Por Manu Roibal

Cómo usar coma flotante y *long* desde *Borland C*

Personalmente, prefiero con mucho el entorno de desarrollo de *Borland*. Muchas veces oigo cosas como que no pueden hacerse funciones *C* para Clipper que manejen coma flotante u operaciones de *longs* con *Borland C*.

Esto, que en principio parece cierto, no lo es. Como ejemplo, *FAST.LIB* usa operaciones con datos de todo tipo en funciones creadas con *Borland C*.

¿Cómo se consigue esto? Hagamos un pequeño inciso para explicarlo.

Borland C, al igual que la mayoría de los lenguajes de programación, realiza las operaciones de datos básicos, como *char* e *int,* con código *inline*. Esto es, las operaciones con datos de estos tipos se realizan con código allí donde es necesario. Sin embargo, las operaciones de coma flotante *longs*, y algunas otras, las hace llamando a funciones de librería, ya que sería muy pesado repetir el extenso código de dichas funciones en cada punto del programa que lo necesite. Obviamente, cada fabricante de software tiene sus propios algoritmos basados en análisis numérico para crear las operaciones matemáticas complejas, y eso se traduce en nombres de funciones diferentes. Clipper está hecho en *MSC,* y por lo tanto usa su sistema de funciones matemáticas. Borland C tiene otro muy diferente, y por lo tanto, si creamos una función en Borland C con operaciones de *long, double o float*, estas funciones no se encontrarán en el momento de enlazar el programa.

El problema parece evidente, no se puede hacer ninguna función en Borland C que use *double, float* o *long*. Esto es realmente un calvario, ya que el *C* de *Microsoft* es bastante pobre en muchos aspectos. La solución es crear un módulo que implemente nuestras propias rutinas matemáticas en *MSC* y luego usarlo desde Borland C. Esto, que es de hecho una nimiedad, es muy potente y versátil.

En el fuente 1 se muestra el API de funciones matemáticas que usa *FAST.LIB*. Lógicamente, no lo doy entero; no por conservar la privacidad del mismo, sino porque es muy largo, y con ver la idea es suficiente.

```
// --- Fuente 1 ---------------------------------------------------------

// rota derecha dos longs sin signo y devuelve el resultado
DWORD _fastuShr( DWORD nNumber, int nRota )
  {
  return( nNumber >> nRota );
  }

// rota izquierda dos longs sin signo y devuelve el resultado
DWORD _fastuShl( DWORD nNumber, int nRota )
  {
  return( nNumber << nRota );
  }

// rota derecha dos longs con signo y devuelve el resultado
long _fastShr( long nNumber, int nRota )
  {
  return( nNumber >> nRota );
  }
```

```
// rota izquierda dos longs sin signo y devuelve el resultado
long _fastShl( long nNumber, int nRota )
  {
  return( nNumber << nRota );
  }

// divide dos longs sin signo y devuelve el resultado
long _fastDiv( long nDividendo, long nDivisor )
  {
  return( nDividendo / nDivisor );
  }

// multiplica dos longs sin signo y devuelve el resultado
long _fastMul( long nOper1, long nOper2 )
  {
  return( nOper1 * nOper2 );
  }

// calcula el módulo de dos longs sin signo y devuelve el resultado
long _fastMod( long nNumber, long nModulo )
  {
  return( nNumber % nModulo );
  }

// divide dos double y devuelve el resultado
double _fastDivd( double nOper1, double nOper2 )
  {
  return( nOper1 / nOper2 );
  }

// multiplica dos double sin signo y devuelve el resultado
double _fastMuld( double nOper1, double nOper2 )
  {
  return( nOper1 * nOper2 );
  }

// suma dos double y devuelve el resultado
double _fastAddd( double nOper1, double nOper2 )
  {
  return( nOper1 + nOper2 );
  }

// resta dos double y devuelve el resultado
double _fastSubd( double nOper1, double nOper2 )
  {
  return( nOper1 - nOper2 );
  }
```

Este fichero debería ser compilado con *MSC* y luego, desde Borland C, podríamos hacer lo siguiente:

```
CLIPPER MyFunc()
    {
    long nOper1, nOper2;

    nOper1 = 0x2000L;
    nOper2 = 0x1000L;

    // esto generaría error
    // _retl(  nOper1 * nOper2 );

    // esto no generará error
    _retl( _fastMul( nOper1, nOper2 ) );
    }
```

Como puede apreciarse, ya es posible usar *double* y *long* desde *Borland C*. Tan sólo hay que perder 5 minutos codificando nuestro propio API matemático en *MSC*, el cual es muy sencillo, y luego a usarlo desde Borland C.

Lógicamente, la asignación de valores a variables *long* se hace con código *inline* y no genera llamadas a funciones.

Acabamos de romper el mito de que *Borland C no puede usarse con Clipper cuando hay rutinas de coma flotante o de enteros largos*, que, como se ve, era falso.

Ya hemos visto el funcionamiento de muchos de los subsistemas de Clipper. Pasemos ahora a los ejemplos práticos prometidos anteriormente.

Cooperación en entornos multitarea

Ahora que está tan de moda Windows 95, vamos a ver en el fuente 2 cómo podemos hacer que nuestro programa Clipper sea un poco más educado con el entorno. Ser educado significa que ceda tiempos muertos cuando no está haciendo nada para que otros programas puedan procesarse sin que el inactivo programa Clipper consuma ciclos de reloj en procesos no productivos.

```
// --- Fuente 2 ----------------------------------------------------------

static char lIsUsing = -1;

#define EVENTO_IDLE 0x5108

static void auxFunc( PEVENTO pEvento )
  {
  if (( lIsUsing == 1 ) &&( pEvento->nMessage == EVENTO_IDLE ) )  {
    _AX = 0x1680;
    geninterrupt( 0x2F );
    }
  }

CLIPPER SystemIdle()
  {
  _retl( ( lIsUsing == 1 ) );
  if ( lIsUsing == -1 )
    lIsUsing = _sysRegLow( auxFunc );
  if ( _parinfo( 1 ) == 4 )
    lIsUsing = _parl( 1 );
  }
```

La función *SystemIdle()* devuelve un lógico indicando si el sistema está en uso o no. Si se llama con el siguiente parámetro:

```
SystemIdle( .T. )
```

se activa el sistema. Para ello, se crea un nuevo manejador de eventos y se le registra como manejador de bajo nivel (recibe también los eventos del sistema). Este manejador comprueba que el sistema está en uso mediante la variable *lIsUsing* y que el evento que recibe es el de

IDLE (Clipper siempre genera el mismo evento cuando está ocioso, *e0x5108*). Si todo es correcto, genera una interrupción que está soportada por el API de la mayoría de los sistemas multitarea convencionales, como *Windows*, *Windows 95*, *Windows NT*, *OS/2 2.0* y *WARP* para indicarles que ese proceso está ocioso y que puede dedicarle menos tiempo de ejecución durante un lapso de tiempo. Obviamente, cuando este lapso termine el programa volverá a estar normalmente en ejecución, y dado que estos lapsos son muy pequeños no se aprecia en absoluto pérdida de eficiencia, aunque sí ganancia en el resto de los programas en ejecución.

Obviamente, esta rutina también puede hacerse cooperar con el *DOS* (*int28h*) aunque debido a la escasa, por no decir nula, a excepción de *PRINT*, concurrencia del DOS, apenas merece la pena. También podría adaptarse al *API* de *Desqview*, el cual no soporta la llamada anteriormente explicada para Windows y OS/2.

Impresión mediante PRINT

Uno de los habituales caballos de batalla es la programación de los listados y contactos con la impresora.

En primer lugar, la impresora es un dispositivo lento, y por lo tanto el programa debe estar esperando a que la impresora termine para reanudar su ejecución (en DOS puro sin colas de impresión).

En segundo lugar, normalmente el usuario manazas, y el que no lo es tanto, suelen apagar la impresora durante el listado, dejando el listado a medias y un error de impresión que puede afectar en menor o mayor medida al programa Clipper.

Hay una solución a esto, utilizar el *PRINT* del DOS. Este programa, no por ser de todos conocido desde versiones añejas de DOS, es poco potente. Permite crear una cola de ficheros de impresión e imprimirlos en multitarea con el resto del sistema.
Para hacer funcionar el sistema, tan sólo debemos enviar el listado a un fichero, en lugar de a la impresora, y luego decirle a *PRINT* que imprima ese fichero, y lo hará en multitarea. Lógicamente, la impresión de Clipper hacia un fichero es muy rápida, y tampoco aquí habrá pérdida de rendimiento.

Pero se plantea un problema, ¿a qué fichero enviar el listado? Pues bien, podemos usar la función *_tctemp()* del API de ficheros comentado anteriormente para crear un fichero temporal por cada listado. Dejando todos estos temporales en el mismo directorio y comprobando periódicamente si ya ha acabado de imprimirse cada fichero para borrarlo, no debería haber, y de hecho no hay, problemas de ficheros perdidos ocupando espacio de disco.

Para utilizar todo este sistema, presento aquí todo el interfaz de Clipper con *PRINT*. Lo primero que se debe hacer es comprobar que *PRINT* está cargado en memoria, para no utilizar sus servicios si no está. Una vez comprobado esto, tenemos funciones para enviar ficheros a la cola, borrar ficheros de la cola, borrarlos todos, determinar errores en la cola, retornar los ficheros en espera y reanudar la impresión de la cola.

Todas las rutinas del fuente 3 están preparadas para modo protegido con *ExoSpace,* pero son fácilmente adaptables a *Blinker* y/o *CauseWay*. Tan sólo hay que cambiar el nombre de las estructuras de registros y las llamadas a la interrupción.

```
// --- Fuente 3 ----------------------------------------------------------

// devuelve un lógico indicando si print esta cargado
CLIPPER IsPrint()
  {
```

```
  _AX = 0x100;
  geninterrupt( INT_MISCEL );
  _retl( _AL == 0xFF );
  }


// Borra de la cola el fichero especificado y devuelve un lógico de error
CLIPPER pDelFile()
  {
  EXOREGS sRegs;
  LPBYTE  cFileRM;
  LPBYTE  cFilePM = _xalloclow( _parclen( 1 ) + 1 );

  _bcopy( cFilePM, (LPBYTE)_parc( 1 ), _parclen( 1 ) + 1 );
  cFileRM = ExoRealPtr( cFilePM );
  sRegs.ds = FP_SEG( cFileRM );
  sRegs.dx = FP_OFF( cFileRM );
  sRegs.ax = 0x102;
  _retl( ! ( ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs ) & 1 ) );
  _xfreelow( cFilePM );
  }


// Borra todos los ficheros de la cola y devuelve un lógico de error
CLIPPER pDelAll()
  {
  EXOREGS sRegs;

  sRegs.ax = 0x103;
  _retl( !( ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs ) & 1 ) );
  }


// Añade el fichero especificado a la cola. Debe indicarse su nombre completo
con ruta
// de acceso. Devuelve un lógico.
CLIPPER pAddFile()
  {
  EXOREGS sRegs;
  LPWORD  nAux;
  LPBYTE  cPrintPM = _xalloclow( 5 );
  LPBYTE  cPrintRM;
  LPBYTE  cFilePM  = _xalloclow( _parclen( 1 ) + 1 );
  LPBYTE  cFileRM;

  _bcopy( cFilePM, (LPBYTE)_parc( 1 ), _parclen( 1 ) + 1 );
  cFileRM  = ExoRealPtr( cFilePM );
  cPrintRM = ExoRealPtr( cPrintPM );
  cPrintPM[ 0 ] = 0x0;
  nAux = (LPWORD)( cPrintPM + 1 );
  *nAux = FP_OFF( cFileRM );
  nAux = (LPWORD)( cPrintPM + 3 );
  *nAux = FP_SEG( cFileRM );
  sRegs.ds = FP_SEG( cPrintRM );
  sRegs.dx = FP_OFF( cPrintRM );
  sRegs.ax = 0x101;
  _retl(!(ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs ) & 1 ) );
  _xfreelow( cPrintPM );
  _xfreelow( cFilePM );
  }



// Determina si hay algún error en la cola de impresión
CLIPPER pError()
  {
  EXOREGS sRegs;

  sRegs.ax = 0x106;
  _retl( !( ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs ) & 1 ) );
```

```
  }


// reinicializa la cola tras un error. Devuelve un lógico de error
CLIPPER pRestart()
  {
  EXOREGS sRegs;

  sRegs.ax = 0x105;
  _retl( !( ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs ) & 1 ) );
  }



// devuelve un array con los ficheros que hay en la cola
CLIPPER pQueue()
  {
  EXOREGS sRegs;
  LPBYTE  cPointer;
  LPBYTE  cAux;
  INDEX   nInd;
  WORD    nMax = 0;
  WORD    nSelector;

  sRegs.ax = 0x104;
  ExoRMInterrupt( INT_MISCEL, & sRegs, & sRegs );
  cPointer = ExoProtectedPtr(MK_FP(sRegs.ds,sRegs.si ), 0x1000 );
  nSelector = FP_SEG( cPointer );
  cAux      = cPointer;
  for ( nInd = 1; *cPointer != '\0'; nInd ++ )
    {
    nMax++;
    cPointer += 64;
    }
  _reta( nMax );
  cPointer = cAux;
  for ( nInd = 1; nInd <= nMax; nInd ++ )
    {
    _storc( (LPSTR)cPointer, -1, nInd );
    cPointer += 64;
    }
  ExoFreeSelector( nSelector );
  }
```

Todas las funciones, excepto *pQueue(),* devuelven un lógico mediante:

```
_retl( !( ExoRMInterrupt( INT_MISCEL,
                          & sRegs,
                          & sRegs
                        ) & 1 )
    );
```

Aunque pueda parecer complejo, es muy sencillo. *ExoRMInterrupt()* devuelve después de la interrupción los flags a los cuales se les hace un *AND* con *1* para determinar únicamente el estado del flag CARRY, que está situado en el primer bit de la doble palabra de flags. Este flag es el que *PRINT* actualiza para indicar si hay algún error en la cola o en la impresora.

También es cierto que, al tratarse de funciones que manejan cadenas de caracteres, la programación en modo protegido complica un poco las cosas. Deben bloquearse zonas de memoria en modo real para luego asignarles un puntero real que se pasar a la interrupción.

Por último, comentar la técnica usada en *pQueue()* para retornar un array. Primero hemos usado:

```
_reta( n )
```

para retornar un array de *n* posiciones que luego hemos inicializado mediante:

```
_stor*( valor, -1, PosicionEnElArray )
```

Al poner como segundo parámetro *-1* a las funciones de tipo *_stor,* podemos acceder a modificar los valores del array que se pretende retornar.

En fin, que tan sólo hay que compilar y usar. Estas funciones pueden contener fallos, pero están bastante probadas, ya que, como ya he comentado, han sido extraídas de *FAST.LIB*, al igual que las anteriores.