Clipper por dentro (I)

Por Manu Roibal

Al igual que números anteriores, intentaré hablar sobre temas de interés para programadores Clipper. Espero, por tanto, que el contenido de este artículo pueda llevarse directamente, y sin muchos cambios, a las aplicaciones que habitualmente se desarrollan.

Durante estas líneas intentaremos explicar algunas funciones indocumentadas nuevas y luego hablaremos única y exclusivamente de los problemas que aquejan al programador Clipper, a saber, problemas de impresión, de cooperación en multitarea, métodos de protección contra el pirateo, etc.

Antes de eso, hablaremos de algunas funciones no documentadas que nos serán útiles posteriormente.

Algunas indocumentadas como apoyo a la programación

Comenzaremos explicando rápidamente algunas funciones que nos serán necesarias para la programación de los ejemplos. Todas estas funciones son no documentadas, y por lo tanto su uso es, cuando menos, de no recomendación oficial, aunque sí legal como bien conoce y detalla mi buen amigo **Marino Posadas**. El argumenta, con la razón de su parte y quiero entender que también la ley, que cuando se compra un producto, se compra una licencia de uso del mismo en su totalidad, de lo documentado y de lo no documentado. Durante mucho tiempo grandes fabricantes de software no han documentado ciertos aspectos del mismo para aprovecharse de esa información privilegiada en productos colaterales al principal, que gracias a esa información funcionaban más rápido o mejor o ambos en muchos casos. Esa práctica ha sido la tónica general de Clipper y las *Tools* durante mucho tiempo. Finalmente parece que CA documenta más interioridades cada día. Brindemos por ello.

```
/* --- Fuente 1 ------------------------------------------------------ */
/* Copia nBytes desde pFuente a pDestino */
void _bcopy( LPBYTE pDestino, LPBYTE pFuente, WORD nBytes );

/* Mueve nBytes desde pFuente a pDestino */
void _bmove( LPBYTE pDestino, LPBYTE pFuente, WORD nBytes );

/* Compara nBytes desde pFuente y pDestino */
WORD _bcmp( LPBYTE pDestino, LPBYTE pFuente, WORD nBytes );

/* Inicializa nLen bytes con el valor nValor comenzando en pInicio */
void _bset( LPBYTE pInicio, WORD nValor, WORD nLen );
```

Las funciones del fuente 1 realizan rápidos movimientos y comparaciones de memoria mediante funciones de cadenas *REP MOVS*, *REP CMP*... de los procesadores *Intel*. El único problema es que están pensadas para procesadores 8086 y siempre realizan los movimientos a nivel de *byte,*

aun cuando el 80386 y superiores están ya preparados para hacer movimientos a nivel de cuádruple byte, es decir a nivel de *long*, lo cual es, obviamente, mucho más rápido. No estaría de más que el compilador nos dejase elegir el tipo de procesador en el que va a funcionar nuestro programa y actuase en consecuencia generando código para uno y/o para otro. La compatibilidad seguiría estando garantizada siempre que se generase código para el peor de los procesadores, pero tendríamos la opción de optimizar muchas cosas con las nuevas prestaciones de los microprocesadores. Hay que tener en cuenta que estamos usando un lenguaje de programación que genera código para un procesador, el 8086, con más de 15 años, y eso en informática son varios siglos.

Ahora explicaremos algunas funciones más del sistema de gestión de eventos que nos serán muy útiles en los ejemplos posteriores.

Lo primero que hay que definir es la estructura de eventos. Esta consta de un entero, que no parece contener información de relevancia (tal vez sea reservada), y otro entero que contiene el número del evento generado. Esta es la información que se le pasa a los gestores de eventos para que actúen en consecuencia, analizando el evento que se ha producido.

```
/* --- Fuente 2 ------------------------------------------------------ */
typedef struct
{
   WORD nReserved,
   WORD nMessage,
} EVENTO;

typedef EVENTO * PEVENTO;

typedef void ( far * FUNCION_EVENTO) ( PEVENTO pEvent );


/* Registra un Event Handler nuevo en la tabla de gestores del sistema
   y entra a formar parte del proceso de gestión automáticamente*/

WORD _evRegReceiverFunc(FUNCTION_EVENTO pFunc,WORD nMensaje );


/* Elimina el gestor de eventos de la tabla del sistema. El valor que
  recibe es el que devuelve la función _evRegReceiverFunc() */

void _evDeregReceiver( WORD wAntiguoGestor );


/* Envía un evento. Esta función debe mirarse desde el ángulo contrario
   a las anteriores. Esta genera eventos, las anteriores los recibían y
   procesaban.
   A esta función hay que incluirle el mensaje a envía y a quien enviárselo.
   nQuien indicará a quien enviar el mensaje. Lo enviará a todos los gestores
   cuyo código de gestor sea menor que el indicado. Obviamente 0xFFFF hará
   que se le envíe a todos los gestores. */

void _evSendId( WORD wMessage, WORD wHandlerOrder );
```

Ahora explicaremos algunas funciones que podemos catalogar como misceláneas, pero que siempre viene bien conocer para no tener que trabajar reinventando la rueda. Las codificaremos en C.

Pitido

La primera función es *_beep_().* Su uso parece claro, generar un tono por el *speaker*, con una frecuencia y duración determinadas (es el típico pitido del ordenador). Su prototipo es:

```
void _beep_( void );
```

Errores internos

¿Desea que sus aplicaciones salgan al sistema operativo generando un error interno similar a los de Clipper? Eso puede ser muy útil en sistemas de protección para que el usuario no encuentre el punto de salida del programa:

```
void _ierror( WORD nError );
```

Esta función genera un error y sale al sistema escribiendo el mensaje, ya de todos conocido, desgraciadamente, *error interno número 999* (en caso de haberse asignado a *nError* un valor de 999).

Argumentos y parámetros

Las dos funciones siguientes nos dan información sobre el número de parámetros que ha recibido el programa y sobre los propios parámetros. *_argc* nos indica cuántos parámetros, incluyendo el propio nombre del programa, hay en la línea de comandos del programa. *_argv* es un *array* de cadenas de caracteres, de tantos elementos como indica *_argc*. Estos elementos corresponden a cada uno de los parámetros.

El primer elemento del *array* es el nombre completo del programa en ejecución. Aunque no exista un método que impida renombrar nuestro programa, al menos sabremos cuándo lo hacen. La sintaxis y significado de estas dos variables es idéntica a la de C. De hecho, son variables públicas, no olvidemos que Clipper está hecho en C.

```
WORD     __argc;
LPBYTE * __argv;
```

Fecha y hora del sistema

Las dos funciones siguientes nos dan la fecha y la hora del sistema:

```
void _ostime( LPWORD nHora,
              LPWORD nMinuto,
              LPWORD nSegundo,
              LPWORD nDecimas
            );

void _osdate( LPWORD nDia,
              LPWORD nMes,
              LPWORD nAño,
              LPWORD
              nDiaSemana
            );
```

Estas funciones son muy útiles para calcular tiempos desde C o, incluso, para programar sistemas de protección que funcionen solamente durante cierto número de días, como veremos más adelante.

El *API* de memoria

Vayamos ahora con el API de memoria. Al igual que ocurre con el API de ficheros, el API de memoria está aquí desde Clipper 5.0, pero de forma no documentada.

```
/* Clipper 5.0 & 5.1 memory API */

HANDLE  _vAlloc( USHORT size,
                 USHORT flags
               );
void    _vFree (HANDLE h);

FARP    _vLock  (HANDLE h);
FARP    _vWire  (HANDLE h);
void    _vUnLock(HANDLE h);
void    _vUnWire(HANDLE h);
```

Este es el nuevo API de Clipper 5.2 y 5.3 que, como se ve, es exactamente igual y, tan solo cambia una *x* en la denominación de las funciones. El significado de las funciones es exactamente igual.

```
/* Clipper 5.2 & 5.3 memory API */

HANDLE  _xvalloc(  USHORT size,
                   USHORT flags
                 );
void    _xvfree (HANDLE h);

FARP    _xvlock  (HANDLE h);
FARP    _xvwire  (HANDLE h);
void    _xvunlock(HANDLE h);
void    _xvunwire(HANDLE h);
```

_valloc() y *_vfree()* sirven para reservar y liberar memoria, respectivamente. Después de reservarla, hay que bloquearla para su uso. Eso se hace con *_vlock()* o con *_vwire()*. Tras usarla, y antes de liberarla, hay que desbloquearla mediante *_vunlock()* o *_vunwire()*.

Para explicar la diferencia entre el método de bloqueo y desbloqueo basado en *lock()* y el basado en *wire(),* explicaremos algunos conceptos de memoria virtual.

Un sistema de memoria virtual trabaja con una memoria superior a la realmente instalada en el sistema. Esta memoria está en lo que se llama *Plano V*. La memoria real está en el *Plano R*. Cuando se hace una reserva de memoria, el *VMM* de Clipper reserva la cantidad solicitada en el *Plano V*. Pero esta memoria no puede usarse todavía, dado que sólo son utilizables los datos que están en el *Plano R*. Hay que pensar que el *Plano V* puede estar soportado por un *EPS* o fichero de disco.

Es realmente al bloquear la memoria cuando el *VMM* nos busca un hueco en *el Plano R* y mueve nuestros datos al mismo para que podamos usarlo. Obviamente, al finalizar de usar esos datos debemos desbloquearlos para que el *VMM* se los pueda volver a llevar al *Plano V* y dejar el espacio del *Plano R* libre para otros datos. Hay que tener en cuenta que el *Plano R* es normalmente muy reducido.

Pues bien, la diferencia entre *lock()* y *wire()* es que el primero bloquea la primera porción del *Plano R* con tamaño suficiente para albergar los datos, mientras que *wire()* busca la porción de tamaño suficiente que más abajo esté en la memoria. Se debe usar *wire()* cuando los datos vayan a permanecer mucho tiempo en memoria bloqueados, ya que al estar lo más abajo posible del *Plano R* no molestan en exceso al *VMM* en su algoritmo de sustitución de páginas. Por contra, *wire()* es más lento que *lock(),* ya que, además de buscar un hueco suficiente, busca el mejor hueco suficiente, y por lo tanto se desaconseja su uso para datos que se emplean durante un corto periodo de tiempo.

Un ejemplo de uso de memoria sería:

```
/* --- Fuente 3 ---------------------------------------------------- */
CLIPPER Myfunc()
    {
    HANDLE hHandle;
    LPBYTE  cBuffer;

    hHandle = _valloc( 4096, 0x0 );     // reservo
    cBuffer  = _vlock( nHandle );       // bloqueo

    _bset( cBuffer, 32, 4096 );         // uso
    _retclen( cBuffer, 4096 );

    _vunlock( hHandle );                // desbloqueo
    _vfree( hHandle );                  // libero
    }
```

Lo siguiente que debemos tener es un potente API de ficheros, compatible con todas las versiones de Clipper, y no como ocurre con el que CA ha documentado, que sólo funciona en Clipper 5.2. Veamos el fuente 4.

```
/* --- Fuente 4 ---------------------------------------------------- */
typedef USHORT FHANDLE;
typedef FHANDLE fhandle;
typedef FHANDLE far * FHANDLEP;

WORD     _terror, _horror;

FHANDLE  _tcreat( LPSTR cFileName, WORD wMode );
FHANDLE  _topen( LPSTR cFileName, WORD wMode );
WORD     _tclose( FHANDLE wHandle );
DWORD    _tlseek( FHANDLE wHandle, LONG lRecNo, WORD wPosition );
WORD     _tread( FHANDLE wHandle, LPBYTE cBuffer, WORD wBytes );
WORD     _twrite( FHANDLE wHandle, LPBYTE cBuffer, WORD wBytes );
WORD     _tunlink( LPBYTE cFile );
FHANDLE  _tctemp( LPBYTE cPath, WORD nMode, LPBYTE cFile );
void     _tcommit( void );
```

Las variables públicas *_terror* y *_horror* son actualizadas por todas las funciones de manejo de ficheros para indicar errores. Estos valores son los que devuelve la función *fError()* de Clipper. Como se ve, Clipper dispone de un potente API de ficheros, no documentado de la versión 5.0 que nos permite crear, abrir, cerrar, leer, escribir...

Mención especial merecen algunas funciones, como, por ejemplo *_tctemp,* que nos permite crear auténticos ficheros temporales. Se trata de ficheros con nombres aleatorios creados por el propio sistema operativo, por lo que la posibilidad de repetición es muy baja. A esta función *_tctemp()* debemos indicarle el *path* donde crear el fichero y el modo de apertura del mismo, y ella nos devuelve el nombre en *cFile* y un manejador o *handle* al fichero abierto.

Estas cuatro funciones, sin embargo, sólo están definidas en Clipper 5.2. Su función parece bastante clara, a partir de sus nombres permite crear, borrar, acceder y devolver el directorio en curso:

```
WORD   _tchdir( LPBYTE cPath );
WORD   _trmdir( LPBYTE cPath );
WORD   _tmkdir( LPBYTE cPath );
LPBYTE _tcurdir( WORD nDrive );
```

Otras dos importantes funciones son *_f_first()* y *_f_next()* que permiten buscar la primera ocurrencia de un fichero y las siguientes. Estas funciones tienen una sintaxis idéntica a sus homónimas en C:

```
struct blk
   {
   BYTE bDummy[ 21 ];
   BYTE bAttributes;
   WORD wTime;
   WORD wDate;
   DWORD dwSize;
   BYTE Name[ 13 ];
   };

int _f_first( LPBYTE,
              struct blk *,
              WORD
            );
int _f_next( struct blk * );
```

La función *_dsspace* nos permite calcular el tamaño del disco y el tamaño de la zona libre del mismo.

```
typedef struct
   {
   WORD FreeClusters;
   WORD SectorsByClusters;
   WORD Clusters;
   WORD BytesBySector;
   } dspace;

int _dspace( WORD, dspace * );
```

Multiplicando *FreeClusters * SectorsByClusters * BytesPerSector* obtendríamos la zona libre del disco, medida en bytes. De igual forma, multiplicando *Clusters * SectorByClusters * BytesBySector* obtendríamos el tamaño total del disco.

Al multiplicar estos valores, hay que tener en cuenta que se trata de enteros y, lógicamente, la multiplicación de 2 enteros puede dar lugar a un doble entero o *long*. En este caso, el problema se agrava, ya que tenemos que multiplicar, no 2, sino 3 enteros, con lo cual, muy probablemente (a no ser que los número sean muy bajos), dará lugar a un *long*. Esto haría que nos viésemos forzados a codificar las funciones en Microsoft C.