Capítulo XIII: EL ENSAMBLADOR Y EL LENGUAJE C

El lenguaje C es sin duda el más apropiado para la programación de sistemas, pudiendo sustituir al ensamblador en muchos casos. Sin embargo, hay ocasiones en que es necesario acceder a un nivel más bajo por razones de operatividad e incluso de necesidad (programas residentes que economicen memoria, algoritmos rápidos para operaciones críticas, etc.). Es entonces cuando resulta evidente la necesidad de poder emplear el ensamblador y el C a la vez.

Para comprender este capítulo, basta tener unos conocimientos razonables de C estándar. Aquí se explicarán las funciones de librería necesarias para acceder al más bajo nivel, así como la manera de integrar el ensamblador y el C.

13.1 - USO DEL TURBO C y BORLAND C A BAJO NIVEL.

A continuación veremos algunas funciones, macros y estructuras de la librería DOS.H del Turbo C.

13.1.1 - ACCESO A LOS PUERTOS DE E/S.

```
int inp (int puerto);                              /* leer del puerto E/S una palabra (16 bits) */
int inport (int puerto);                           /* leer del puerto E/S una palabra (16 bits) */
unsigned char inportb (int puerto);                /* leer del puerto E/S un byte (8 bits) */
int outp (int puerto, int valor);                  /* enviar al puerto E/S una palabra (16 bits) */
void outport (int puerto, int valor);              /* enviar al puerto E/S una palabra (16 bits) */
void outportb (int puerto, unsigned char valor);   /* enviar al puerto E/S un byte (8 bits) */
```

Aunque pueden parecer demasiadas, algunas son idénticas (caso de inp() e inport()) y otras se diferencian sólo ligeramente en el tipo de los datos devueltos, lo cual es irrelevante si se tiene en cuenta que el dato devuelto es descartado (caso de outp() y outport()). En general, lo normal es emplear inport() e inportb() para la entrada, así como outport() y outportb() para la salida. Por ejemplo, para enviar el EOI al final de una interrupción hardware se puede ejecutar: outportb(0x20, 0x20);

13.1.2 - ACCESO A LA MEMORIA.

```
int peek (unsigned seg, unsigned off);               /* leer la palabra (16 bits) en seg:off */
char peekb (unsigned seg, unsigned off);             /* leer el byte (8 bits) en seg:off */
void poke (unsigned seg, unsigned off, int valor);   /* poner palabra valor (16 bits) en seg:off */
void pokeb (unsigned seg, unsigned off, char valor); /* poner byte valor (8 bits) en seg:off */
unsigned FP_OFF (void far *puntero);                 /* obtener offset de variable tipo far */
unsigned FP_SEG (void far *puntero);                 /* obtener segmento de variable tipo far */
void far *MK_FP (unsigned seg, unsigned off);        /* convertir seg:off en puntero tipo far */
```

Las funciones peek(), peekb(), poke() y pokeb() tienen una utilidad evidente de cara a consultar y modificar las posiciones de memoria. Cuando se necesita saber el segmento y/o el offset de una variable del programa, las macros FP_OFF y FP_SEG devuelven dicha información. Por último, con MK_FP es posible asignar una dirección de memoria absoluta a un puntero far. Por ejemplo, si se declara una variable:

```
char far *pantalla_color;
```

se puede hacer que apunte a la memoria de vídeo del modo texto de los adaptadores de color con:

```
pantalla_color = MK_FP (0xB800, 0);
```

y después se podría limpiar la pantalla con un bucle: `for (i=0; i<4000; i++) *pantalla_color++=0;`

13.1.3 - CONTROL DE INTERRUPCIONES.

```
void enable(void);        /* habilitar interrupciones hardware, equivalente a STI */
void disable(void);       /* inhibir interrupciones hardware, equivalente a CLI */
```

13.1.4 - LLAMADA A INTERRUPCIONES.

Para llamar a las interrupciones es conveniente conocer antes ciertas estructuras y uniones.

```
struct WORDREGS {
    unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS {
    unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};

union   REGS    {
    struct  WORDREGS x;
    struct  BYTEREGS h;
};

struct  SREGS   {
    unsigned int es;  unsigned int cs;  unsigned int ss;  unsigned int ds;
};

struct  REGPACK {
    unsigned    r_ax, r_bx, r_cx, r_dx;
    unsigned    r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

A continuación, se listan las funciones que permiten invocar las interrupciones:

```
int int86(int interrupción, union REGS *entrada, union REGS *salida);
int int86x(int interrupción, union REGS *entrada, union REGS *salida, struct REGS *rsegmento);
void intr(int interrupción, struct REGPACK *registros);
```

Las dos primeras funciones se basan en la declaración de dos uniones: una para entrada y otra para salida, que simbolizan los valores iniciales (antes de llamar a la interrupción) y finales (tras la llamada) en los registros. Si se desea que la misma unión que indica los valores iniciales devuelva los finales, se puede indicar por duplicado:

```
union REGS regs;

regs.h.ah = 0;
regs.h.al = 0x13;                   /* VGA 320x200 - 256 colores */
int86 (0x10, &regs, &regs);         /* cambiar modo de vídeo */
```

La diferencia entre int86() e int86x() reside en que la última permite trabajar con los registros de segmento (la estructura SREGS se puede inicializar con los valores que tienen que tener los registros de segmento antes de llamar a la interrupción; a la vuelta, dicha estructura habrá sido modificada para indicar el valor devuelto en los registros de segmento tras la interrupción).

Hay quien prefiere trabajar con **REGPACK**, que con una sola estructura permite también operar con los registros de segmento y la emplea tanto para enviar como para recibir los resultados. El inconveniente, poco relevante, es que sólo admite registros de 16 bits, lo que suele obligar a hacer desplazamientos y forzar el empleo de máscaras para trabajar con las *mitades* necesarias:

```
struct REGPACK bios;

bios.r_ax = 0x13;                   /* VGA 320x200 - 256 colores */
intr (0x10, &bios);                 /* cambiar modo de vídeo */
```

13.1.5 - CAMBIO DE VECTORES DE INTERRUPCIÓN.

```
void interrupt (*getvect(int interrupción))();               /* obtener vector de interrupción */
void setvect (int interrupción, void interrupt (*rutina)()); /* establecer vector de interrupción */
```

La función getvect() devuelve un puntero con la dirección del vector de interrupción indicado. La función setvect() permite desviar un vector hacia la rutina de tipo interrupt que se indica. Interrupt es una palabra clave del Turbo C que será explicada en el futuro. Por ahora, baste el siguiente programa de ejemplo:

```
void interrupt nueva_rutina();      /* nuestra función de interrupción */
void interrupt (*vieja_rutina)();   /* variable para almacenar el vector inicial */

int main()
{
  vieja_rutina = getvect (5);   /* almacenar dirección de INT 5 (activada con Print Screen) */
  setvect (5, nueva_rutina);    /* desviar INT 5 a nuestra propia rutina de control */
  . . .
  . . .                         /* resto del programa */
  . . .
  setvect (5, vieja_rutina);    /* restaurar rutina inicial de INT 5 */
}
```

```
void interrupt nueva_rutina()   /* rutina de control de INT 5 */
{
  . . .
}
```

13.1.6 - PROGRAMAS RESIDENTES.

```
void keep (unsigned char errorlevel, unsigned tamaño);
```

La función anterior, basada en el servicio 31h del DOS, permite a un programa realizado en C quedar residente en la memoria. Además del código de retorno, es preciso indicar el tamaño del área residente (en párrafos). Es difícil determinar con precisión la memoria que ocupa un programa en C. Sin embargo, en muchos casos la siguiente fórmula puede ser válida:

```
keep (0,   (_SS + ((_SP + area_de_seguridad)/16) - _psp));
```

En los casos en que no lo sea, se le puede hacer que vuelva a serlo aumentando el tamaño del área de seguridad (que en los programas menos conflictivos será 0). Tanto _psp como _SS y _SP están definidas ya por el compilador, por lo que la línea anterior es perfectamente válida (sin más) al final de un programa.

13.1.7 - VARIABLES GLOBALES PREDEFINIDAS INTERESANTES.

```
_version    /* devuelve la versión del DOS de manera completa */
_osmajor    /* devuelve el número principal de versión del DOS: ej., 5 en el DOS 5.0 */
_osminor    /* devuelve el número secundario de versión del DOS: ej., 0 en el DOS 5.0 */
_psp        /* segmento del PSP */
_stklen     /* contiene el tamaño de la pila, en bytes */
_heaplen    /* almacena el tamaño inicial del heap, en bytes (0 para maximizarlo) */
```

De estas variables predefinidas, las más útiles son quizá las que devuelven la versión del DOS, lo que ahorra el esfuerzo que supone averiguarlo llamando al DOS o empleando la función de librería correspondiente. También es útil _psp, que permite un acceso a este área del programa de manera inmediata.

13.1.8 - INSERCIÓN DE CÓDIGO EN LÍNEA.

```
void _ _emit_ _ (argumento,...);
void geninterrupt (int interrupción);
```

Por medio de _ _emit_ _() se puede colocar código máquina de manera directa dentro del programa en C. No es conveniente hacerlo así porque así, ya que alterar directamente los registros de la CPU acabará alterando el funcionamiento esperado del compilador y haciendo fallar el programa. Sin embargo, en un procedimiento dedicado exclusivamente a almacenar código *inline* (en línea), es seguro este método, sobre todo si se tiene cuidado de no alterar los registros SI y DI (empleados muy a menudo por el compilador como variables de tipo *register*). Por medio de geninterrupt() se puede llamar directamente a una interrupción: geninterrupt (interr) es exactamente lo mismo que _ _emit_ _(0xCD, interr) ya que 0xCD es el código de operación de INT. Por ejemplo, para volcar la pantalla por impresora se puede ejecutar geninterrupt(5). Con los símbolos _AX, _AL, _AH, _BX, _BL, _BH, _CX, _CL, _CH, _DX, _DL, _DH, _SI, _DI, _BP, _SP, _CS, _DS, _ES, _SS y _FLAGS se puede acceder directamente a los registros de la CPU. Hay que tomar también precauciones para evitar efectos laterales (una asignación tipo _DS=0x40 no afectará sólo a DS).

13.1.9 - LAS PALABRAS CLAVE INTERRUPT Y ASM.

Con **interrupt <declaración_de_función>**; se declara una determinada función como de tipo interrupción. En estas funciones, el compilador preserva y restaura todos los registros al comienzo y final de las mismas; finalmente, retorna con IRET. Por tanto, es útil para funciones que controlan interrupciones. Para emplear esto, se debería compilar el programa con la opción *test stack overflow* y las variables tipo registro desactivadas. Con **asm** se pueden insertar instrucciones en ensamblador, como se verá más adelante.

13.2 - INTERFAZ C (BORLAND/MICROSOFT) - ENSAMBLADOR.

13.2.1 - MODELOS DE MEMORIA.

Los modelos de memoria constituyen las diversas maneras de acceder a la memoria por parte de los compiladores de C. En el caso del Turbo C se pueden distinguir los siguientes:

TINY: Se emplea en los programas donde es preciso apurar el consumo de memoria hasta el último byte. Los 4 registros de segmento (CS, DS, ES, SS) están asignados a la misma dirección, por lo que existe un total de 64 Kb donde se mezclan código, datos y pila. Los programas de este tipo pueden convertirse a formato COM.

SMALL: Se utiliza en aplicaciones pequeñas. Los segmentos de código y datos son diferentes y no se solapan. Por ello, hay 64 kb para código y otros 64 Kb a repartir entre datos y pila.

	Segmentos			Punteros	
Modelo	Código	Datos	Pila	Código	Datos
Tiny	64 Kb			near	near
Small	64 Kb	64 Kb		near	near
Medium	1 Mb	64 Kb		far	near
Compact	64 Kb	1 Mb		near	far
Large	1 Mb	1 Mb		far	far
Huge	1 Mb	1 Mb (Bloques > 64 Kb)		far	far

MEDIUM: Este modelo es ideal para programas largos que no manejan demasiados datos. Se utilizan punteros largos para el código (que puede extenderse hasta 1 Mb) y cortos para los datos: la pila y los datos juntos no pueden exceder de 64 Kb.

COMPACT: Al contrario que el anterior, este modelo es el apropiado para los programas pequeños que emplean muchos datos. Por ello, el programa no puede exceder de 64 Kb aunque los datos que controla pueden alcanzar el Mb, ya que los punteros de datos son de tipo far por defecto.

LARGE: Empleado en las aplicaciones grandes y también por los programadores de sistemas que no tienen paciencia para andar forzando continuamente el tipo de los punteros (para rebasar el límite de 64 Kb). Tanto los datos como el código pueden alcanzar el Mb, aunque no se admite que los datos estáticos ocupen más de 64 Kb. Este modo es el que menos problemas da para manejar la memoria, no siendo quizá tan lento y pesado como indica el fabricante.

HUGE: Similar al anterior, pero con algunas ventajas: por un lado, todos los punteros son normalizados automáticamente y se admiten datos estáticos de más de 64 Kb. Por otro, y gracias a esto último, es factible manipular bloques de datos de más de 64 Kb cada uno, ya que los segmentos de los punteros se actualizan correctamente. Sin embargo, este modelo es el más costoso en tiempo de ejecución de los programas.

13.2.2 - INTEGRACIÓN DE MÓDULOS EN ENSAMBLADOR.

LA SENTENCIA ASM

La sentencia **asm** permite incluir código ensamblador dentro del programa C, utilizando los mnemónicos normales del ensamblador. Sin embargo, el uso de esta posibilidad está más o menos limitado según la versión del compilador. En Turbo C 2.0, los programas que utilizan este método es necesario salir a la línea de comandos para compilarlos con el tradicional compilador de línea, lo cual resulta poco atractivo. En Turbo C++ 1.0, se puede configurar adecuadamente el compilador para que localice el Turbo Assembler y lo utilice automáticamente para ensamblar, sin necesidad de salir del entorno integrado. Sin embargo, es a partir del Borland C++ cuando se puede trabajar *a gusto*: en concreto, la versión Borland C++ 2.0 permite ensamblar sin rodeos código ensamblador incluido dentro del listado C. El único inconveniente es la limitación del hardware disponible: para un PC/XT, el Turbo C 2.0 es el único compilador aceptablemente rápido. Sin embargo, en un 286 es más recomendable el Turbo C++, mientras que en un 386 modesto (o incluso en un 286 potente) resulta más interesante emplear el Borland C++ 2.0: las versiones 3.X de este compilador son las más adecuadas para un 486 o superior (bajo DOS).

La sintaxis de **asm** se puede entender fácilmente con un ejemplo:

```
main()
{
  int dato1, dato2, resultado;

  printf("Dame dos números: "); scanf("%d %d", &dato1, &dato2);

    asm push ax; push cx;
    asm mov  cx,dato1
    asm mov  ax,0h
  mult:
    asm add  ax,dato2
    asm loop mult
    asm mov  resultado,ax
    asm pop  cx; pop ax;

    printf("Su producto por el peor método da: %d", resultado);
}
```

Como se ve en el ejemplo, los registros utilizados son convenientemente preservados para no alterar el valor que puedan tener en ese momento (importante para el compilador). También puede observarse lo fácil que resulta acceder a las variables. Ah, cuidado con BP: el registro BP es empleado mucho por el compilador y no conviene tocarlo (ni siquiera guardándolo en la pila). De hecho, la instrucción MOV CX,DATO1 será compilada como MOV CX,[BP-algo] al ser una variable local de main().

Esta es la única sintaxis soportada por el Turbo C 2.0; sin embargo, en las versiones más modernas del compilador se admiten las llaves '{' y '}' para agrupar varias sentencias **asm**:

```
      asm {
        push ax; push cx;
        mov  cx,dato1
        mov  ax,0h }
    mult: asm {
        add  ax,dato2
        loop mult
        mov  resultado,ax
        pop  cx; pop ax;
        }
```

SUBRUTINAS EN ENSAMBLADOR

Cuando las rutinas a incluir son excesivamente largas, resulta más conveniente escribirlas como ficheros independientes y ensamblarlas por separado, incluyéndolas en un fichero de proyecto (*.PRJ) seleccionable en los menús del compilador.

Para escribir este tipo de rutinas hay que respetar las mismas definiciones de segmentos que realiza el compilador. Hoy en día existe algo más de flexibilidad; sin embargo, aquí se expone el método general para mezclar código de ensamblador con C.

Veamos el siguiente programa en C:

```
int variable;
extern dato;
extern funcion();

main()
{
  int a=21930; char b='Z';

  variable = funcion (a, b, 0x12345678);

}
```

La variable *variable* es una variable global del programa a la que no se asigna valor alguno en el momento de definirla. Tanto *a* como *b* son variables locales del procedimiento main() y son asignadas con un cierto valor inicial; *funcion()* no aparece por ningún sitio, ya que será codificada en ensamblador en un fichero independiente. A dicha función se le pasan 3 parámetros. La manera de hacerlo es colocándolos en la pila (empezando por el último y acabando por el primero). Por ello, el compilador meterá primero en la pila el valor 1234h y luego el 5678h (necesita dos palabras de pila porque es un dato de tipo long). Luego coloca en la pila el carácter almacenado en la variable *b*: como los valores que se apilan son siempre de 16 bits, la parte alta está a 0. Finalmente, deposita el dato entero *a*. Seguidamente, llama a la función *funcion()* con un CALL que puede ser de dos tipos: corto (CALL/RET en el mismo segmento) o largo (CALL/RETF entre distintos segmentos). Esta llamada a la función, por tanto, provoca un almacenamiento adicional de 2 bytes (modelos TINY, SMALL y COMPACT) o 4 (en los restantes modelos de memoria, que podríamos llamar *largos*).

El esqueleto de la subrutina en ensamblador que ha de recibir esos datos y, tras procesarlos, devolver un resultado de tipo int es el siguiente:

```
DGROUP          GROUP    _DATA, _BSS


_DATA           SEGMENT WORD PUBLIC 'DATA'
                PUBLIC _dato            ; _dato será accesible desde el programa C
_dato           DW    0                 ; valor inicial a 0
_DATA           ENDS


_BSS            SEGMENT   WORD PUBLIC 'BSS'
                EXTRN  _variable:WORD   ; variable externa
_info           DW    ?                 ; sin valor inicial
_BSS            ENDS


_TEXT           SEGMENT BYTE PUBLIC 'CODE'
                ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP

                PUBLIC _funcion         ; _funcion será accesible desde el programa C

_funcion        PROC  NEAR              ; funcion() del C
                PUSH  BP
                MOV   BP,SP
                MOV   BX,[BP+4]         ; recuperar variable 'a'
```

```
                MOV   CX,[BP+6]         ; recuperar variable 'b'
                MOV   AX,[BP+8]         ; AX = 5678h
                MOV   DX,[BP+10]        ; DX = 1234h -> DX:AX = 12345678h
                ; ...
                ; ...
                ADD   CX,BX             ; cuerpo de la función
                ADD   CX,AX
                SUB   CX,DX
                ; ...
                ; ...
                MOV   AX,CX             ; resultado (tipo int)
                MOV   SP,BP
                POP   BP
                RET
_funcion        ENDP

_TEXT           ENDS
                END
```

Como se puede observar, se respetan ciertas convenciones en cuanto a los nombres de los segmentos y grupos. En el segmento **_DATA** se definen las variables inicializadas (las que tienen un valor inicial): *_dato* podría haber sido accedida perfectamente desde el programa en C, ya que es declarada como pública. Por otro lado, en el segmento **_BSS** se definen o declaran las variables que no son inicializadas con un valor inicial (como es el caso de la variable *_variable* del programa C, que fue definida simplemente como *int variable*: en el listado ensamblador se la declara como externa ya que está definida en el programa C). El compilador de C precede siempre de un subrayado a todas las variables y funciones cuando compila, motivo por el cual hay que hacer lo propio en el listado ensamblador. Al tratarse de un modelo de memoria *pequeño*, _BSS y _DATA están agrupados. En el segmento **_TEXT** se almacena el código, es decir, las funciones definidas: en nuestro caso, sólo una (el procedimiento *_funcion*). Como es de tipo NEAR, sólo se podrá emplear con programas C compilados en un modelo de memoria TINY, SMALL o COMPACT (para los demás modelos hay que poner FAR en lugar de NEAR). Esta función de ejemplo en ensamblador no utiliza ninguna variable, pero tanto *_variable* (la *variable* del programa C) como, por supuesto, *_info* o *_dato* son plenamente accesibles.

A la hora de acceder a las variables, hay que tener en cuenta el modelo de memoria: como no emplea más de 64 Kb para código (modelos TINY, SMALL o COMPACT), el compilador sólo ha colocado en la pila el offset de la dirección de retorno (registro IP). Nosotros apilamos después BP (ya que lo vamos a manchar) por lo que el último dato que apiló el programa C antes de llamar a la rutina en ensamblador habrá de ser accedido en [BP+4]. La ventaja de inicializar BP es que luego se pueden introducir datos en la pila sin perder la posibilidad de acceder a los parámetros de la rutina que llama. Si el procedimiento fuera de tipo FAR (modelos MEDIUM, LARGE y HUGE), todos los accesos indexados sobre la pila se incrementarían en dos unidades (por ejemplo, [BP+6] en vez de [BP+4] para acceder a la variable *a*) debido a que también se habría almacenado CS en la llamada. Como se puede observar, la rutina no preserva ni restaura todos los registros que va a emplear: sólo es necesario devolver intactos DS, SS, BP y (por si se emplean variables *register*) SI y DI; los demás registros pueden ser libremente alterados. Como la función es de tipo entero, devuelve el resultado en AX; si fuera de tipo long lo devolvería en DX:AX.

El modelo de memoria también cuenta en los parámetros que son pasados a la rutina en ensamblador cuando no son pasados por valor (es decir, cuando se pasan punteros). En el ejemplo, podríamos haber pasado un puntero que podría ser de tipo corto (para cargarlo en BX, por ejemplo, y efectuar operaciones tipo [BX]). Sin embargo, si se pasan punteros a variables de tipo far (o si se emplea un modelo de memoria COMPACT, LARGE o HUGE) es necesario cargar la dirección con una instrucción LES de 32 bits.

Esta rutina de ejemplo en ensamblador es sólo demostrativa, por lo que no debe el lector intentar encontrar alguna utilidad práctica, de ahí que incluso ni siquiera emplee todas las variables que define.

Evidentemente, cuando el programa C retome el control, habrá de equilibrar la pila sumando 8 unidades a SP (para compensar las 4 palabras que apiló antes de llamar a la función en ensamblador). En general, el funcionamiento general del C en las llamadas a procedimientos se basa en apilar los parámetros empezando por el último y llamar al procedimiento: éste, a su vez, preserva BP y lo hace apuntar a dichos parámetros (a los que accederá con [BP+desp]); a continuación, le resta a SP una cantidad suficiente para que quepan en la pila todas las variables locales (a las que accederá con [BP-desp]); antes de retornar restaura el valor inicial de SP y recupera BP de la pila. Es entonces cuando el procedimiento que llamó, al recuperar el control, se encarga de sumar el valor adecuado a SP para equilibrar la pila (devolverla al estado previo a la introducción de los parámetros).

Desde las rutinas en ensamblador también se puede llamar a las funciones del compilador, apilando adecuadamente los parámetros en la pila (empezando por el último) y haciendo un CALL al nombre de la función precedido de un subrayado: no olvidar nunca al final sumar a SP la cantidad necesaria para reequilibrar la pila.

AVISO IMPORTANTE: Algo a tener en cuenta es que el compilador de C es sensible a las mayúsculas: funcion() no es lo mismo que FUNCION(). Por ello, al ensamblar, es **obligatorio** emplear como mínimo el parámetro **/mx** del ensamblador con objeto de que no ponga todos los símbolos automáticamente en mayúsculas (con /mx se respetan las minúsculas en los símbolos globales y con /ml en todos los símbolos). En MASM 6.0, el equivalente a /mx es /Cx y la opción /Cp se corresponde con /ml.

- **Volver al Índice**