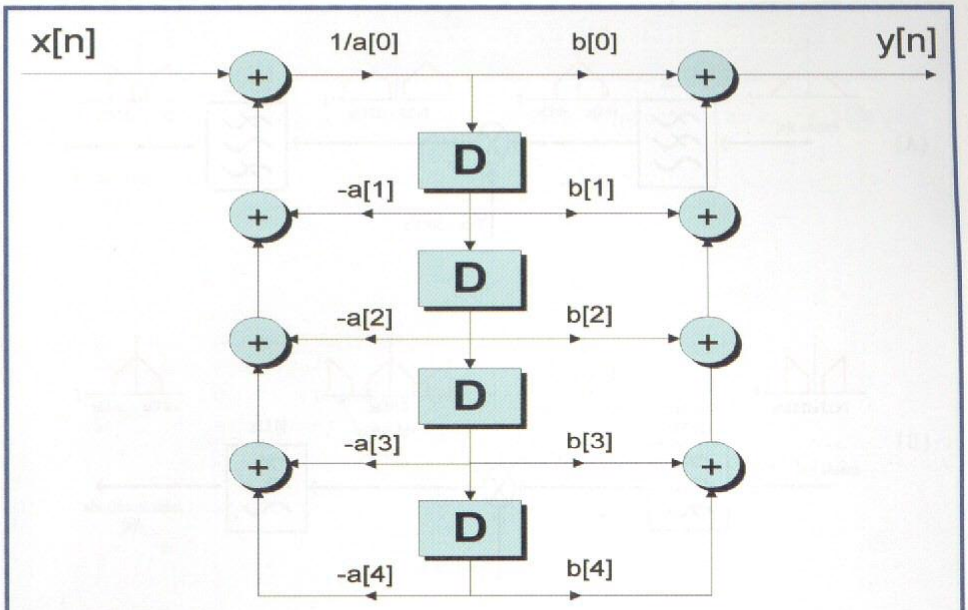
Un filtro digital se caracteriza por tener una determinada función de respuesta en frecuencia. Si al filtro se le introduce una señal de sonido, la salida tiene por espectro la multiplicación del espectro de la señal original por la respuesta del filtro. Por ejemplo, un ecualizador de un equipo de sonido tiene una respuesta en frecuencia con valores mayores en unas frecuencias que en otras, con lo que se atenúan algunas frecuencias y se realzan otras.

Los tipos básicos de filtros que vamos a usar son:

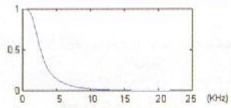
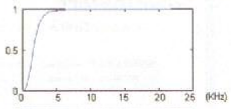
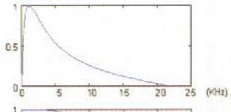
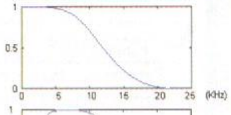
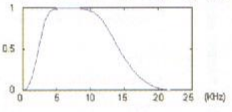
- *Paso bajo*: Sólo deja pasar las frecuencias menores de una frecuencia, llamada de corte.
- *Paso alto*: Sólo deja pasar las componentes de frecuencia mayor que la de corte.
- *Paso banda*: Solo deja pasar señales que se encuentren entre dos frecuencias dadas.

En la práctica, un filtro no deja pasar limpiamente una banda de frecuencias y bloquea completamente otras, sino que las atenúa en mayor o menor medida. Por eso, además de su tipo, un filtro se caracteriza por su orden. Cuanto mayor sea este, mejor realiza la separación de frecuencias, aunque órdenes demasiado altos pueden provocar distorsión y efectos indeseables.

Si queremos implementar un filtro digital, debemos procesar el *array* con las muestras de entrada *x[n]*, como se muestra en la Figura 2, siendo *a[k]* y *b[k]* precisamente las constantes que definen el comportamiento del filtro (tipo, frecuencia de corte, orden). Los bloques *D* son retardadores, es decir que si a su entrada tenemos el valor *x[i]*, a la salida tenemos *x[i-1]*. Además, los valores sobre las flechas indican una multiplicación por dicho valor. El número de retardadores necesarios depende del orden del filtro y su tipo. Por ejemplo, un filtro de paso de banda necesita más que otro de paso bajo. Como se puede ver, la salida sólo depende de los valores anteriores de la señal, requisito indispensable si queremos que nuestro sistema trabaje en tiempo real.



2 Esquema general de un filtro digital.

Indice	Tipo de filtro	Frecuencia(s) de corte	Respuesta en frecuencia	a[k]	b[k]
0	Paso bajo 2º orden	2 KHz		1 -1.601 0.668	0.817 -1.635 0.817
1	Paso alto 2º orden	2 Khz		1 -1.601 0.668	0.817 -1.635 0.817
2	Paso banda 1º orden	300-3400 Hz		1 -1.616 0.633	0.183 0 -0.183
3	Paso bajo 2º orden	10 Khz		1 -1.171 0.177	0.251 0.503 0.251
4	Paso banda 2º orden	2800-12800 Hz		1 -1.272 0.6341 -0.307 0.177	0.251 0 -0.503 0 0.2514

3 Los filtros implementados en el programa.

Se demuestra que el diagrama de bloques de la figura equivale a la siguiente ecuación:

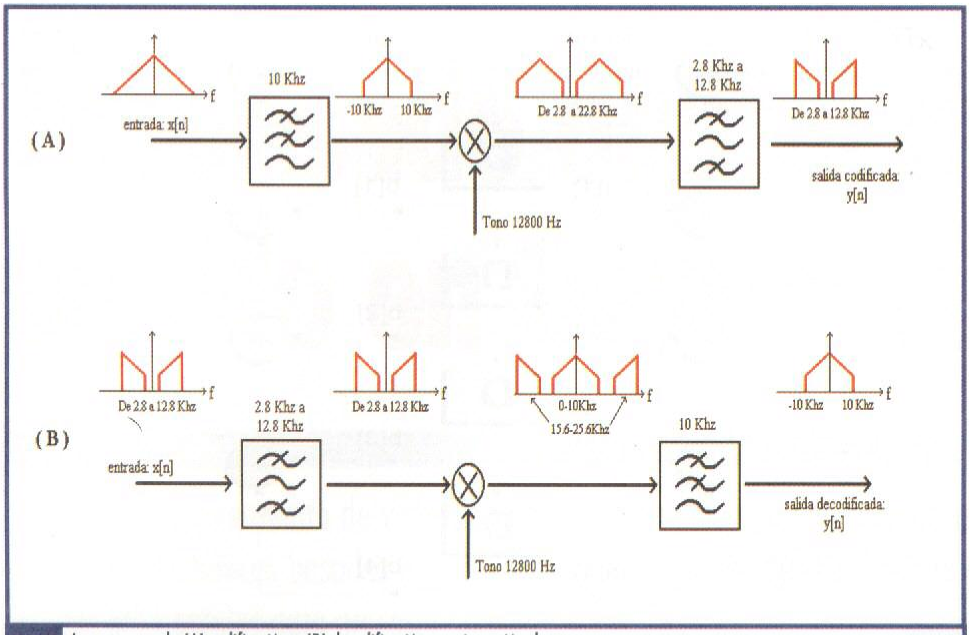
```
y[n]=(   b[0]*x[n]+b[1]*x[n-1]+...-
a[1]*y[n-1]-a[2]*y[n-2]-... )/a[0]
```

Para implementar un filtro, hay que usar un bucle que calcule cada *y[n]* según esta ecuación, como se ve en el Listado 1.

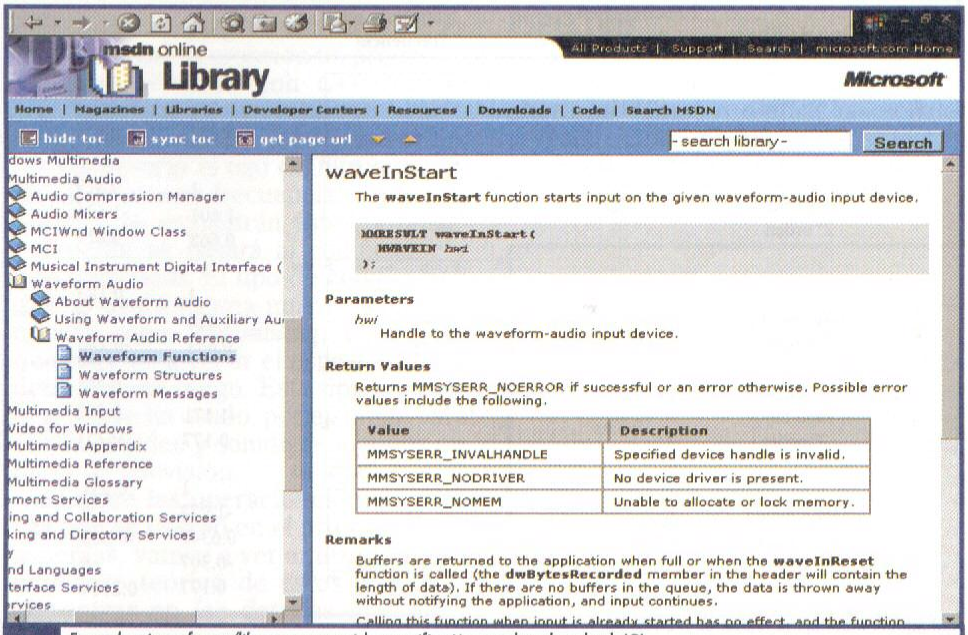
En nuestro programa necesitaremos dos filtros para la codificación o descodificación, como se verá más abajo. Además, se han añadido otros tres para experimentar con ellos. En la Figura 3 se ven sus respuesta en frecuencia junto con los valores de las constantes *a[k]* y *b[k]* correspondientes. Los filtros usados son de *Butterworth*, que tienen una respuesta en frecuencia suave, como se ve en la figura. El cálculo de las constantes *a[k]* y *b[k]* es bastante complicado, aunque actualmente hay paquetes matemáticos que permiten calcular estos valores fácilmente.

CODIFICADOR/DECODIFICADOR

Para codificar una señal de entrada, primero se pasa por un filtro de paso bajo de 10 *Khz*, para limitarla en frecuencia, luego se multiplica por un tono de 12.8 *Khz*, con lo que su espectro se desplaza, para finalmente con otro filtro de paso de banda, quedarnos sólo con las mitades inferiores de estos espectros. Como se ve en el dibujo del espectro de la señal de salida, lo que antes estaba en 0 *Hz* ahora está en 12.8 *Khz*, y lo que estaba en 10 *Khz* ahora



4 Los procesos de (A) codificación y (B) decodificación, por inversión de espectro.



5 En *msdn.microsoft.com/library* encontrará la especificación completa de todo el *API*.

está en 2.8 *Khz*. Así, las frecuencias bajas y las altas se han intercambiado, por lo que el sonido resultante es ininteligible. Pero se puede recuperar siguiendo el proceso de la figura: Un filtro de paso de banda desde 2.8 *Khz* hasta 12.8 *Khz*, para asegurarnos de que no entra al sistema el ruido fuera de ese intervalo (siempre presente), tras lo cual, se multiplica por el mismo tono que antes, consiguiendo que el espectro de la señal vuelva a su lugar original, pero produciendo como se ve en la figura, unos duplicados no deseados, que se filtran con un filtro de paso bajo de 10 *Khz*. Como se ve, el sistema es similar a la modulación de radio de banda lateral única, pero en bajas frecuencias.

En la figura se puede ver que el espectro de la señal descodificada es igual al de entrada al codificador, excepto que se le han quitado las componentes superiores a 10 *KHz*. Sin embargo, esto no es importante para señales de voz o sonido, ya que típicamente la mayor parte de la señal, recae por debajo de dicha frecuencia.

La implementación en *C++* se ve en el Listado 2. Sigue exactamente lo expuesto en la Figura 4. Para generar el tono a 12.8 *KHz* se usa la función coseno de frecuencia angular $W=2 \cdot p \cdot 12800/44100$, que es la frecuencia digital de ese tono, y como se puede ver, depende de la frecuencia de muestreo que usemos con la tarjeta de sonido.

API DE SONIDO

El *API* de sonido de *Windows* nos permite aislarnos de la tarjeta de sonido, ofreciendo una interfaz estándar independiente del *hardware* instalado.

Existen dos bloques casi idénticos en estas funciones, uno para grabación, y otro para reproducción de sonido. Para el uso de un canal de grabación o de reproducción, hay que seguir unos pasos, que se detallan a continuación.

1) Abrir el canal con *waveInOpen* o *waveOutOpen*, especificando el formato de las muestras, frecuencia de muestreo, número de bits por muestra,... además del mecanismo de *callback* que se quiere usar. Casi todos los parámetros se pasan en una estructura del tipo *WAVEFORMATEX*, cuyos miembros se muestran en la Tabla 1. En nuestro ejemplo usaremos el método de *callback* de ventana, por lo que se enviarán mensajes a la ventana del programa indicando diversos eventos, como se verá más abajo. El formato de la grabación será el más sencillo: *PCM* (*Pulse code modulation*), en el que cada muestra se codificará como una palabra binaria indicando su valor, lo que nos permite trabajar directamente con los valores de las muestras. Hay más formatos, como, por ejemplo, el *Adaptive Delta Pulse Code Modulation* (*ADPCM*). Al abrir un canal obtenemos un *handle HWAVEIN* o *HWAVEOUT* que nos servirá para llamar a las demás funciones.

TABLA 1. Miembros de la estructura WAVEFORMATEX

Miembro	Descripción
WORD wFormatTag	Tipo de formato de compresión. El más sencillo de usar es WAVE_FORMAT_PCM.
WORD nChannels	1 si es mono, y 2 si es estéreo.
DWORD nSamplesPerSec	Muestras por segundo.
WORD nBlockAlign	Número de bytes por muestra, típicamente 1 o 2.
DWORD nAvgBytesPerSec	Debe ser el producto de nSamplesPerSec y nBlockAlign.
WORD wBitsPerSample	Según el valor de nBlockAlign sea 1 o 2, debe valer 8 o 16.
WORD cbSize	Información extra. Cero para PCM.

2) Preparar las cabeceras. Las cabeceras (*WAVEHDR*) albergan cada una un puntero a un *buffer* puesto por el programador, para almacenar los datos a grabar o conteniendo los datos a reproducir, según se usen con *waveInPrepareHeader* o con *waveOutPrepareHeader*, respectivamente. Estos *buffers* son los fragmentos más pequeños que podemos grabar o reproducir, y su tamaño depende de las necesidades de cada aplicación.

3) Añadir las cabeceras. Si lo que queremos es grabar, debemos añadir la cabecera preparada con su correspondiente *buffer* con *waveInAddBuffer*. *Windows* almacenará en una cola todos los *buffers*, y los devolverá a la aplicación, según se vayan llenando, empezando por el más antiguo. Para que comience la grabación, hay que llamar a la función *waveInStart* y para terminar de grabar, se llama a *waveInStop*. Si lo que se quiere es reproducir, se usa *waveOutWrite*, e igualmente, *Windows* los irá guardando en una cola de reproducción. A diferencia del caso de la grabación, ahora la reproducción comienza de inmediato al añadir un *buffer*.

4) Liberar cabeceras. Cuando no se vayan a usar más, hay que liberar las cabeceras con *waveInUnprepareBuffer o waveOutUnprepareBuffer*. Es importante señalar que esta función no libera la memoria apuntada por los punteros a los *buffers*, por lo que hay que liberarla explícitamente si se trata de memoria dinámica.

5) Cerrar los canales. Con las funciones *waveInClose* y *waveOutClose* indicamos que no vamos a realizar más operaciones sobre dichos canales, y los *handles* que obtuvimos ya no son válidos. Si se quiere volver a realizar una operación de grabación o reproducción hay que volver a abrir los canales.

En el Listado 3 se muestra la forma de abrir el canal de grabación, preparar los *buffers* y añadirlos a la cola. Debido a que en cada llamada a cada función del *API* puede ocurrir un error, puede ser de utilidad crear una función como *procErr* para evitar repetidos bloques *if* comprobando errores.

MENSAJES

Como se ha dicho más arriba, uno de los métodos que dispone el *API* para informar de eventos es mediante mensajes enviados a la ventana. Usando el evento *OnMessage* del componente *ApplicationEvents* en *C++ Builder* podemos procesar fácilmente estos mensajes, como se ve en el Listado 4. *Windows* envía mensajes para avisar de la apertura y cierre de un canal, así como de la terminación de un *buffer*, ya sea de grabación o reproducción.

LISTADO 1. Implementación del filtro digital

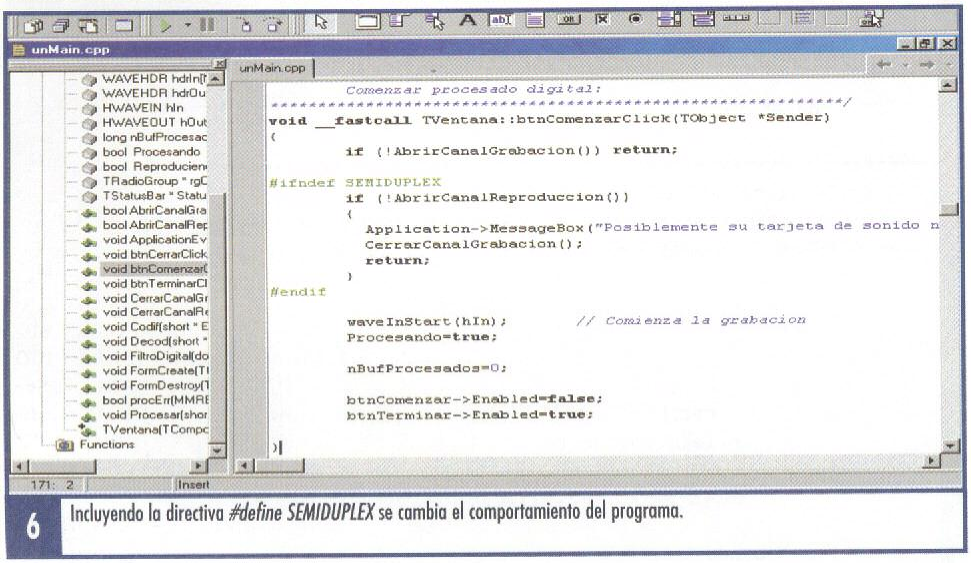
```
void TVentana::FiltroDigital(double *a,double *b,short *x,short *y,long lon)
{
    int      n,k;
    double   v;

    for (n=0;n<lon;n++)
    {
         v=0.0;
        for (k=0;k<=N_RETARDADORES;k++)
if (n>=k) v+= b[k]*x[n-k];
        for (k=1;k<=N_RETARDADORES;k++)
if (n>=k) v-= a[k]*y[n-k];

        v=v/a[0];
        y[n]=(short)v;

    }  //n
  }
```

LISTADO 2. Las funciones de decodificación y codificación

```
/*************************************************************
     Decodifica por inversion de espectro
*************************************************************/
void TVentana::Decod(short *Ent,short *Sal,long lon)
{
     int     n;
     double  W= PI_2*12800.0/FREQ_MUESTREO;
     short   *tmp=new short[TAM_BUF];

     // F. Paso banda
     FiltroDigital(FILTRO_As[4],FILTRO_Bs[4],Ent,tmp,lon);
     // Multiplicar por un tono:
     for (n=0;n<lon;n++)  {
           tmp[n]=(short) ( ((double)tmp[n])*cos(n*W) );
      }
     // F.paso bajo
     FiltroDigital(FILTRO_As[3],FILTRO_Bs[3],tmp,Sal,lon);

      delete[] tmp;
}
/*************************************************************
     Codifica por inversion de espectro
*************************************************************/
void TVentana::Codif(short *Ent,short *Sal,long lon)
{
    double  W= PI_2*12800.0/FREQ_MUESTREO;
     short   *tmp=new short[TAM_BUF];

     // F.paso bajo
     FiltroDigital(FILTRO_As[3],FILTRO_Bs[3],Ent,tmp,lon);
     // Multiplicar por un tono:
     for (int n=0;n<lon;n++) {
           tmp[n]=(short) ( ((double)tmp[n])*cos(n*W) );
      }
// F. paso banda
     FiltroDigital(FILTRO_As[4],FILTRO_Bs[4],tmp,Sal,lon);

      delete[] tmp;
}
```

LISTADO 3. Abrir el canal de grabación

```
bool TVentana::AbrirCanalGrabacion()
{
      WAVEFORMATEX   wf;
      int                  i;

      wf.wFormatTag= WAVE_FORMAT_PCM;
      wf.nChannels= 1;
      wf.nSamplesPerSec= FREQ_MUESTREO;
      wf.nBlockAlign= 2;
      wf.nAvgBytesPerSec= wf.nBlockAlign*wf.nSamplesPerSec;
      wf.wBitsPerSample= 16;
      wf.cbSize=0;

      // Abrir canal
      if ( procErr( waveInOpen(&hIn, WAVE_MAPPER, &wf,(DWORD)Handle,NULL,CALLBACK_WINDOW) ) )
            return false;

      // Preparar los buffers y añadirlos a la cola
      for (i=0;i<N_BUF;i++)
      {
            hdrIn[i].lpData=(char*)bufIn[i];
            hdrIn[i].dwBufferLength=sizeof(short)*TAM_BUF;
            hdrIn[i].dwFlags=0;
            hdrIn[i].dwUser=i;

            if ( procErr ( waveInPrepareHeader (hIn,&hdrIn[i],sizeof(WAVEHDR) ) ) )
            {
                  CerrarCanalGrabacion();
                  return false;
            }

            if ( procErr( waveInAddBuffer( hIn,&hdrIn[i],sizeof(WAVEHDR) ) ) )
            {
                  CerrarCanalGrabacion();
                  return false;
            }
      }
      return true;
}
```



6 Incluyendo la directiva *#define SEMIDUPLEX* se cambia el comportamiento del programa.

Otro método que permite el *API* para avisar de eventos es el uso de una función como método de *callback*, en lugar de mensajes a una ventana. En este caso el proceso es idéntico, aunque nuestra función no es llamada por la ventana al recibir mensajes, sino directamente por el *API* de sonido.

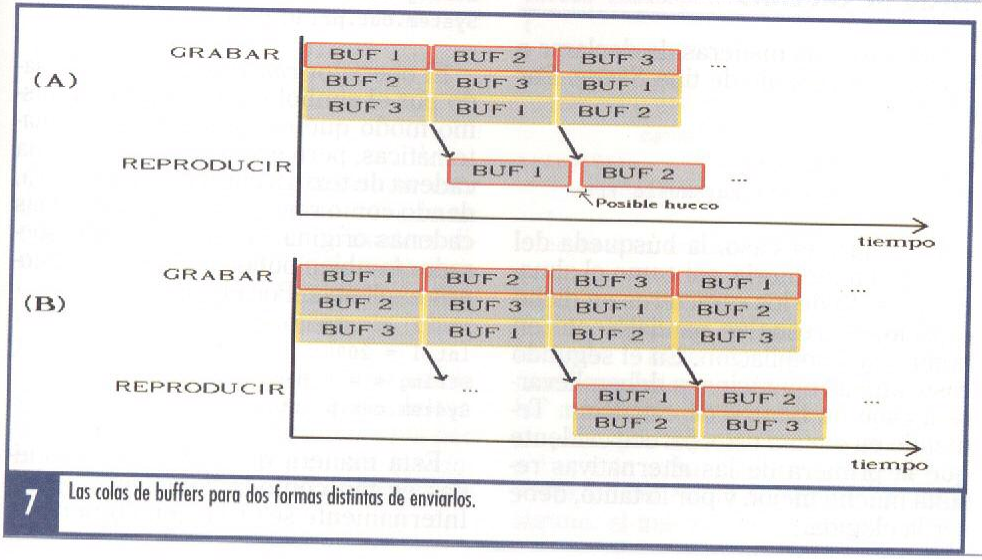
PROCESADO

Para conseguir un efecto de grabación y reproducción continuado, debemos mantener en todo momento las colas de entrada y salida con varios *buffers*, de forma que cuando se termina la grabación en uno de ellos, se continua grabando en el siguiente, se procesan los datos del *buffer* lleno para crear un *buffer* de salida que se añade a la cola de salida, mientras que reutilizamos el *buffer* de entrada añadiéndolo una vez más a la cola de entrada. De esta forma, conseguimos un efecto de procesado en tiempo real, con un retardo proporcional al tamaño de los *buffers*, por lo que no conviene que los *buffers* sean excesivamente grandes.

Ahora bien, si partimos de una situación inicial con la cola de salida vacía (el caso del comienzo del programa), tenemos el problema de que al durar exactamente igual tiempo los *buffers* de salida que los de grabación, debería tardarse exactamente igual en recibir, procesar, y enviar cada *buffer*, para que no queden posibles momentos de silencio entre cada

LISTADO 4. Procesado de mensajes de sistemas de sonido (para la versión en tiempo real

```
void __fastcall TVentana::ApplicationEventsMessage(tagMSG &Msg,
      bool &Handled)
{
WAVEHDR      *hdrRecibido; // La cabecera devueltaint            Index;    // Y su numero
switch (Msg.message){
case MM_WOM_OPEN:  // Se abre canal de salida    Reproduciendo=true;      cbReproduciendo->Checked=true;
          break;

     case MM_WOM_CLOSE: // Se cierra canal de salida
          Reproduciendo=false;
          cbReproduciendo->Checked=false;
          break;

     case MM_WIM_OPEN: // Se abre canal de entrada
          Grabando=true;
          cbGrabando->Checked=true;
          break;

     case MM_WIM_CLOSE: // Se cierra canal de entrada
          Grabando=false;
          cbGrabando->Checked=false;
          break;


     case MM_WIM_DATA: // Se ha terminado de grabar un buffer:
      hdrRecibido=(WAVEHDR*)(Msg.lParam);  Index=hdrRecibido->dwUser;
      if (! Procesando)       // Se ha cortado la grabacion       break;       // Procesar el fragmento re-
cibido
      Procesar((short*)hdrRecibido->lpData,(short*)hdrOut[Index].lpData,TAM_BUF );

 if (nBufProcesados==1)
       // Si es el 2º, sacar el 1º tambien...
       procErr( waveOutWrite(hOut,&hdrOut[Index-1],sizeof(WAVEHDR)));

      if (0!=nBufProcesados)
       procErr( waveOutWrite(hOut,&hdrOut[Index],sizeof(WAVEHDR) ) );

      nBufProcesados++;

 // Volver a añadir el bufer de grabacion: procErr(waveInAddBuffer( hIn,hdrRecibido,sizeof(WAVEHDR) ));
          break;

     case MM_WOM_DONE: // Buffer terminado de reproducir
          break;
     };      // fin del switch

}
```



7 Las colas de buffers para dos formas distintas de enviarlos.

buffer de salida. Para evitar este efecto indeseable, el primer *buffer* de salida no se envía hasta que no está listo el segundo, como se ve en el Listado 4, con lo que se da un margen de seguridad que asegura un buen comportamiento, ya que siempre habrá dos *buffers* en la cola de salida.Para aquellos cuyas tarjetas de sonido no soporten reproducción y grabación simultánea, se ha añadido la posibilidad de compilar el programa con la directiva *#define SEMIDUPLEX*, con lo cual el programa graba muestras de unos pocos segundos, las procesa, las reproduce y vuelve a empezar, permaneciendo en silencio mientras graba.