

(3)



# Unidad 3

## Interfaz del ensamblador con el lenguaje C

***SISTEMAS BASADOS EN MICROPROCESADORES***

**Grado en Ingeniería Informática  
Doble Grado en Ingeniería Informática y Matemáticas**

**EPS - UAM**

(3)

# Índice

## 3. Interfaz del ensamblador con el lenguaje C.

- 3.1. Características generales.
- 3.2. El ejemplo del lenguaje C.
- 3.3. Los distintos modelos del lenguaje C.
- 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados.

(3)

## 3.1. Características generales

- Muchas aplicaciones escritas en lenguajes de alto nivel requieren partes críticas escritas en ensamblador (ejecución en tiempo real, uso de instrucciones multimedia tipo MMX, etc.)
- Implica poder llamar desde programas escritos en lenguajes de alto nivel compilables a rutinas escritas en ensamblador.
- También es posible llamar desde programas escritos en ensamblador a rutinas escritas en lenguajes de alto nivel compilables.
- Factible si programas de ensamblador siguen las convenciones (nomenclatura, paso de parámetros y resultados, ...) de los lenguajes de alto nivel.

(3)

## 3.2. El ejemplo del lenguaje C (I)

- La mayoría de aplicaciones que requieren interactuar con ensamblador escritas en lenguaje C (y C++).
- El lenguaje C tiene construcciones típicas de alto nivel (bucles, tipos estructurados, recursividad, ...), pero también permite un control a muy bajo nivel (acceso a puertos de E/S, manipulación de bits, ...).
- Los compiladores de C permiten el enlace con programas escritos en ensamblador sólo si siguen las **mismas convenciones** aplicadas por el compilador.
- Programa en C se compila a código objeto, programa en ensamblador se ensambla a código objeto y montador (*linker*) del compilador de C genera el ejecutable enlazando esos ficheros objeto.

(3)

## 3.2. El ejemplo del lenguaje C (II)

- Convenciones del lenguaje C relacionadas con:
  - Uso de direcciones cortas (*near*) o largas (*far*) para acceder a datos (variables) y/o procedimientos: modelo de memoria.
  - Nomenclatura de segmentos, variables y procedimientos.
  - Paso de parámetros a procedimientos y devolución de resultados.

(3)

### 3.3. Los distintos modelos del lenguaje C (I)

- Cuando se compila un programa en C se debe escoger un modelo de memoria.
- Cada modelo determina la ubicación en memoria de los segmentos lógicos (código, datos y pila) y si se usan direcciones cortas o largas para acceder a ellos.
- Seis modelos de memoria en Turbo C:
  - TINY
  - SMALL
  - MEDIUM
  - COMPACT
  - LARGE
  - HUGE

(3)

## 3.3. Los distintos modelos del lenguaje C (II)

### • TINY

- Mínima ocupación de memoria.
- Los cuatro registros de segmento (**CS**, **SS**, **DS** y **ES**) son idénticos. El programa ocupa hasta 64 KB.
- Código, datos y pila en el mismo segmento físico.
- Programas compilados en este modelo pueden convertirse a .COM (*drivers*) mediante la utilidad EXE2BIN de DOS o usando la opción /t del montador.
- Punteros cortos (*near*) para código y datos.

### • SMALL

- Programas pequeños en los que no es necesario mínima ocupación de memoria.
- Un segmento físico para código (hasta 64 KB) y otro para datos y pila (hasta 64 KB).
- Punteros cortos (*near*) para código y datos.

(3)

## 3.3. Los distintos modelos del lenguaje C (III)

### • MEDIUM

- Programas grandes que usan pocos datos.
- Varios segmentos físicos para código (hasta 1 MB) y uno para datos y pila (hasta 64 KB).
- Punteros largos (*far*) para código y cortos (*near*) para datos.

### • COMPACT

- Programas pequeños que usan muchos datos.
- Un segmento físico para código (hasta 64 KB) y varios para datos y pila (hasta 1 MB).
- Punteros cortos (*near*) para código y largos (*far*) para datos.

### • LARGE

- Programas grandes que usan muchos datos.
- Varios segmentos físicos para código (hasta 1 MB) y para datos y pila (hasta 1 MB). En total no puede superarse 1MB.
- Punteros largos (*far*) para código y datos.



(3)

## 3.3. Los distintos modelos del lenguaje C (IV)

### • HUGE

- Similar al LARGE con algunas ventajas y desventajas.
- Punteros normalizados (offset < 16).
- Variables globales estáticas pueden superar 64 KB (posible manipular bloques de datos de más de 64 KB).
- Compilador inserta código que actualiza automáticamente registros de segmento de datos (punteros a datos siempre normalizados).
- Modelo más costoso en tiempo de ejecución.

(3)

### 3.3. Los distintos modelos del lenguaje C (V)

Modelo	Segmentos			Punteros	
	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 de más de 64 KB)		FAR	FAR

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (I)

### Nomenclatura

- El compilador de C siempre nombra de igual forma los segmentos lógicos que utiliza:
  - El segmento de código se llama **\_TEXT**.
  - El segmento **\_DATA** contiene las variables globales inicializadas.
  - El segmento **\_BSS** contiene las variables globales NO inicializadas.
  - El segmento de pila lo define e inicializa el compilador de C en la función *main*.
  - En los modelos pequeños de datos (*tiny*, *small* y *medium*), todos los segmentos de datos están agrupados con el nombre **DGROUP**:

**DGROUP    GROUP    \_DATA, \_BSS**

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (II)

### Nomenclatura

- El compilador de C añade un **\_** delante de todos los nombres de variables y procedimientos:

- Ejemplo:

```
int a = 12345;  
char b = 'A';  
char c[] = "Hola mundo";  
int d = 12;
```

- Se compila como:

```
_DATA SEGMENT WORD PUBLIC 'DATA'  
PUBLIC _a, _b, _c, _d  
_a DW 12345  
_b DB 'A'  
_c DB "Hola mundo", 0  
_d DW 12  
_DATA ENDS
```

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (III)

### Nomenclatura

- El compilador de C añade un `_` delante de todos los nombres de variables y procedimientos:

- Ejemplo:

```
main()
{
    funcion();
}
```

- Se compila como:

```
_TEXT SEGMENT BYTE PUBLIC 'CODE'

_main PROC FAR
    CALL _funcion
    RET
_main ENDP

_TEXT ENDS
```

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (IV)

### Nomenclatura

- Las variables y procedimientos de ensamblador que sean accedidos desde programas en C deben llevar un `_` por delante que no aparece en C.
- El lenguaje C distingue entre mayúsculas y minúsculas: `funcion()` y `FUNCION()` son procedimientos distintos.
- Es necesario que el ensamblador también distinga entre mayúsculas y minúsculas.
- En TASM se consigue ensamblando con las opciones `/mx` (fuerza distinción para símbolos públicos) o `/ml` (fuerza distinción para todos los símbolos).

(3)

### 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (V)

#### Paso de parámetros

- En lenguaje C, un procedimiento que llama a otro apila sus parámetros antes de ejecutar el **CALL**.
- Los procedimientos de ensamblador que llamen a funciones de C también han de apilar sus parámetros.
- Los parámetros se apilan en orden inverso a como aparecen en la llamada de C: empezando por el último y acabando por el primero.
- Tras retornar de la subrutina, se extraen de la pila los parámetros sumando al registro **SP** el tamaño en bytes de los parámetros.
- Los parámetros de un byte (char) se apilan con dos bytes (el más significativo vale 0).

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (VI)

### Paso de parámetros

- Los parámetros se apilan en formato *little endian*: palabra menos significativa en dirección menor y byte menos significativo en dirección menor.
- Para pasar por parámetro punteros a funciones o a datos, es necesario saber en qué modelo de memoria se está compilando el programa en C, para apilar el registro de segmento (modelo largo) o no apilarlo (modelo corto).



(3)

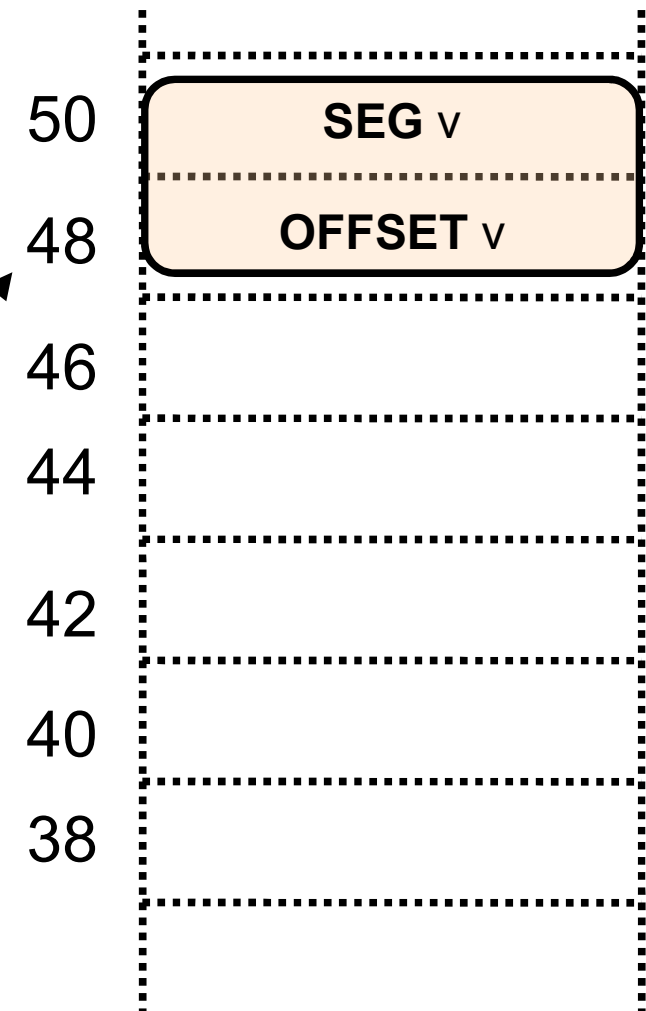
## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (VII)

### Paso de parámetros (ejemplo)

- Prototipo de la función:  
`void funcion ( char, long int, void * );`

- Llamada en modelo largo  
(punteros **FAR** para datos y código)

`funcion ( 'P', 0x23, &v );`

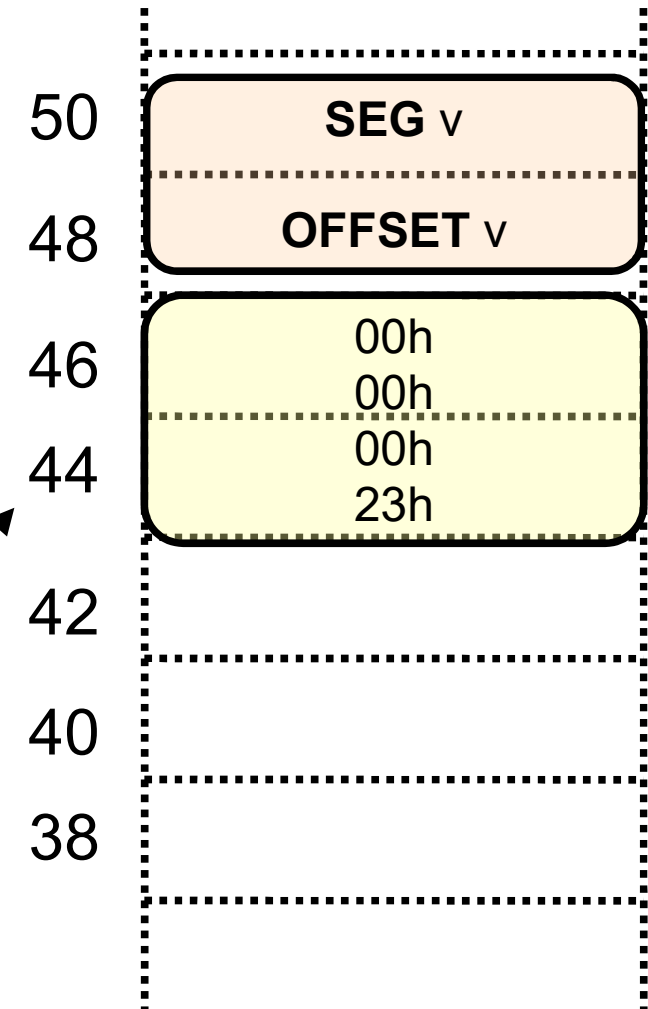


(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (VIII)

### Paso de parámetros (ejemplo)

- Prototipo de la función:  
`void funcion ( char, long int, void * );`
- Llamada en modelo largo  
(punteros **FAR** para datos y código)  
`funcion ( 'P', 0x23, &v );`

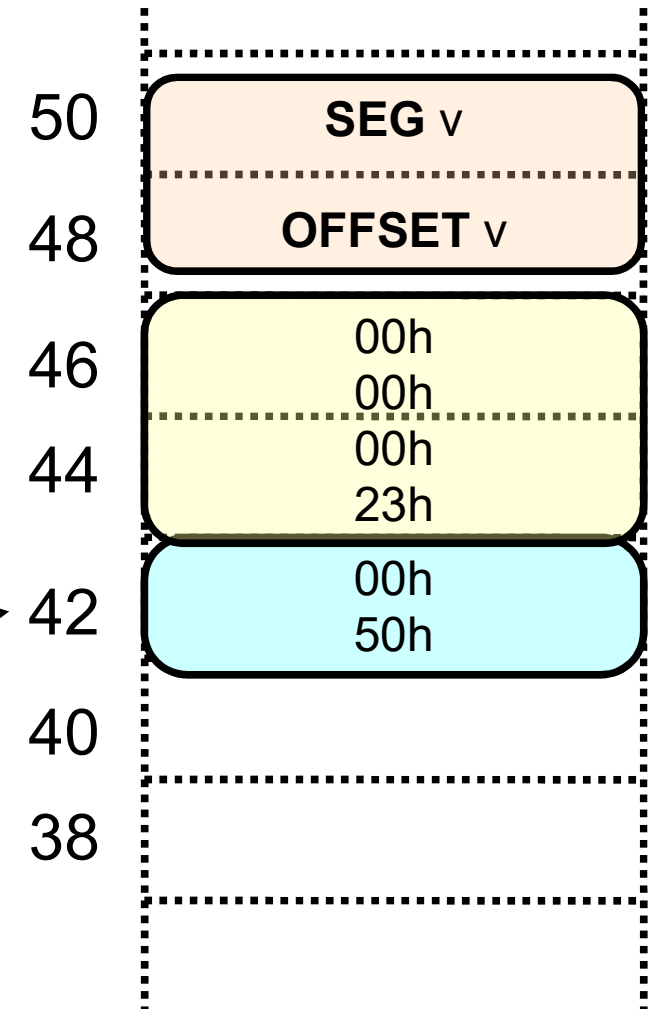


(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (IX)

### Paso de parámetros (ejemplo)

- Prototipo de la función:  
`void funcion ( char, long int, void * );`
- Llamada en modelo largo  
(punteros **FAR** para datos y código)  
`funcion ( 'P', 0x23, &v );`



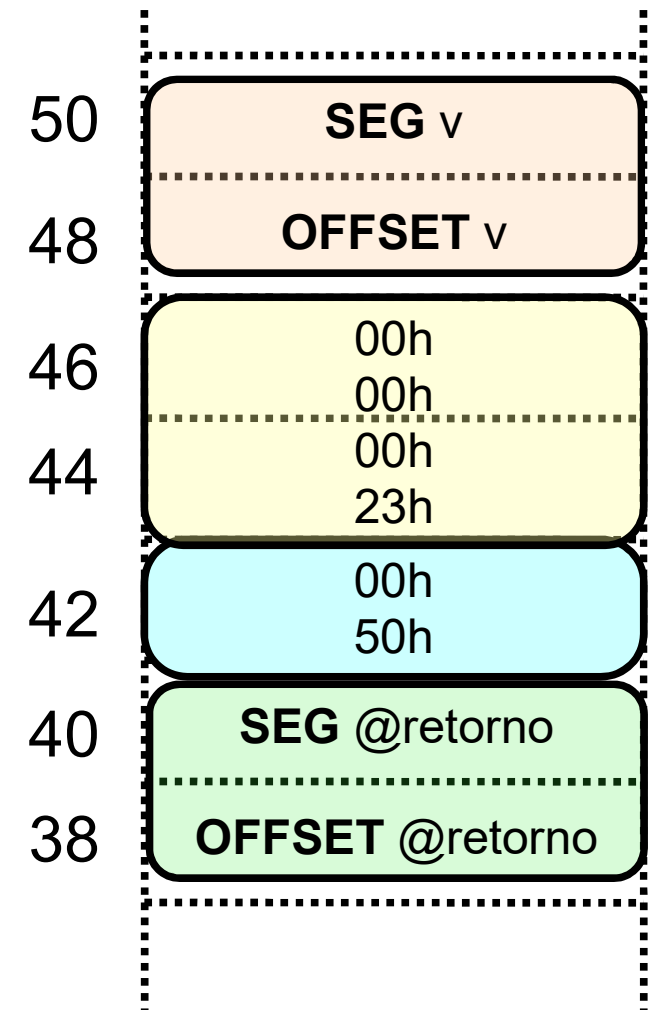
(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (X)

### Paso de parámetros (ejemplo)

- Prototipo de la función:  
`void funcion ( char, long int, void * );`
- Llamada en modelo largo  
(punteros **FAR** para datos y código)  
`funcion ( 'P', 0x23, &v );`

`call _funcion`

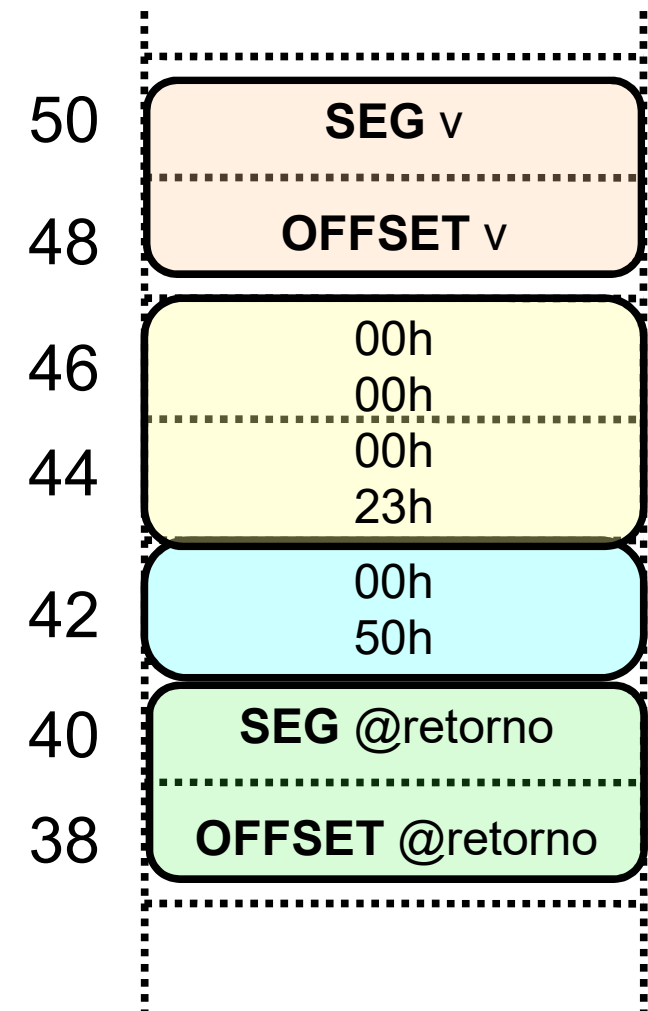
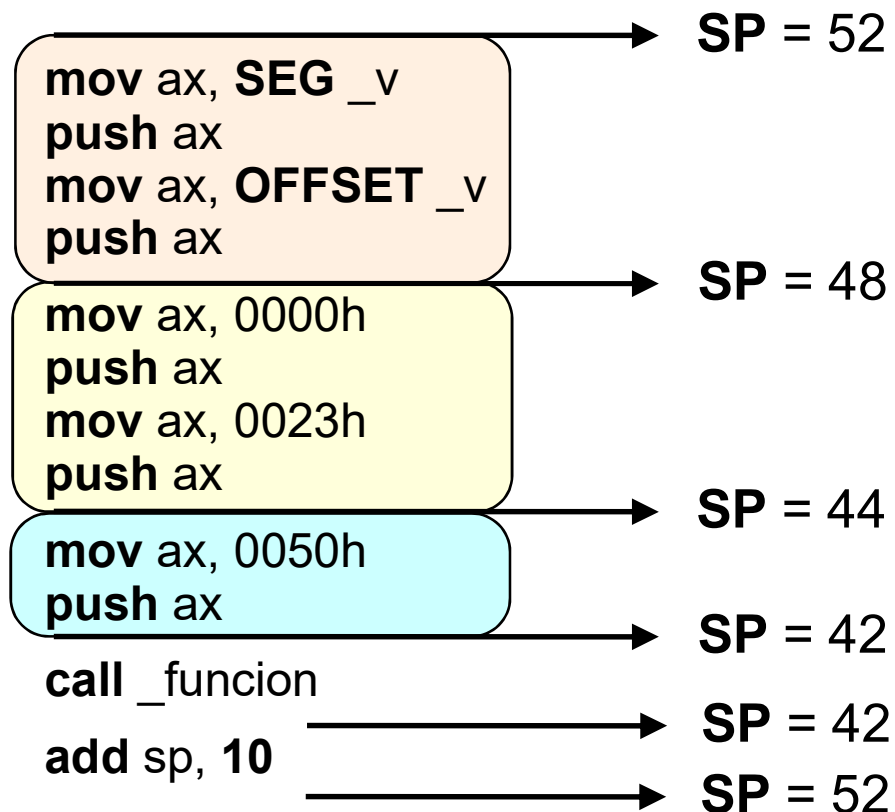


(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XI)

### Paso de parámetros (ejemplo)

```
void funcion ( char, long int, void * );  
funcion ( 'P', 0x23, &v );
```



(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XII)

### Acceso a parámetros (ejemplo)

```
void funcion ( char, long int, void * );
```

```
_funcion PROC FAR
```

```
  push bp
```

```
  mov bp, sp
```

bx := **OFFSET** v

es := **SEG** v

```
  les bx, [ bp + 12 ]
```

```
  mov dx, [ bp + 10 ]
```

```
  mov ax, [ bp + 8 ]
```

```
  mov cx, [ bp + 6 ]
```

```
  ...
```

```
  ret
```

```
_funcion ENDP
```

**SP = BP**

50

**SEG v**

48

**OFFSET v**

46

00h

00h

44

00h

23h

42

00h

50h

40

**SEG @retorno**

38

**OFFSET @retorno**

36

**BP inicial**

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XIII)

### Devolución de resultados

- Las variables de retorno de una función con una longitud de 16 bits se devuelven al procedimiento llamante en **AX** y las de 32 bits en **DX:AX**.

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XIV)

### Ejemplo 1

*/\* Variable visible (externa) desde la rutina de ensamblador \*/*

**int** variable\_c;

*/\* Variable definida como pública en ensamblador \*/*

**extern int** dato\_as;

*/\* Declaración de la función de ensamblador (podría estar en un **include**) \*/*

**int** funcion ( **int** a, **char far** \*p, **char** b ) ;

main()

{

**int** a = 123;

*/\* Declaración de variables locales de C \*/*

**char** b = 'F';

**char far** \*p;

*/\* Llama a la función y guarda en variable\_c el valor devuelto en AX \*/*

variable\_c = funcion ( a, p, b );

}



(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XV)

```
DGROUP GROUP _DATA, _BSS
```

; Se agrupan segmentos de datos en uno

```
_DATA SEGMENT WORD PUBLIC 'DATA'
```

```
    PUBLIC _dato_as
```

```
    _dato_as DW 0
```

```
_DATA ENDS
```

; Segmento de datos público

; Declaración de \_dato\_as como público

; Reserva de \_dato\_as e inicialización

```
_BSS SEGMENT WORD PUBLIC 'BSS'
```

```
    EXTRN _variable_c : WORD
```

```
    _sin_as DW ?
```

```
_BSS ENDS
```

; Segmento de datos público

; Declaración de \_variable\_c como externa y de

; tipo WORD (definida en el módulo C)

; Variable no accesible por el módulo C

```
_TEXT SEGMENT BYTE PUBLIC 'CODE'
```

```
    ASSUME CS:_TEXT, DS:DGROUP
```

```
    PUBLIC _funcion
```

```
    _funcion PROC NEAR
```

```
        PUSH BP
```

```
        MOV BP, SP
```

```
        MOV BX, [ BP+4 ]
```

```
        LDS SI, [ BP+6 ]
```

```
        MOV CX, [ BP+10 ]
```

```
        ...
```

```
        MOV AX, CX
```

```
        POP BP
```

```
        RET
```

```
    _funcion ENDP
```

```
_TEXT ENDS
```

```
END
```

; Definición del segmento de código

; Hace accesible a funcion desde C

; En C es funcion ()

; Para poder utilizar BP para direccionar la pila

; se carga con puntero a cima de pila

; Guarda a en BX , BX=123

; Guarda p en DS:SI

; Guarda b en CX, CL='F', CH =0

; Como función es int el valor se devuelve en AX

; Restaura BP

; Retorna a procedimiento llamante

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XVI)

### Ejemplo 2

`char * strchr (char *string, int character);`

datos **SEGMENT**

micadena **DB** "Esto es una cadena ASCIIZ", 0

datos **ENDS**

codigo **SEGMENT**

**mov** ax,'a' /\* Se apila carácter buscado \*/  
**push** ax

**mov** ax, **SEG** micadena /\* Se apila cadena de caracteres \*/  
**push** ax

**mov** ax, **OFFSET** micadena  
**push** ax

**call** **FAR** \_strchr /\* Llamada al procedimiento \*/  
**add** sp, 6 /\* Equilibrado de la pila \*/

**mov** ds, dx /\* Puntero retornado en **DX:AX** \*/  
**mov** si, ax

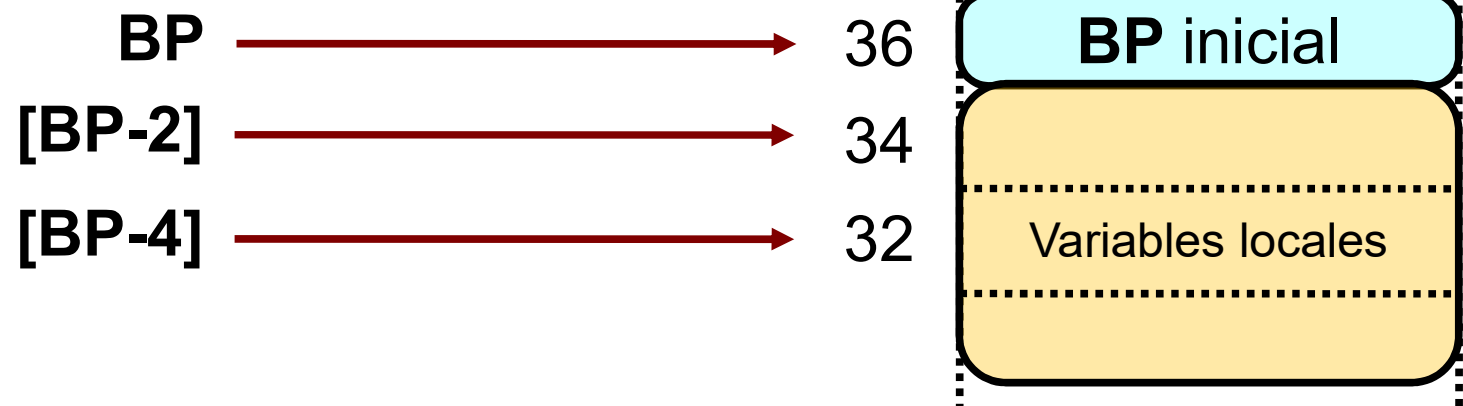
codigo **ENDS**

(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XVII)

### Definición de variables locales

- Las rutinas de C definen las variables locales que necesitan en la pila, por encima de BP.
- Se introducen en mismo orden en que están declaradas.
- Se acceden mediante [BP-2], [BP-4], ....



(3)

## 3.4. Convenios de nomenclatura, paso de parámetros, devolución de resultados (XVIII)

- La instrucción `asm` permite insertar ensamblador en el programa en C (ensamblador *inline*).

```
main
{
    int d1 = 5, d2 = 4, resultado;
    asm {
        push cx
        push ax
        mov ax, 0
        mov cx, d2
        cmp cx, 0
        jz final
    }
    mult:
    asm {
        add ax, d1
        dec cx
        jnz mult
    }
    final:
    mov resultado, ax
    pop ax
    pop cx
}
printf ("resultado %d\n", resultado); }
```

No se debe usar **BP**, ya que lo utiliza el compilador para acceder a las variables locales.