

Sessions de Laboratori

Arquitectura de Computadors i Sistemes Operatius I

Departament d'Arquitectura de Computadors

E. Ayguadé, M. González, E. Salamí

Curs 2004-2005 (Q2)

Sesión 1: Revisión de C

Objetivo

El objetivo de esta primera sesión es presentar el entorno de programación TurboC, hacer un pequeño repaso de los conocimientos de C adquiridos en la asignatura de Introducción a los Ordenadores y familiarizarse con el uso de los operadores a nivel de bit que ofrece el lenguaje de programación C.

Desarrollo

Supongamos que tenemos un programa en C con la siguiente secuencia de instrucciones:

```
#include <stdio.h>
main() {
    int x;
    scanf("%d", &x);
    x = x * 5;
    printf("%d\n", x);
}
```

Este programa lee un número entero a través del teclado, lo multiplica por cinco y por último escribe el resultado en pantalla.

La comunicación entre humano y computador se realiza habitualmente mediante caracteres ('A', 'a', 'B', '3', etc.). Por ejemplo, cuando el computador ha de escribir un número entero en pantalla ('printf("%d\n", x);'), ha de convertir el número a la secuencia de caracteres que lo representan. Este algoritmo es el siguiente:

x=13247	—————▶	'1' '3' '2' '4' '7'
13247 ÷ 10 = 1324	→	resto = 7
1324 ÷ 10 = 132	→	resto = 4
132 ÷ 10 = 13	→	resto = 2
13 ÷ 10 = 1	→	resto = 3
1 ÷ 10 = 0	→	resto = 1

Este es el mismo algoritmo que usaríamos para pasar un número de decimal a binario:

x ₁₀ =49	—————▶	x ₂ =110001
49 ÷ 2 = 24	→	resto = 1
24 ÷ 2 = 12	→	resto = 0
12 ÷ 2 = 6	→	resto = 0
6 ÷ 2 = 3	→	resto = 0
3 ÷ 2 = 1	→	resto = 1
1 ÷ 2 = 0	→	resto = 1

Hay que hacer notar que en este algoritmo se obtienen los dígitos del resultado en el orden inverso a cómo los escribiríamos y que el número de dígitos es variable.

De forma análoga, cuándo se introduce información en un computador a través del teclado (`scanf("%d", &x);`), la rutina `scanf`, ha de leer los dígitos de teclado uno por uno y a partir de ellos obtener el número entero correspondiente.

Programa 1

Realizad un programa escrito en C que muestre por pantalla el valor de una variable entera positiva, utilizando únicamente, la siguiente instrucción de entrada/salida:

```
printf("%c", b);
```

siendo `b` una variable de tipo carácter.

Para realizar este ejercicio hay que utilizar los siguientes operadores de C:

Operación	Semántica	Ejemplo
<code>x = y / z;</code>	división entera	<code>x = 134 / 10; /* x == 13 */</code>
<code>x = y % z;</code>	resto de la división entera	<code>x = 134 % 10; /* x == 4 */</code>

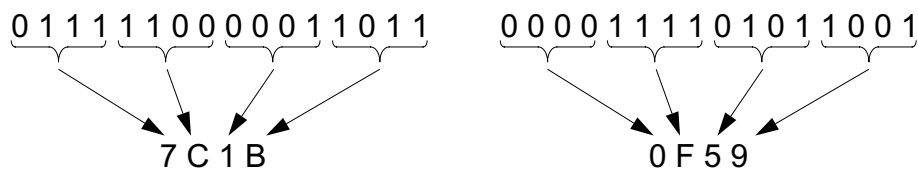
Programa 2

Repetid el ejercicio anterior, pero ahora mostrando el número en binario (base 2).

Programa 3

Repetid el anterior ejercicio, pero ahora mostrando el número en hexadecimal (base 16). Tened en cuenta que los dígitos en hexadecimal son: 1, 2, 3, ..., 9, a, b, c, d, e, f.

El algoritmo utilizado en los tres programas anteriores es muy costoso ya que utiliza las operaciones de división (/) y módulo (%). A continuación vamos a utilizar un enfoque distinto. Para ello recordad que *“toda la información almacenada en un computador se guarda en binario”*. En muchas aplicaciones de bajo nivel, por motivos de velocidad, se utiliza la representación hexadecimal para mostrar datos. La representación en hexadecimal no es más que una forma más compacta de expresar números en binario. Veamos como se codificarían en hexadecimal dos números binarios de 16 bits:



Todos los datos almacenados en un computador tienen un número fijo de bits (8, 16, 32, etc.). Esto lleva a que normalmente, en hexadecimal, se impriman todos los dígitos que forman un dato, incluyendo los ceros en la izquierda.

El lenguaje de programación C permite trabajar directamente sobre constantes especificadas en hexadecimal. Las constantes que comienzan con el prefijo '0x' se interpretan como números en hexadecimal. Igualmente C dispone de operandos que permiten manipular bits. Algunas de estas operaciones en C se muestran en la siguiente tabla:

Operación	Semántica	Ejemplo
<code>x = y >> z;</code>	desplaza y, z bits a la derecha	<code>x = 0xffff >> 7; /* x==0x01ff */</code>
<code>x = y << z;</code>	desplaza y, z bits a la izquierda	<code>x = 0x0040 << 9; /* x==0x8000 */</code>
<code>x = y & z;</code>	realiza una AND bit a bit de y con z	<code>x = 0x4ad5 & 0x0ff0; /* x==0x0ad0 */</code>
<code>x = y z;</code>	realiza una OR bit a bit de y con z	<code>x = 0x11 0x44; /* x==0x55</code>

Las operaciones sobre bits suelen ser muy rápidas, porque son muy simples de implementar en hardware y porque la mayoría de los computadores disponen de instrucciones de lenguaje máquina para realizarlas.

Programa 4

Repetid el programa 3 pero utilizando los operadores de manipulación de bits en lugar del algoritmo de división y módulo, y teniendo en cuenta los siguientes puntos:

- Un número definido como `int` en turboC tiene 16 bits (4 dígitos hexadecimales).
- Se han de imprimir siempre todos los dígitos, aunque sean cero.
- A modo de ejemplo, después de ejecutar estas dos operaciones en C:

```

                                /* x == 0x3d45 */
y = x >> 8;                    /* y == 0x003d */
y = y & 0x000f; /* y == 0x000d */

```

En y tenemos el segundo dígito hexadecimal de x.

Guía breve para la utilización de TurboC

En este apartado veremos como hacer el seguimiento de un programa. Entendemos por seguimiento de un programa, la posibilidad de ejecutarlo paso a paso, controlando en todo momento el valor de las diferentes variables que intervienen.

¿Qué herramientas nos ha de proporcionar el TurboC para hacer el seguimiento de un programa?. Necesitaremos lo siguiente:

- Ejecución paso a paso de nuestro programa. Esto quiere decir que nos ha de permitir ejecutar una única instrucción y que el programa se quede parado hasta una nueva orden, para así poder examinar el valor de las variables que nos interesen. Este tipo de seguimiento se denomina “paso a paso” o “step by step”.
- Mostrar el valor de todas las variables de nuestro programa. En el TurboC esto se hace en la ventana “watch”.
- Ejecución controlada del programa, parándose allí donde nosotros le indiquemos y a partir de este punto continuar el seguimiento en modo “paso a paso”. Esto se denomina “punto de parada” o “breakpoint”.

Cuando sospechemos que uno de nuestros programas no es correcto, lo que tenemos que hacer es depurarlo, es decir, hacer un seguimiento paso a paso para encontrar en qué punto nos hemos equivocado. Para poder hacer una depuración (debugging) el primer paso es cargar el programa mediante **F10/File/Load (F3)**. A continuación compilaremos el programa mediante **F10/Compile/Compile_to_OBJ** y **F10/Compile/Make_EXE**.

Una vez tenemos compilado el programa lo ejecutaremos paso a paso para ver que hace. Para esto hemos de ejecutar instrucción a instrucción con **F10/Run/Trace_into (F7)**. También vemos al lado otra opción: **F10/Run/Step_over (F8)**. ¿Cuál es la diferencia entre las dos? **F7** entra dentro de las funciones ejecutándolas instrucción a instrucción mientras que si estamos sobre una llamada a una función y pulsamos **F8**, se ejecuta la función completa pero sin depurarla paso a paso. Esto sirve para hacer un seguimiento de programas evitando ver paso a paso cosas que ya hemos comprobado que funcionan.

Con lo visto hasta ahora, somos capaces de determinar si las instrucciones de nuestro programa se ejecutan en el orden que queremos. No obstante, necesitamos alguna cosa más para poder descubrir todos los posibles errores de nuestro programa, como puede ser ver los valores de las variables sobre las que estamos trabajando.

Para hacer el seguimiento de una variable hemos de hacer **F10/Break&watch/Add_watch (Ctrl-F7)**. Pruébese de hacer el seguimiento de un programa con **F7** y pulsar **Ctrl-F7**. Aparece una ventana **Add_watch** donde podemos poner el nombre de la variable que queremos seguir. Dentro de la ventana **Watch** podemos ver el valor de la variable que hemos escrito y su variación a lo largo de la ejecución. El número de variables que podemos seguir es muy grande y es posible que no todas las variables quepan en la pantalla. Para movernos por la pantalla **Watch** hemos de hacer un **cambio de pantalla (F6)**. Para volver a la pantalla de edición pulsaremos de nuevo **F6**.

Otra cosa muy interesante es el breakpoint, que es un punto de parada en la ejecución del programa. Si queremos que nuestro programa se ejecute rápidamente hasta una cierta instrucción y al llegar se detenga, colocaremos el cursor sobre dicha instrucción y colocaremos un breakpoint mediante **F10/Break&watch/Toggle_breakpoint (Ctrl-F8)**. La instrucción queda marcada de otro color. Si ahora hacemos un **Run F10/Run/Run (Ctrl-F9)** el programa se ejecutará desde el principio parándose en el breakpoint. En este momento, podemos seguir ejecutando paso a paso (**F7**) o seguidamente (**Ctrl-F9**) hasta el final o el siguiente breakpoint.

Los breakpoints y los watches se pueden borrar, editar, etc. Todas sus opciones se encuentran en el menú **F10/Break&watch**.

Si necesitamos ayuda para editar podemos utilizar el **help de edición (F1)**. Si necesitamos ayuda concreta sobre una palabra o función concretas del lenguaje C podemos utilizar el **help de C (Ctrl-F1)** colocando el cursor encima de la palabra o función que queremos consultar.

Sesión 2: Introducción al i8086

Objetivo

La segunda sesión es una introducción al lenguaje ensamblador del i8086 y al uso de las herramientas de programación asociadas. El objetivo es que el alumno se familiarice con el entorno de programación del lenguaje ensamblador del i8086.

Esquema general de un programa en ensamblador

Todos nuestros programas en ensamblador tendrán el siguiente aspecto:

```
datos SEGMENT
... ;Aquí vendrían los datos de nuestro programa
datos ENDS

codigo SEGMENT
    ASSUME CS: codigo, DS: datos
inicio: MOV AX, datos
        MOV DS, AX

... ;Aquí vendría el código de nuestro programa

        MOV AH, 4CH
        INT 21H
codigo ENDS
END inicio
```

Declaración de variables

La declaración de variables en ensamblador se realiza a través de directivas que asocian espacios de memoria a nombres de variable. Las directivas nos sirven para reservar espacio para una determinada variable. Ese espacio de memoria puede estar inicializado. La forma de declarar una variable en el ensamblador del i8086 es:

```
nombre_variable TIPO expresión
```

Algunos de los posibles tipos son:

- DB (define byte), la variable declarada ocupa un byte (8 bits).
- DW (define word), la variable declarada ocupa 2 bytes (16 bits).
- DD (define double word), la variable declarada ocupa 4 bytes (32 bits).

Veamos algunos ejemplos:

```
indeterm DW ?           ; declara una variable sin inicializar
un_char  DB 'Z'         ; declara un carácter inicializado a 'Z'
vocales  DB 'AEIOU'     ; declara un vector de caracteres inicializado
numero   DW 16          ; declara un entero inicializado a 16
vector   DW 1,3,5,7,9   ; declara un vector de enteros inicializado
vect_1   DW 10 DUP(1)   ; declara un vector de 10 enteros inicializado a 1s.
vect_b   DB 10 DUP('1') ; declara un vector de 10 bytes inicializado a '1'.
vect_i   DW 100 DUP(?)  ; declara un vector de 100 enteros no inicializado.
```

El ensamblador reserva e inicializa las variables definidas, además las coloca en memoria en posiciones consecutivas, tal y como ha indicado el programador.

Descripción de la práctica

En esta sesión se propone la implementación de los siguientes programas.

Programa 1

Escribid un programa en lenguaje ensamblador del i8086 que seleccione una serie de elementos de un vector `vA` y los acumule en una variable entera `acum`. Los elementos a sumar vienen dados por el contenido del vector `vB`: cada casilla de este vector contiene un índice que identifica un elemento del vector `vA` para ser acumulado. Al primer elemento del vector le corresponde el índice 0. El último elemento del vector `vB` siempre es un `-1` indicando el fin del vector.

La declaración de variables para esta versión podría ser la siguiente:

```
vA    DW  23, 45, 6, 78, 42, 69, 123, 1024, 60
vB    DW  4, 1, 5, 2, 6, -1
acum  DW  ?
```

El resultado de la ejecución debería dejar en `acum` el valor 285 (=42+45+69+6+123).

Programa 2

Escribid un programa en lenguaje ensamblador del i8086 que, dada una frase almacenada en un vector `vFrase`, aplique una permutación de letras indicada por un vector `vPerm`. Así pues, si en la casilla 13 del vector `vPerm` hay un 5, el carácter en la posición 13 del vector `vFrase` tiene que intercambiarse con el carácter en la posición 5. Los elementos de los vectores están numerados de 0 en adelante. El programa termina cuando se encuentra un `-1` en el vector `vPerm`.

La declaración de variables para esta versión podía ser la siguiente:

```
vFrase DB  'La declaracion de variables para este programa es esta.'
vPerm  DW  3, 6, 13, 4, 7, 7, 7, 7, 33, 41, 38, -1
```

El resultado de la ejecución debería dejar en `vFrase` los siguientes valores:

```
'dlneaLcaegpio de variables para rste croarama es esta.'
```

Programa 3

Escribid un programa en lenguaje ensamblador del i8086 que, dada una frase almacenada en un vector y acabada por un punto ('.'), cuente el número de palabras de la frase y deje el resultado en el registro `DX`. Entre palabra y palabra puede haber más de un blanco, pero podéis suponer que no hay blancos iniciales (antes de la primera palabra), ni blancos finales (entre la última palabra y el punto).

La declaración de variables para esta versión podría ser la siguiente:

```
frase DB  'La declaración de variables para este programa es esta.'
```

Pasos a seguir

- Diseñad el algoritmo, las primeras veces es aconsejable escribir primero el código en C.
- Edición del programa. El programa se puede editar utilizando el TurboC, o cualquier editor de texto. Se ha de generar un fichero con la extensión “.ASM” (p.ej. *practica.ASM*).
- Ensamblado y Montaje del programa. Ver la sección “Guía breve...” anexa a esta sesión.
- Depuración y Ejecución del programa. Ver la sección “Guía breve...” anexa a esta sesión.

Guía breve para la utilización del Turbo Assembler y Turbo Debugger

Programa Assemblador (Turbo Assembler)

Per traduir (assemblar) un programa font en llenguatge assemblador a un programa objecte mitjançant el programa Turbo Assembler utilitzarem la següent comanda:

```
tasm /zi programa
```

on:

- `tasm` es el programa assemblador (Turbo Assembler)
- `programa` es el nom del fitxer que conté el programa font (ha de tenir l'extensió `asm`).
- `/zi` es una opció del procés assemblat (en minúscules ja que si no es ignora) que indica que s'ha d'afegir la informació simbòlica per poder fer servir el programa depurador.

El programa assemblador admet moltes més opcions. Si voleu conèixer-les, podeu invocar el programa `tasm` sense cap paràmetre. El programa assemblador genera un fitxer objecte amb el mateix nom que el del programa font, però amb l'extensió `obj`.

Programa enllaçador o muntador (Linker)

El programa enllaçador genera a partir de un o varis fitxers objecte el programa executable amb extensió `exe`. Això es fa amb la comanda següent:

```
tlink /v programa
```

on

- `tlink` es el programa muntador (enllaçador o linker)
- `programa` es el nom del fitxer objecte (ha de tenir l'extensió `obj`).
- `/v` es una opció del enllaçador (ha de ser en minúscules ja que si no es ignora) que indica que s'ha d'afegir la informació simbòlica per poder fer servir el programa depurador.

El programa enllaçador admet moltes més opcions. Si voleu conèixer-les podeu invocar al programa `tlink` sense cap paràmetre.

Programa depurador (Turbo Debugger)

Per depurar un programa en assemblador es pot fer servir el Turbo Debugger que es un depurador simbòlic bastant potent. A continuació es descriu de forma resumida el seu funcionament. La comanda per executar aquest programa es `td` i té el següent format:

```
td programa
```

on:

- `td` es el programa Turbo Debugger
- `programa` es el nom del fitxer executable (ha de tenir l'extensió `exe`).

Quan es posa en marxa mostra un aspecte semblant al de la Figura 1. A la part superior hi ha una línia amb una sèrie de menús que es descriuran a continuació. Per accedir als menús es pot fer servir:

- la tecla F10 i després la primera lletra de la paraula del menú,
- les tecles del cursor o
- la combinació `Alt+primera_lletra_menú`.

També podem executar directament `td` i després carregar el programa executable des de la pròpia aplicació del Turbo Debugger amb la comanda `Open` del menú `File`. Es pot canviar el directori de treball amb `Change dir`.

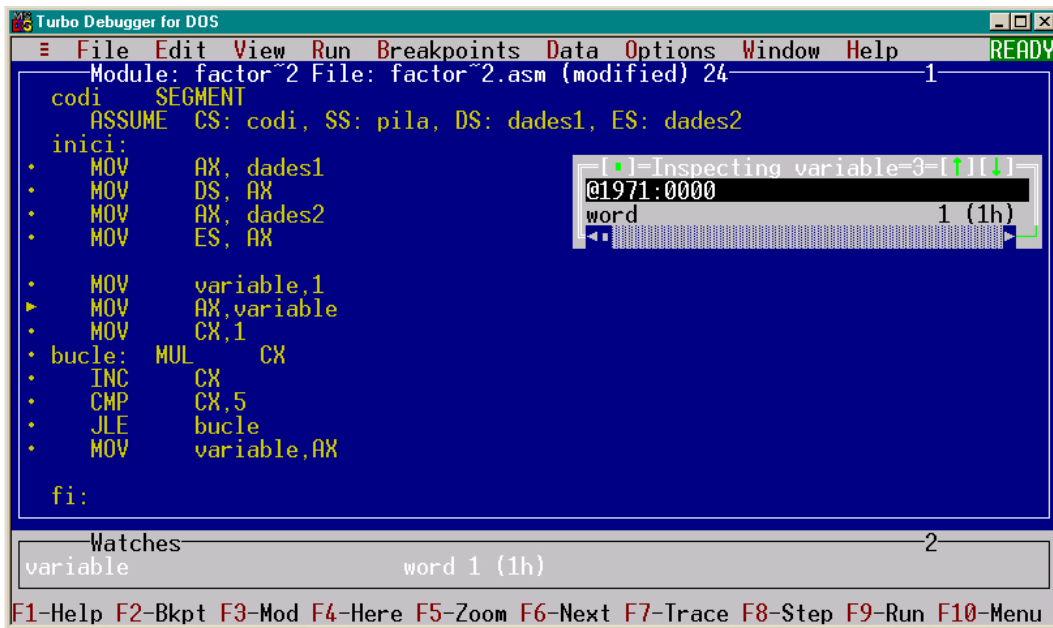


Figura 1. Turbo Debugger amb un programa carregat amb dues variables visibles (una a la finestra watch i l'altre a una finestra especial per aquesta variable).

A partir d'aquí es poden fer servir totes les opcions d'execució pas a pas (Trace into, Step over, etc.) i veure les variables (Inspect, Add watch, etc.) per buscar els errors del programa.

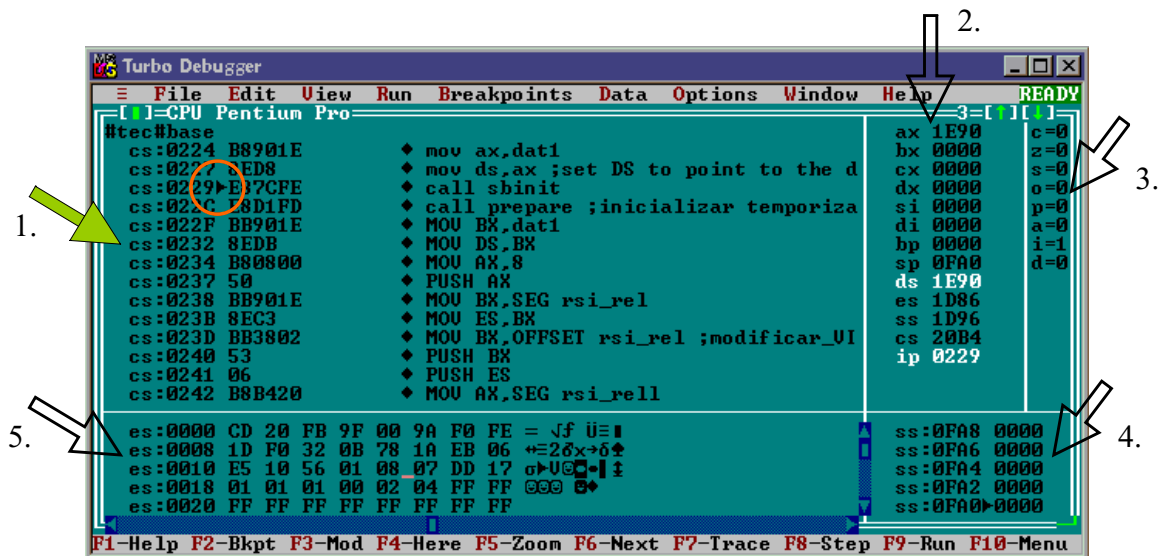


Figura 2. Turbo Debugger amb la finestra de la CPU.

Aquesta es la finestra (Figura 2) que s'utilitzarà normalment per depurar un programa (F10-View-CPU). La finestra permet visualitzar totes les dades necessàries per la depuració d'un programa:

1. **Codi:** Conté el codi del programa que estem depurant. El triangle negre marcat per la circumferència indica que el programa s'ha executat fins aquí.
2. **Registres:** Mostra el contingut dels registres de la CPU. Els registres que han canviat el seu valor es posen de color blanc.

3. **Flags:** Mostra els flags del processador.
 4. **Pila:** Mostra la pila del processador.
 5. **Dades:** Mostra les dades.
- Per situar-nos en cadascun dels apartats utilitzarem la tecla TAB.

Es possible averiguar en quina posició de memòria es troben emmagatzemades les variables del nostre programa (seqüència Menú view, Apartat Variables). Ens apareixerà una finestra com la mostrada en la Figura 3.

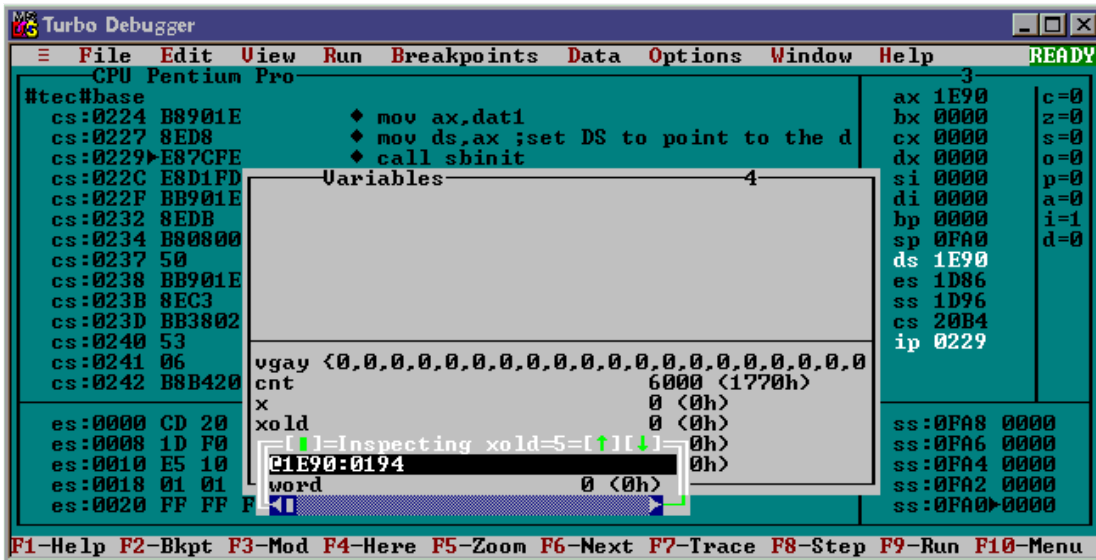


Figura 3. Consultar els valors de les variables amb el Turbo Debugger.

En l'apartat de dades es també possible anar a una posició de memòria concreta utilitzant Ctrl-G. Per visualitzar el contingut d'una posició de memòria només cal introduir la adreça desitjada dins de la finestra que apareixerà. Per exemple, podem posar la adreça 1E90:0194 directament o utilitzar el segment de dades ds:0194.

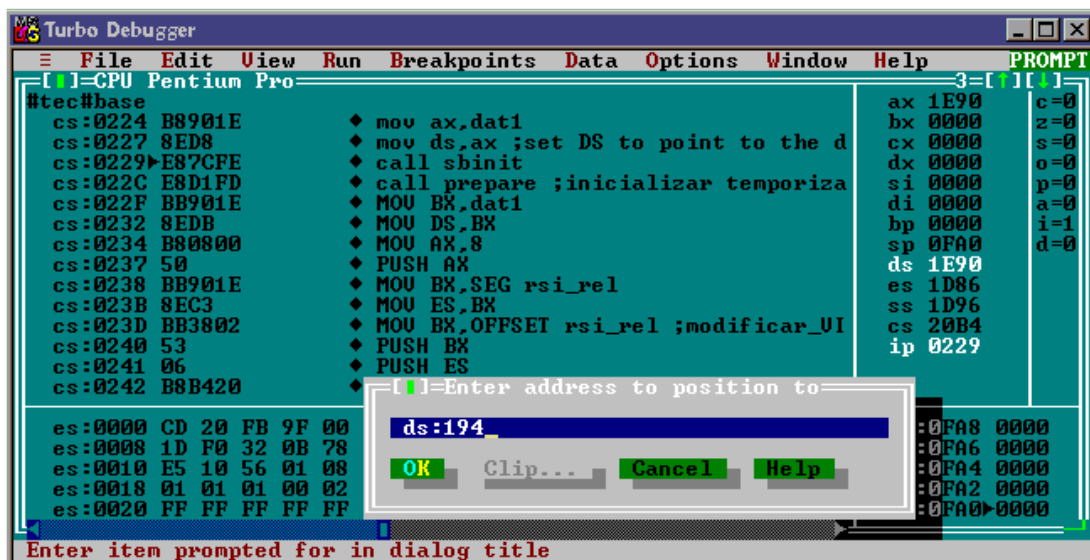


Figura 4. Turbo Debugger introduint una adreça de memòria

Menús principals del Turbo Debugger

File

Open

Carrega el fitxer amb el programa a depurar.

Os shell

Surt temporalment al MS-DOS. Es pot tornar amb la comanda exit.

Quit

Surt del Turbo Debugger.

View

Stack

Mostra el contingut de la pila.

Variables

Mostra l'adreça i el contingut de les variables del programa.

CPU

Mostra el contingut dels registres, els flags, el segment de codi amb les instruccions, el segment de dades i la pila (Figura 2).

Run

Run

Executa el programa.

Go to cursor

Executa el programa fins la línia on és el cursor.

Trace into

Executa una línia i entra a les subrutines.

Step over

Executa una línia i no entra a les subrutines.

Program reset

Inicia el programa i deixa el cursor a la seva primera línia.

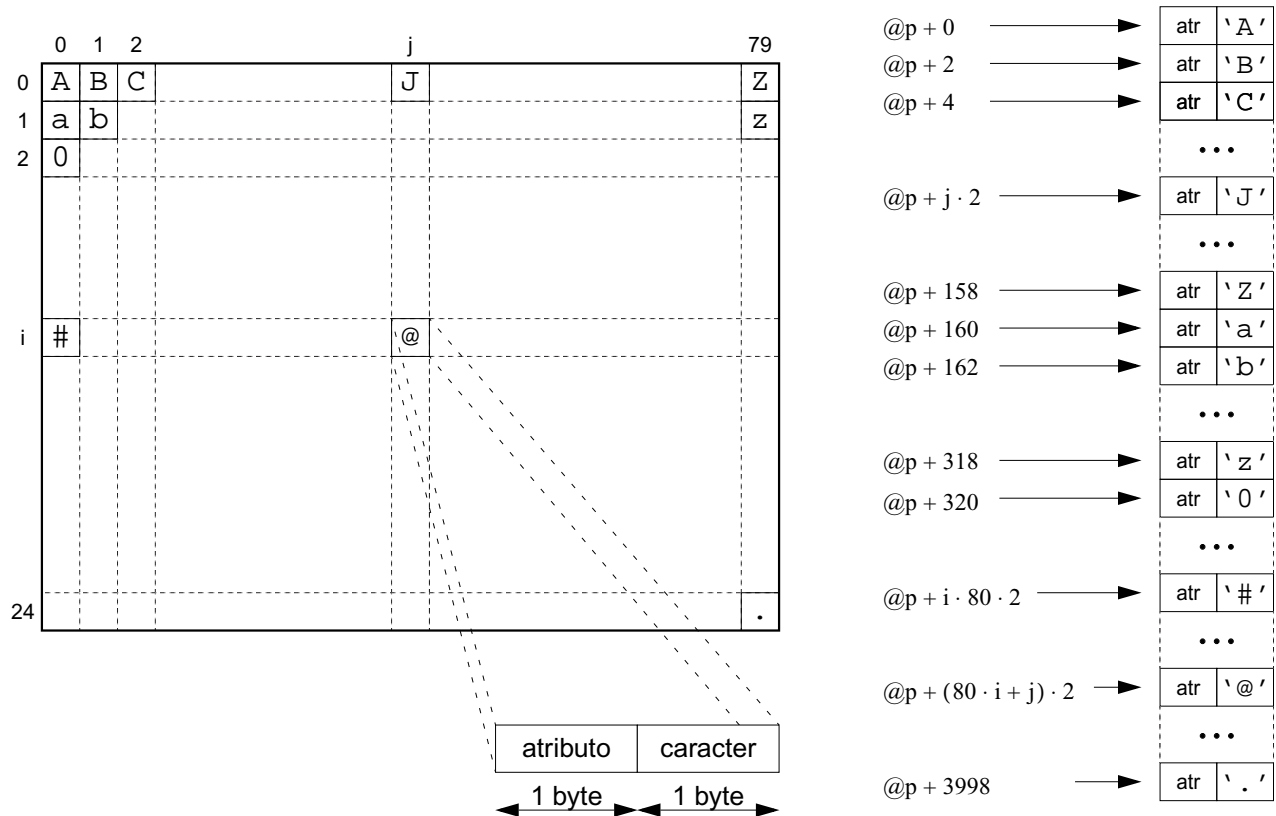
Sesión 3: Estructuras de datos

Objetivo de la práctica

El objetivo de esta sesión de laboratorio es poner en práctica los ejercicios de traducción de C a lenguaje ensamblador del i8086 propuestos durante las sesiones de teoría. En esta sesión trabajaremos con estructuras complejas de datos: matrices y estructuras. Para la realización de esta sesión será necesario conocer el funcionamiento de la pantalla en modo texto.

La pantalla en modo texto

La pantalla en modo texto de los computadores personales compatibles, puede verse como una matriz de 80 columnas por 25 filas. En cada una de las entradas de esta matriz se codifica un carácter mediante una palabra de 16 bits, en el byte bajo de la palabra se guarda el código ASCII del carácter y en el byte alto el atributo que caracteriza su visualización. Internamente, esta matriz está almacenada por filas en una determinada zona de memoria.



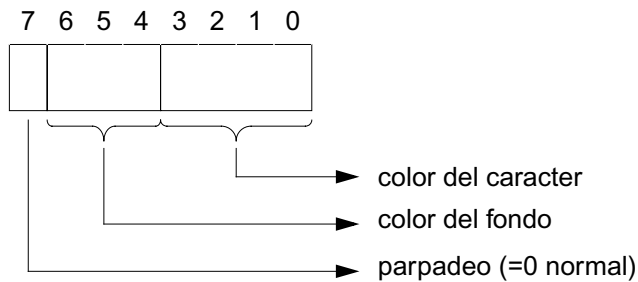
La figura muestra la relación existente entre la pantalla y el buffer de memoria que la representa. Para visualizar un carácter en la posición (i, j) de pantalla, debe escribirse su código ASCII y el atributo correspondiente en la entrada (i, j) de la matriz que representa la pantalla. Debido a que la matriz está almacenada por filas en posiciones consecutivas de memoria, la entrada (i, j) de la matriz corresponde a la entrada $80 \times i + j$ del buffer que contiene la pantalla. En definitiva, si queremos escribir en la posición (i, j) de la pantalla un carácter CAR, con un atributo ATR, lo haremos en las siguientes direcciones:

$$[@p + (80 \cdot i + j) \cdot 2] \leftarrow \text{CAR}$$

$$[@p + (80 \cdot i + j) \cdot 2 + 1] \leftarrow \text{ATR}$$

siendo @p, la dirección inicial de la pantalla.

Cada una de las entradas del buffer de memoria es una palabra de 2 bytes. En el byte alto se guarda el atributo del carácter indicado en el byte bajo de la palabra. Este atributo tiene dos interpretaciones dependiendo de si el monitor es B/N o color. En la siguiente figura se muestra la interpretación del atributo:



código	color ⁺
000	negro
001	azul
010	verde
011	cyan
100	rojo
101	púrpura
110	marrón
111	blanco

+ el bit 3 del color del carácter sólo influye en la intensidad del mismo.

Nótese que una selección poco cuidadosa del atributo, por ejemplo carácter negro y fondo negro, puede dar la impresión de que no estamos escribiendo nada en la pantalla, cuando en realidad lo que ocurre es que no podemos distinguirlo.

Un detalle muy importante a conocer es la dirección de memoria en dónde se encuentra la primera posición del buffer de pantalla (@p). Esta dirección depende de la placa gráfica disponible y del modo de video actual. Los modos de video que trabajan con la pantalla en modo alfanumérico son 2, 3 y 7. Las direcciones iniciales de la pantalla son B800:0000, B000:0000 y B000:0000 respectivamente (todas las direcciones están en hexadecimal con el formato segmento:desplazamiento). Para obtener la dirección de pantalla, sólo nos resta conocer el modo de video activo. MS-DOS contiene en la dirección 0000:0449 el modo de video activo en ese momento. A continuación se muestran en C y en ensamblador las subrutinas necesarias para calcular la dirección actual de la pantalla:

```

int far *dir_pantalla()
{
    unsigned char far *punt;

    punt = 0x449;
    if (*punt == 7) return(0xb0000000);
    else if (*punt == 2) return(0xb8000000);
    else if (*punt == 3) return(0xb8000000);
    else return(-1);
}

_dir_pantalla PROC FAR
    push es
    xor ax, ax
    mov es, ax
    mov dl, es:[449H]
    cmp dl, 7
    je b0
    cmp dl, 2
    je b8
    cmp dl, 3
    je b8
error: mov ax, -1
        mov dx, -1
        jmp fin
b0:    mov dx, 0B000h
        jmp fin
b8:    mov dx, 0B800H
fin:   pop es
        ret
_dir_pantalla ENDP

```

Descripción de la práctica

En primer lugar se propone traducir al lenguaje ensamblador del i8086 un programa escrito en el lenguaje de programación de alto nivel C (programa 1), para posteriormente, y basándose en él, confeccionar un programa en ensamblador capaz de realizar la función requerida a partir de las especificaciones dadas (programa 2).

Programa 1

Este programa borra la pantalla y a continuación muestra una frase en horizontal a partir de una coordenada determinada. Se recomienda leer atentamente las notas que aparecen.

```
#define LONG_FRASE 4
#define X_POS 40
#define Y_POS 18

typedef struct {
    char caracter;
    char atributo;
} elemento;

typedef elemento pant[25][80]; /* nota 1 */
extern pant far *dir_pantalla(); /* nota 2 */

char frase[LONG_FRASE+1]="HOLA";

main()
{
    pant far *pantalla; /* nota 3 */
    int i, j, k;

    pantalla = dir_pantalla(); /* nota 4 */

    /* borrar pantalla */
    for (i=0; i<25; i++) {
        for (j=0; j<80; j++) {
            (*pantalla)[i][j].caracter = ` `;
            (*pantalla)[i][j].atributo = 0x00;
        }
    }

    /* dibujar frase horizontal */
    for (k=0; k<LONG_FRASE; k++) {
        (*pantalla)[Y_POS][X_POS+k].caracter = frase[k];
        (*pantalla)[Y_POS][X_POS+k].atributo = 0x40;
    }
}
```

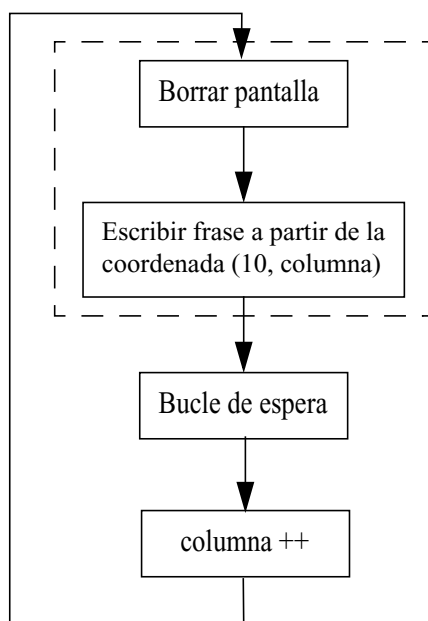
Notas:

- Nota 1: esta instrucción define el “tipo de datos” `pant`, como una matriz de 25 filas por 80 columnas, cada uno de los elementos de esta matriz es una estructura con 2 caracteres que corresponden al carácter y al atributo respectivamente.
- Nota 2: rutina externa que devuelve la dirección inicial de la pantalla en modo texto. Esta rutina, ya implementada en ensamblador en el programa que se os proporciona, devuelve como resultado dicha dirección en los registros DX y AX.

- Nota 3: no es necesario codificar en ensamblador la variable `pantalla`. En el programa en ensamblador deben utilizarse los registros `ES/BX` para almacenar dicho puntero.
- Nota 4: a partir de esta sentencia, que ya está traducida en el programa que se os proporciona, los registros `ES/BX` están inicializados con la dirección `DX/AX` que devuelve la función `dir_pantalla()`, es decir, contienen la dirección de memoria a partir de la cual se almacena la matriz de la pantalla.

Programa 2

A partir del programa anterior se pide la codificación de un programa en ensamblador que muestre la frase anterior (“HOLA”) moviéndose en horizontal sobre la línea 10 de la pantalla. Para ello, el algoritmo que se propone es el siguiente:



El bucle de espera se debe incorporar para poder visualizar lo escrito en pantalla, ya que si no el texto se escribiría y se borraría a tal velocidad (propia de la máquina) que no daría tiempo a ver nada. Para codificar dicho bucle de espera se propone algo tan sencillo como realizar un bucle anidado de un determinado número de iteraciones que no ejecute ninguna acción, con lo cual se provocará una pérdida de tiempo a la CPU. El número de iteraciones total deberá estar sobre 50000000, aunque este valor dependerá de la velocidad de la máquina e incluso de la percepción subjetiva de cada usuario, por lo que se propone realizar varias pruebas hasta encontrar el valor adecuado.

Por último, como condición de salida (fin de la ejecución) se propone que el mensaje recorra la línea horizontal de izquierda a derecha únicamente dos veces.

Sesión 4: Subrutinas, enlace entre Ensamblador i8086 y C

Objetivo de la práctica

El objetivo de esta práctica es familiarizarse con la interficie entre el lenguaje de programación de alto nivel C, y el lenguaje ensamblador del i8086. Se mostrará como obtener un ejecutable a partir de un módulo escrito en C (que contiene el programa principal), y de otro módulo escrito en ensamblador (que contiene las subrutinas).

Las herramientas que se utilizarán en la realización de esta práctica son el *Turbo Assembler* (descrito en las sesiones anteriores), el *Turbo C*, y el *Turbo Debugger*.

Enlace entre código ensamblador y C

Interficie en ensamblador

A continuación se describen los aspectos que hay que tener en cuenta para poder mezclar código escrito en ensamblador de i8086 y código escrito en C.

Las rutinas en ensamblador deben estar implementadas siguiendo el mecanismo de gestión de subrutinas explicado en clase de teoría:

- El paso de parámetros se realiza a través de la pila empezando por el parámetro de la derecha.
- Los vectores se pasan siempre por referencia.
- Todas las direcciones son de tipo FAR (segmento:desplazamiento).
- El retorno de resultados es a través del registro AX.

El fichero en ensamblador contendrá el segmento de código con un conjunto de rutinas. Debe tenerse en cuenta que las rutinas que estén compartidas por los ficheros C y ensamblador tienen el mismo nombre en los dos ficheros, excepto que en el fichero ensamblador éste debe empezar por el carácter ‘_’. El segmento de código que implementaremos tendrá el siguiente aspecto:

```
codigo SEGMENT PUBLIC
    ASSUME CS:codigo
    _rutina1 PROC FAR
        ...
        ;código de la rutina 1
        ...
    _rutina1 ENDP
PUBLIC _rutina1
    _rutina2 PROC FAR
        ...
        ;código de la rutina 2
        ...
    _rutina2 ENDP
PUBLIC _rutina2
codigo ENDS
END
```

Si el fichero ensamblador accede a variables globales definidas en el fichero C, éstas deberán estar declaradas al principio del fichero ensamblador (fuera del segmento de código) de la siguiente forma:

```
extrn _nomvar: tipo
```

donde *nomvar* es el nombre de la variable tal y como aparece en el fichero C (empezando por ‘_’), y el parámetro *tipo* es el tamaño que ocupa la variable, pudiendo ser (textualmente) *byte*, *word*, o *dword*.

Finalmente tener en cuenta que el fichero ensamblador sólo debe ser ensamblado, ya que el montaje es llevado a cabo por el Turbo C. El ensamblaje se realizará, asumiendo que el fichero ensamblador se llama *nombre.asm*, de la siguiente forma:

```
A:\> tasm /mx /zi nombre
```

La opción */mx* indica al Turbo Assembler que conserve mayúsculas y minúsculas en el nombre de las variables y rutinas, que es tal y como lo hace el entorno del Turbo C. La correcta ejecución de este comando da como resultado el fichero *nombre.obj*.

Interficie en C

Para crear desde el Turbo C un ejecutable a partir de más de un fichero, se tiene que crear lo que se llama un proyecto. Para esto se debe crear un fichero proyecto (que lleva como extensión “.PRJ”) y que contiene únicamente las siguientes líneas:

```
nomfitxer1.c  
nomfitxer2.c  
nomfitxer3.obj  
...  
nomfitxerN.obj
```

donde *nomfitxeri.c* es el nombre de cada fichero en C que forme parte del programa, y *nomfitxerj.obj* es el nombre de cada uno de los ficheros objeto (ficheros previamente ensamblados) que se vaya a utilizar. El programa ejecutable creado por el Turbo C lleva el mismo nombre que el fichero *project*, pero con extensión *exe*.

Al abrir el Turbo C y cargar el programa en C, hay que ir a la opción *Project* y seleccionar la subopción *Project Name*. Aparecerá una ventana en la que hay que especificar el nombre del fichero *project*. A partir de este momento ya se puede compilar y ejecutar de la forma habitual como se hace para cualquier programa en C.

Se pueden tener tantos ficheros C como se desee (siempre y cuando estén declarados en el fichero *project*). Para referenciar una rutina implementada en un fichero diferente al actual, se tiene que informar al Turbo C que se está usando una rutina externa, de la siguiente forma:

```
extern void far nomrutina(parametros);
```

donde *parametros* es la lista de parámetros tal y como se declararía en la rutina en C, esto es, tipo de parámetro y nombre de cada parámetro separado por comas.

Las variables que deban ser compartidas tanto por alguna de las rutinas del programa escrito en C, como por cualquier rutina que se encuentre en otro fichero diferente, ha de ser declarada como variable global. Esto es, debe ser declarada al principio del fichero en C, fuera de cualquier rutina.

Una vez finalizada la escritura del programa principal, teniendo en cuenta las consideraciones descritas anteriormente, ya se puede compilar y ejecutar como es habitual en cualquier programa escrito en C.

Descripción de la práctica

En esta sesión de laboratorio se pide la programación en ensamblador de un conjunto de rutinas que permitirán escribir o dibujar en posiciones concretas de la pantalla. En concreto se pide la implementación de las siguientes rutinas:

```
void escribir_pantalla(pant far *p, int fil, int col, elemento letra)
{
    (*p)[fil][col] = letra;
}

void escribir_tramo(pant far *p, int fil, int col, char tramo)
{
    int i;
    char mask;
    elemento blanco;

    blanco.caracter = ' ';
    blanco.atributo = 0x40;
    mask = 0x01;

    for (i=7; i>=0; i--) {
        if (tramo & mask) {
            escribir_pantalla(p, fil, col+i, blanco);
        }
        mask = mask << 1;
    }
}

void escribir_figura(pant far *p, int N, char *tramos, int *fil, int *col)
{
    int i;
    for (i=0; i<N; i++) {
        escribir_tramo(p, fil[i], col[i], tramos[i]);
    }
}
```

donde los tipos `pant` y `elemento` son los mismo que se utilizaron en la anterior sesión de laboratorio.

La rutina `escribir_tramo` dibuja un tramo de línea acorde con los bits en la variable `tramo`. Un 1 significa dibujar un espacio relleno, un 0 significa dejar un espacio en blanco. Así pues, un tramo a dibujar podría ser una línea continua de 8 espacios ($\text{tramo} = 255_{10} = 1111111_2$) o bien una línea discontinua ($\text{tramo} = 170_{10} = 10101010_2$).

Para comprobar el desarrollo correcto de estas rutinas, se deberá usar el programa en C que se os proporcionará en la sesión de laboratorio. Se recomienda probar cada rutina por separado antes de pasar a la siguiente.

Sesión 5: Gestión de los dispositivos de Entrada / Salida

Objetivo de la práctica

El objetivo de esta práctica es familiarizarse con la gestión de algunos de los dispositivos de entrada y salida del IBM-PC. En esta sesión de laboratorio se ha de gestionar la pantalla, el reloj y el teclado.

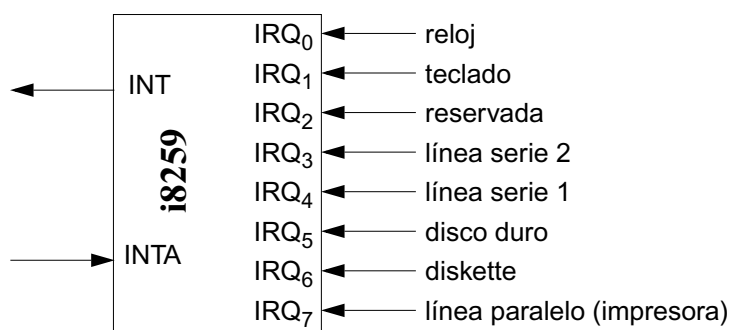
Descripción de los periféricos del PC

Los dispositivos de E/S de los ordenadores personales compatibles son diferentes de los utilizados en las clases de teoría. En este apartado se describen de forma breve los periféricos que deberán utilizarse para la realización de esta sesión, a excepción de la pantalla que ya fue descrita en la sesión 3.

1. Controlador de Interrupciones

Los ordenadores personales compatibles disponen de un controlador de interrupciones basado en el i8259. Este controlador se encarga de recibir las peticiones de interrupción de los periféricos del sistema y decide qué interrupción ha de atender el procesador.

Este controlador de interrupciones puede recibir peticiones de hasta 8 dispositivos diferentes, a través de las líneas IRQ_i ($0 \leq i \leq 7$). La siguiente figura muestra cómo está conectado el controlador de interrupciones en algunos ordenadores personales compatibles.



Cuando hay más de una petición de interrupción, el controlador ha de seleccionar una de ellas en función de un algoritmo de prioridades. El controlador está programado de tal forma que la IRQ_0 (reloj) es la más prioritaria, y la IRQ_7 (impresora) la menos prioritaria.

Cuando el controlador de interrupciones acepta una interrupción, activa la señal de interrupción del procesador (INT), cuando el procesador acepta la interrupción lo indica activando la señal INTA. En respuesta a esta señal el controlador de interrupciones envía, a través del bus de datos, un número para identificar cual es la interrupción que ha de atender el procesador. Este número es $8 + i$ (IRQ_i), y es el número que utiliza el procesador para indexar en el vector de interrupciones y obtener la dirección de la rutina de atención a la interrupción correspondiente.

El controlador mantiene información de las interrupciones pendientes de servir y de las que se están sirviendo. Si se está sirviendo una interrupción cualquiera y llega otra de más prioridad, el controlador interrumpirá al procesador. Esta propiedad obliga a que las rutinas de interrupción, al acabar, envíen una señal al controlador de interrupciones indicando que la rutina de interrupción ha terminado.

Del controlador de interrupciones sólo nos interesan dos registros (ambos están en el espacio de direcciones de E/S):

- El **registro de control** que está en la dirección 0x20. Este registro se ha de utilizar para avisar al controlador de interrupciones que se ha acabado de servir una interrupción. Esta operación se puede realizar mediante la siguiente subrutina:

```
#define EOI 0x20
#define RegIntCon 0x20

void EndOfInterrupt(){
    outp(RegIntCon, EOI);
}
```

- El **registro de máscara** que está en la dirección 0x21. Este registro de 8 bits permite enmascarar de forma individual cada una de las 8 interrupciones que soporta el controlador de la siguiente forma:

- bit $i == 0$, la interrupción IRQ_i está permitida
- bit $i == 1$, la interrupción IRQ_i está inhibida

Inicialmente todas las interrupciones están permitidas.

Una rutina para inhibir la IRQ_i , podría ser la siguiente:

```
#define RegIntMasc 0x21

void InhInt(int i) {
    unsigned char m;

    m = inp(RegIntMasc);
    m = m | (0x01 << i);
    outp(RegIntMasc, m);
}
```

Una rutina para permitir la IRQ_i , podría ser la siguiente:

```
void PermInt(int i) {
    unsigned char m;

    m = inp(RegIntMasc);
    m = m & ~(0x01 << i);
    outp(RegIntMasc, m);
}
```

/ NOTA IMPORTANTE: la i corresponde a la $IRQ(i)$, es decir, 0 para el reloj, 1 para el teclado, etc. */*

El lenguaje ensamblador del i8086 permite prohibir todas las interrupciones mediante la instrucción **CLI**. De la misma forma, se pueden permitir interrupciones mediante la instrucción **STI**. Estas instrucciones no afectan al controlador de interrupciones, sino que hacen que el procesador ignore las peticiones de interrupción del controlador.

Finalmente, hay que señalar que cada vez que se pone en marcha una rutina de atención a una interrupción, lo primero que hace el procesador es inhibir interrupciones. En consecuencia, si la rutina no indica lo contrario (mediante una instrucción **STI**), no se aceptará ninguna nueva interrupción hasta que acabe la rutina.

2. Reloj

El reloj es uno de los dispositivos de entrada / salida más simples de los ordenadores personales compatibles. Su única función es generar interrupciones a una frecuencia fija de 18,2 interrupciones por segundo.

El reloj genera interrupciones a través de la entrada IRQ₀, en consecuencia es siempre la más prioritaria. La entrada del vector de interrupciones asignada a este dispositivo es la 8.

Recordad que la última instrucción de la rutina de interrupción de reloj siempre ha de ser una llamada a la rutina EndOfInterrupt () (o una equivalente).

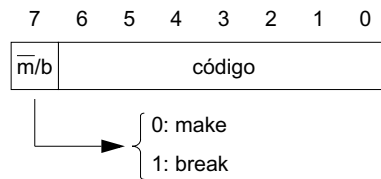
3. Teclado

Los teclados de los ordenadores personales compatibles están controlados por un microprocesador muy simple. Este microprocesador realiza las siguientes funciones:

- Detecta cuándo se ha pulsado (o soltado) una tecla.
- Si alguna tecla permanece pulsada más de medio segundo, a partir de ese momento y a intervalos regulares, genera operaciones de entrada / salida que simulan el efecto de pulsar una tecla.
- Mantiene un buffer con las últimas 20 teclas, en previsión de que el procesador no pueda leerlas a la suficiente velocidad en un instante determinado.

Código de rastreo

Cada vez que se pulsa o libera una tecla, el controlador de teclado genera un código de rastreo de 8 bits que deposita en el registro de datos de teclado que se encuentra en la dirección 0x60 del espacio de direcciones de entrada salida. Este código identifica la tecla pulsada (o liberada). El código de rastreo de una misma tecla es diferente al pulsar (**make**) que al liberar (**break**). El formato del código de rastreo es el siguiente:



En la siguiente figura aparece el valor de *código* para una teclado expandido de 102 teclas.

1	59	60	61	62	63	64	65	66	67	68	87	88	-	70	-					
41	2	3	4	5	6	7	8	9	10	11	12	13	14	82	71	73	69	53	55	74
15	16	17	18	19	20	21	22	23	24	25	26	27	28	83	79	81	71	72	73	78
58	30	31	32	33	34	35	36	37	38	39	40	43	72	75	76	77	75	76	77	28
42	86	44	45	46	47	48	49	50	51	52	53	54	75	80	77	79	80	81	28	
29	56	57					56	29	75	80	77	82	83	28						

Nótese que el código de rastreo no tiene ninguna relación con el código ASCII del carácter asociado a cada tecla.

Sincronización

La sincronización de las operaciones de entrada / salida por teclado se puede realizar tanto por encuesta como por interrupciones:

- **Sincronización por interrupciones.** Cada vez que se pulsa (o libera) una tecla se produce una interrupción a través de la IRQ_1 . La entrada del vector de interrupciones asignada a este dispositivo es la 9.
- **Sincronización por encuesta.** Para trabajar por encuesta con el teclado es imprescindible inhibir las interrupciones de teclado a través del controlador de interrupciones. Una vez inhibidas las interrupciones de teclado se puede trabajar con el teclado por encuesta. Para ello utilizaremos el registro de estado de teclado que se encuentra en la dirección $0x64$ del espacio de direcciones de entrada salida. De este registro, el bit 0 nos indica, cuando se activa a 1, que hay un dato pendiente de leer del registro $0x60$ (registro de datos de teclado).

Soporte de TurboC a las operaciones de Entrada/Salida

A continuación se presentan las herramientas básicas que ofrece TurboC para realizar operaciones de Entrada / Salida.

- Es imprescindible utilizar la librería “dos.h”:

```
#include <dos.h>
```

- Acceso a los puertos de entrada / salida de los controladores:

```
outp(port, dato); /* ES[port] <- dato */
dato = inp(port); /* dato <- ES[port] */
```

- Inhibir / permitir interrupciones

```
disable(); /* Inhibir interrupciones: CLI */
enable(); /* Permitir interrupciones: STI */
```

- Definición de una rutina de interrupción

```
void interrupt RAI()
{
    ... /* código de la interrupción */
}
```

- Montar el vector de interrupciones: antes de montar el vector de interrupciones, con nuestra rutina de interrupción (RAI), es imprescindible salvar la rutina de interrupción del Sistema Operativo, para restaurarla antes de acabar el programa:

```
void main()
{
    void far *raiSO; /* puntero a una rutina */
    ... /* resto de la declaracion de variables */

    /* montar el nuevo vector de interrupciones y salvar el antiguo */
    disable();
    raiSO = getvect(numRAI);
    setvect(numRAI, (void interrupt (*)()) RAI);
    enable();

    ... /* programa principal */

    /* restaurar el antiguo vector de interrupciones */
    disable();
    setvect(numRAI, (void interrupt (*)()) raiSO);
    enable();
}
```

Comentarios previos

Antes de comenzar con la descripción de la práctica es necesario realizar algunas puntualizaciones:

- No se podrá utilizar ninguna rutina de librería para realizar operaciones de entrada / salida. Sólo se podrán utilizar las rutinas incluidas en la librería `dos.h`.
- Para la realización de esta sesión se os facilitará el fichero objeto `PANTALLA.OBJ`, que además de la rutina `escribir_pantalla` diseñada en la sesión 4, contiene también una rutina llamada `leer_pantalla` que devuelve el elemento que hay en una determinada fila/columna. Toda la entrada / salida por pantalla deberá realizarse llamando a dichas funciones, cuya declaración es la siguiente:

```
void escribir_pantalla(pant far *p, int fil, int col, elemento letra);
elemento leer_pantalla(pant far *p, int fil, int col);
```

- Toda la programación de entrada / salida se realizará en C.

Descripción de la práctica

Al acabar esta práctica se ha de obtener un programa que nos permita jugar a un juego muy sencillo que consiste en matar una “pulga” que salta de forma aleatoria, persiguiéndola y atrapándola con una “bola” cuyo movimiento se controla por el teclado. Este programa utiliza la pantalla y el reloj y el teclado por interrupciones. El programa será construido paso a paso y se recomienda que todas las versiones sean guardadas.

Para la realización de esta práctica se os proporcionará un programa en C en el que están declarados los siguientes tipos de datos y variables globales:

```
/** Definicion tipos de datos **/

typedef struct {
    int fil;
    int col;
} posicion;

typedef struct {
    char caracter;
    char atributo;
} elemento;

typedef elemento pant[25][80];

/** Variables globales **/

pant far *pantalla;

elemento blanco = { ' ', 0x07 };
elemento borde = { '*', 0x0B };
elemento pulga = { 'X', 0x0E };
elemento bola = { 'O', 0x0A };

posicion pos_pulga = { 8, 50 };
posicion pos_bola = { 16, 30 };
```

Al igual que en la sesión anterior, `pantalla` es un puntero a la dirección inicial de la pantalla. Se dispone también de cuatro variables inicializadas con el carácter y atributo que representan los distintos elementos del juego: la bola, la pulga, el borde y los espacios libres. Por último, las variables

`pos_pulga` y `pos_bola` deberán indicar en todo momento la fila y columna en que se encuentran los elementos `pulga` y `bola` respectivamente. Estas variables están inicializadas con las coordenadas en que aparecerán la pulga y la bola al inicio del juego.

Versión 1

El programa que se os proporciona dibuja en pantalla un margen de 25 por 80 con el carácter '*'. En esta primera versión, se ha de modificar dicho programa para que, una vez dibujado el margen, espere a que se pulse y suelte la barra espaciadora (ha de esperar el "break" de la tecla). En el momento en que se suelte la barra espaciadora, la bola (carácter 'O') y la pulga (carácter 'X') aparecerán en pantalla y el programa se acabará.

Versión 2

Modificar el programa anterior para que, una vez que la bola y la pulga hayan aparecido en la pantalla, se llame dos veces por segundo a la función `salta_pulga`. Esta función, que ya está programada, hace saltar la pulga de forma aleatoria en un radio de 5 posiciones a su alrededor, controlando que la pulga no salga del margen. Como argumento recibe por referencia la posición de la pulga antes de realizar el salto y la actualiza con la nueva posición. El programa acaba a los 10 segundos de haber comenzado.

Versión 3

En esta versión se debe controlar el movimiento de la bola mediante los cursores del teclado. Cada vez que alguien pulsa una de estas teclas ("make" de las teclas) la bola se mueve una posición en la dirección que corresponda. El programa ha de controlar que la bola no salga del margen. El programa acaba cuando se pulse y suelte la tecla ESC (detectar el "break" de la tecla) o cuando la bola atrape la pulga, pero no por tocar uno de los márgenes. Se recomienda mantener la condición de finalizar a los 10 segundos hasta asegurarse de que el ESC funciona correctamente. Para detectar si la bola ha atrapado la pulga o que la bola no pise el margen se ha de utilizar la función de `leer_pantalla`.