(3)

UAM
Universidad Autónoma
de Madrid

Unit 3
Interface Between Assembly and C Languages

MICROPROCESSOR-BASED SYSTEMS

Degree in Computer Science Engineering
Double Degree in Computer Engineering and Mathematics

EPS - UAM

Index

3.1. General features

- Many applications written in high-level languages require critical parts written in assembly language (real time execution, use of multimedia instructions such as MMX, etc.)
- Need to be able to call assembly routines from programs written in compatible high-level languages.
- Also possible to call routines written in high-level compilable languages from assembly routines.
- Feasible if assembly programs follow the conventions (nomenclature, passing of parameters and results, …) of the high-level languages.

3.2. The example of C language (I)

- Most applications requiring interaction with assembly language are written in C (and C++).
- C language has typical high-level constructs (loops, structured types, recursion, …), but also allows control at a very low level (access to I/O ports, bit manipulation, …).
- C compilers allow linkage with assembly programs only if they follow the **same conventions** applied by the compiler.
- C program compiled into object file, assembly program assembled into object file and linker of the C compiler generates executable file by linking the object files.

3.2. The example of C language (II)

- C language conventions related to:
  - Use of near or far addresses for accessing data (variables) and/or procedures: memory model.
  - Nomenclature of segments, variables and procedures.
  - Parameter passing to procedures and return of results.

3.3. The different models of C language (I)

- A memory model must be chosen when a C program is compiled.
- Every model determines the memory location of the logic segments (code, data and stack) and if near or far addresses are used for accessing them.
- Six memory models in Turbo C:
  - TINY
  - SMALL
  - MEDIUM
  - COMPACT
  - LARGE
  - HUGE

3.3. The different models of C language (II)

- TINY
  - Minimum memory occupation.
  - The four segment registers (**CS**, **SS**, **DS** y **ES**) are identical.
  - The program occupies up to 64 KB.
  - Code, data and stack on the same physical segment.
  - Programs compiled in this model can be converted to .COM (*drivers*) through the EXE2BIN utility of DOS or using the linker option /t.
  - Near pointers for code and data.

- SMALL
  - Small programs in which minimum memory occupation is not necessary.
  - A physical segment for code (up to 64 KB) and another segment for data and stack (up to 64 KB).
  - Near pointers for code and data.

3.3. The different models of C language (III)

- MEDIUM
  - Big programs that use few data.
  - Several physical segments for code (up to 1 MB) and a segment for data and stack (up to 64 KB).
  - Far pointers for code and near pointers for data.
- COMPACT
  - Small programs that use many data.
  - A physical segment for code (up to 64 KB) and several segments for data and stack (up to 1 MB).
  - Near pointers for code and far pointers for data.
- LARGE
  - Big programs that use many data.
  - Several physical segments for code (up to 1 MB) and for data and stack (up to 1 MB). Not possible to exceed 1 MB in total.
  - Far pointers for code and data.

3.3. The different models of C language (IV)

- HUGE
  - Similar to LARGE with some advantages and disadvantages.
  - Normalized pointers (offset < 16).
  - Static global variables can exceed 64 KB (possible to manipulate data blocks larger than 64 KB).
  - Compiler inserts code that automatically updates data segment registers (data pointers always normalized).
  - Most costly model in terms of execution time.

3.3. The different models of C language (V)

Model	Segments			Pointers	
	Code	Data	Stack	Code	Data
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 (blocks larger than 64 KB)		FAR	FAR

3.4. Conventions on nomenclature, parameter passing and return of results (I)

Nomenclature

- The C compiler always gives the same name to the logical segments it uses:
  - The code segment is called **_TEXT**.
  - Segment **_DATA** contains the initialized global variables.
  - Segment **_BSS** contains the uninitialized global variables.
  - The stack segment is defined and initialized by the C compiler in the *main* function.
  - In the small data models (*tiny*, *small* and *medium*), all data segments are grouped under the name **DGROUP**:

DGROUP **GROUP** _DATA, _BSS

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (II)

Nomenclature

- The C compiler adds _ right before all names of variables and procedures:
  - Example:

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

  - It is compiled as:

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

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (III)

Nomenclature

- The C compiler adds _ right before all names of variables and procedures:
  - Example:

```
main()
{
        function();
}
```

  - It is compiled as:

```
_TEXT SEGMENT BYTE PUBLIC 'CODE'

_main PROC FAR
      CALL _function
      RET
_main ENDP

_TEXT ENDS
```

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (IV)

Nomenclature

- The assembly variables and procedures accessed from C programs must be preceded by _ , which does not appear in C.
- The C language distinguishes between lowercase and uppercase: function() and FUNCTION() are different procedures.
- It is necessary that the assembler also distinguishes between lowercase and uppercase.
- In TASM, this is done by assembling with options /mx (it forces distinction for public symbols) or /ml (it forces distinction for all symbols).

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (V)

Parameter passing

- In C language, a procedure that calls another stacks its parameters before executing **CALL**.
- The assembly procedures that call C functions must also stack their parameters.
- Parameters are stacked in reversed order with respect to their position in the C call: starting with the last and ending with the first.
- After returning from the subroutine, parameters are extracted from the stack by adding their size in bytes to register **SP**.
- Single-byte parameters (char) are stacked using two bytes (the most significant byte is 0).

3.4. Conventions on nomenclature, parameter passing and return of results (VI)

Parameter passing

- Parameters are stacked in *little endian* format: least significant word in lowest address and least significant byte in lowest address.
- In order to pass pointers to functions or data, it is necessary to know the memory model in which the C program is being compiled, so as to stack the segment register (far model) or not (near model).

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (VII)

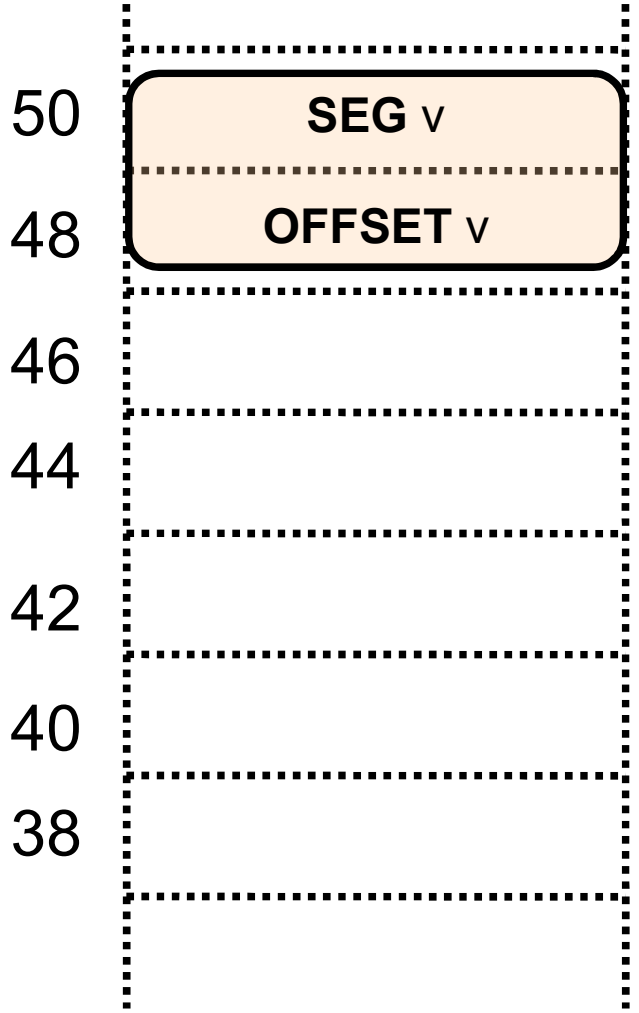
Parameter passing (example)

- Function prototype:

  void function (**char**, **long int**, **void** *);

- Call in far model (**FAR** pointers for data and code)

  function ('P', 0x23, &v);



3.4. Conventions on nomenclature, parameter passing and return of results (VIII)

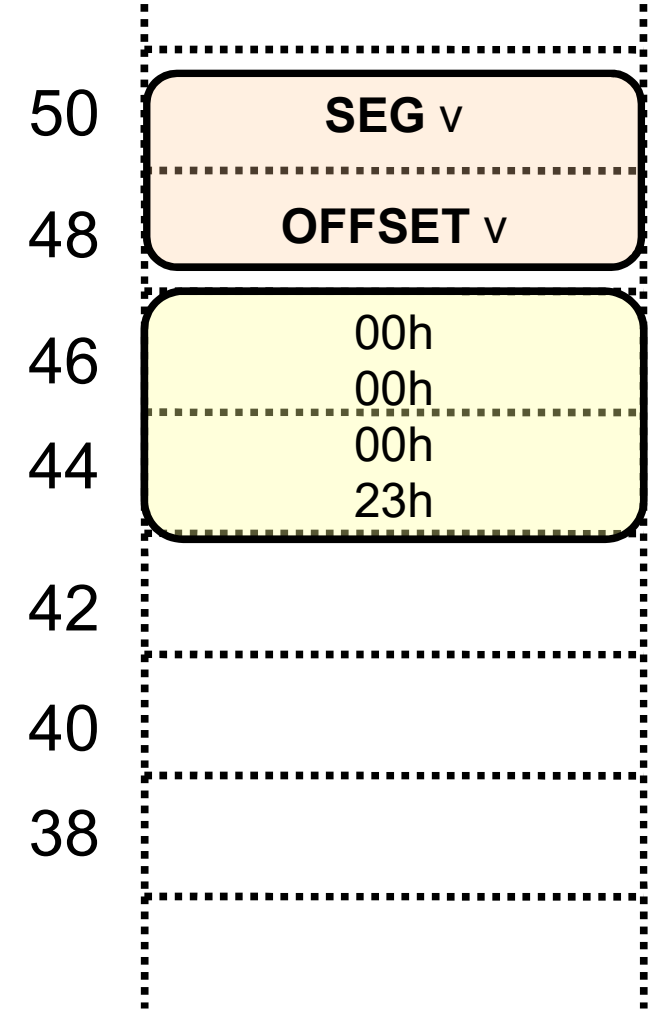
Parameter passing (example)

- Function prototype:

  void function (**char**, **long int**, **void** *);

- Call in far model (**FAR** pointers for data and code)

  function ('P', 0x23, &v);



3.4. Conventions on nomenclature, parameter passing and return of results (IX)

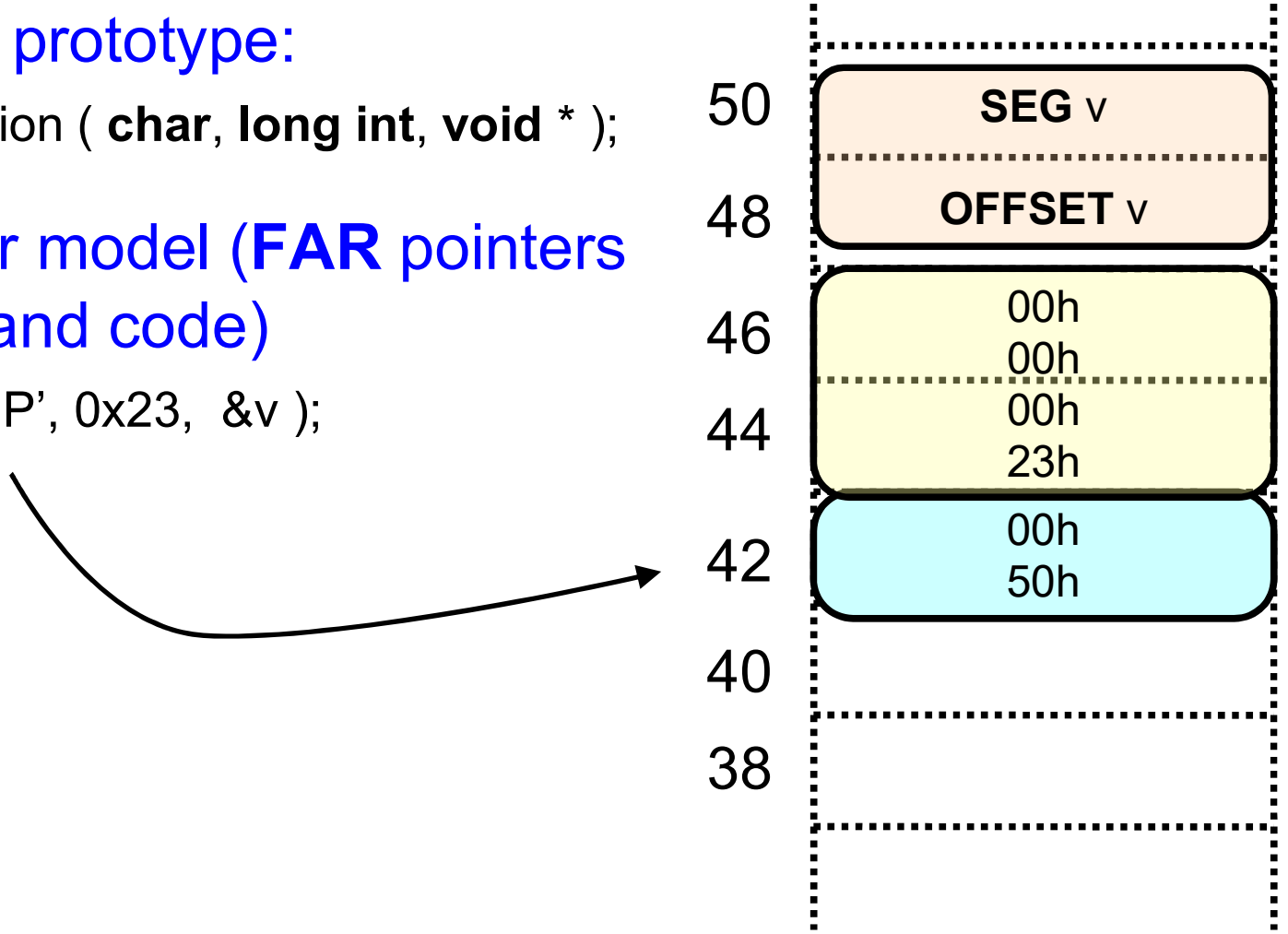
Parameter passing (example)

- Function prototype:

  void function (**char**, **long int**, **void** *);

- Call in far model (**FAR** pointers for data and code)

  function ('P', 0x23, &v);



(3)

3.4. Conventions on nomenclature, parameter passing and return of results (X)

Parameter passing (example)

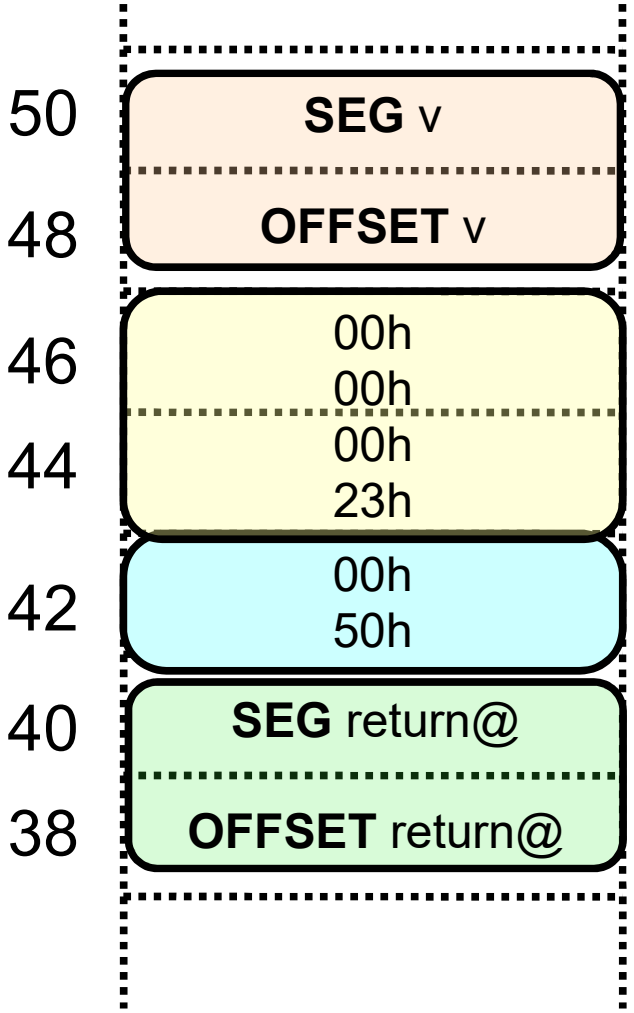
- Function prototype:

  void function (**char**, **long int**, **void** *);

- Call in far model (**FAR** pointers for data and code)

  function ('P', 0x23, &v);

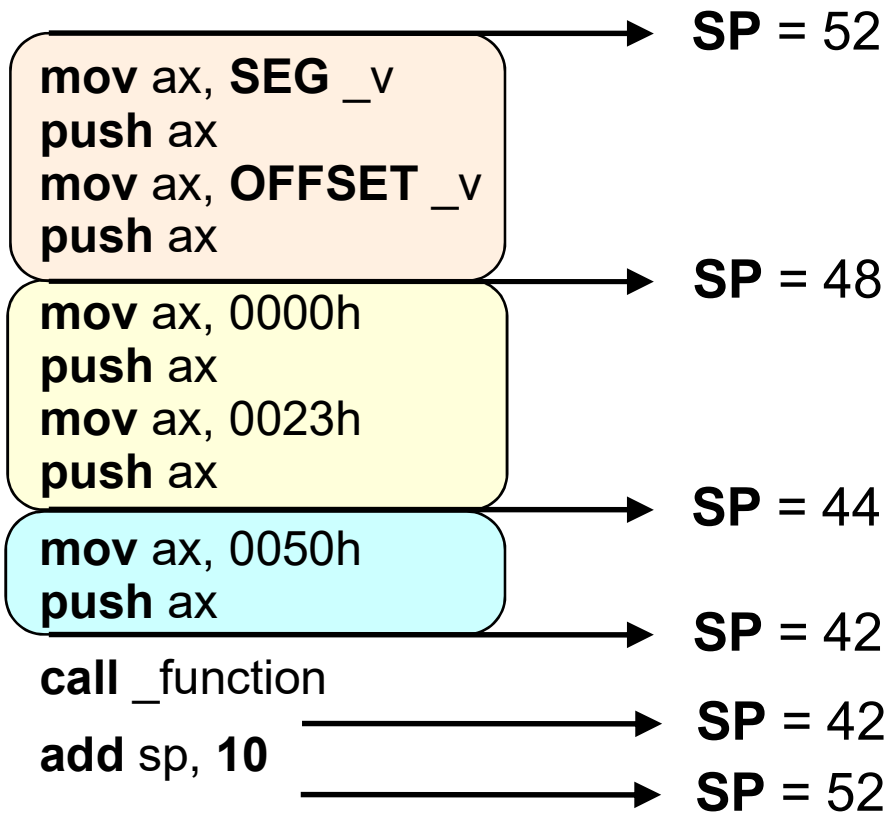
call _function

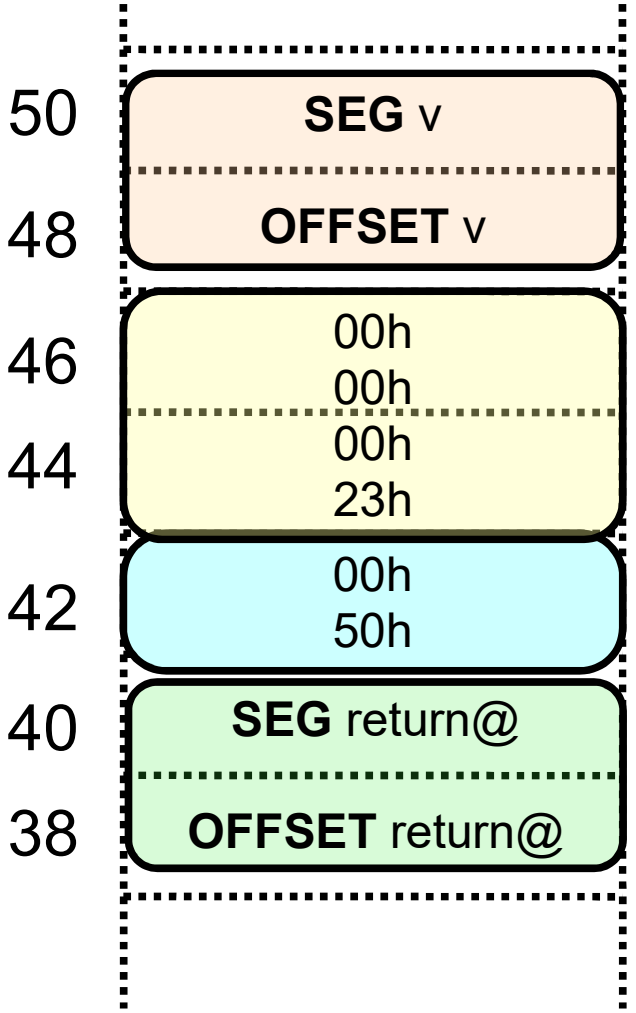


3.4. Conventions on nomenclature, parameter passing and return of results (XI)

Parameter passing (example)

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





3.4. Conventions on nomenclature, parameter passing and return of results (XII)

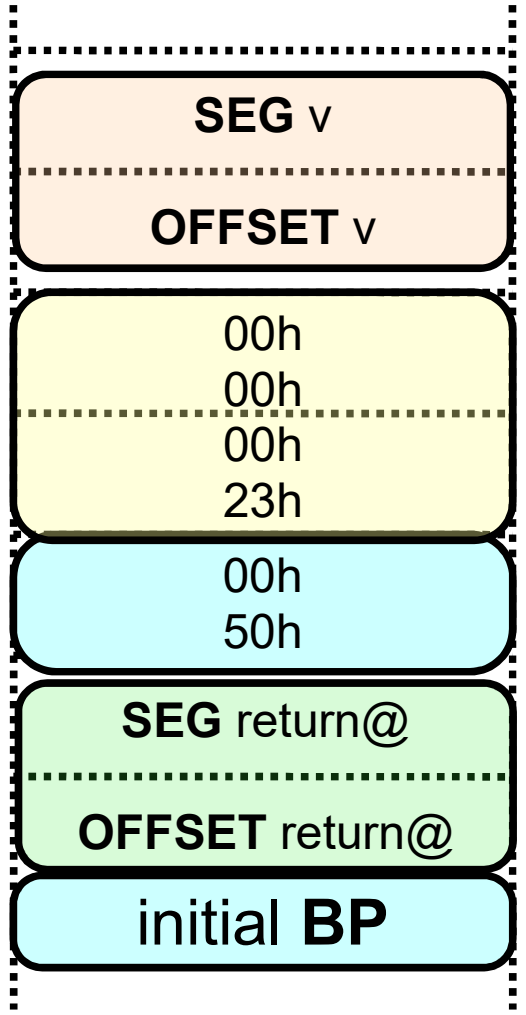
Parameter passing (example)

void function (**char**, **long int**, **void** *);

```
_function PROC FAR
        push bp
        mov bp, sp

        les bx, [ bp + 12 ]
        mov dx, [ bp + 10 ]
        mov ax, [ bp + 8 ]
        mov cx, [ bp + 6 ]
        …

        ret
_function ENDP
```

bx := **OFFSET** v
es := **SEG** v

SP = BP



3.4. Conventions on nomenclature, parameter passing and return of results (XIII)

Return of results

- The return variables of the function with length of 16 bits are returned to the calling procedure through **AX** and the ones of 32 bits through **DX:AX**.

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (XIV)

Example 1

```
/* Visible variable (external) from the assembly routine */
int variable_c;

/* Variable defined as public in the assembly program */
extern int data_as;

/* Declaration of the assembly function (it could be within an include) */
int function ( int a, char far *p, char b ) ;

main()
{
    int a = 123;              /* Declaration of C local variables */
    char b = 'F';
    char far *p;

/* Call the function and store in variable_c the value returned through AX */
    variable_c = function ( a, p, b );
}
```

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (XV)

```
DGROUP GROUP _DATA, _BSS                ; Data segments are grouped
_DATA SEGMENT WORD PUBLIC 'DATA'        ; Public data segment
    PUBLIC _data_as                     ; Declaration of _data_as as public
    _data_as    DW      0               ; Allocation of _data_as and initialization
_DATA ENDS

_BSS SEGMENT WORD PUBLIC 'BSS'          ; Public data segment
    EXTRN _variable_c : WORD            ; Declaration of _variable_c as external and of
                                        ; type WORD (defined in the C module)
    _hidden_as  DW      ?               ; Variable not accessible from the C module
_BSS ENDS

_TEXT SEGMENT BYTE PUBLIC 'CODE'        ; Definition of the code segment
    ASSUME CS:_TEXT, DS:DGROUP
PUBLIC _function                        ; Make function accessible from C
_function  PROC NEAR                    ; It is function() in C
    PUSH BP                             ; For using BP in order to address the stack,
    MOV BP, SP                          ; it is loaded with pointer to the top
    MOV BX, [ BP+4 ]                    ; Store a in BX , BX=123
    LDS SI, [ BP+6 ]                    ; Store p in DS:SI
    MOV CX, [ BP+10 ]                   ; Store b in CX, CL='F', CH =0
    ...

    MOV AX, CX                          ; Value returned through AX since function is int
    POP         BP                      ; Restore BP
    RET                                 ; Return to calling procedure
_function ENDP
_TEXT ENDS
END
```

3.4. Conventions on nomenclature, parameter passing and return of results (XVI)

Example 2

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

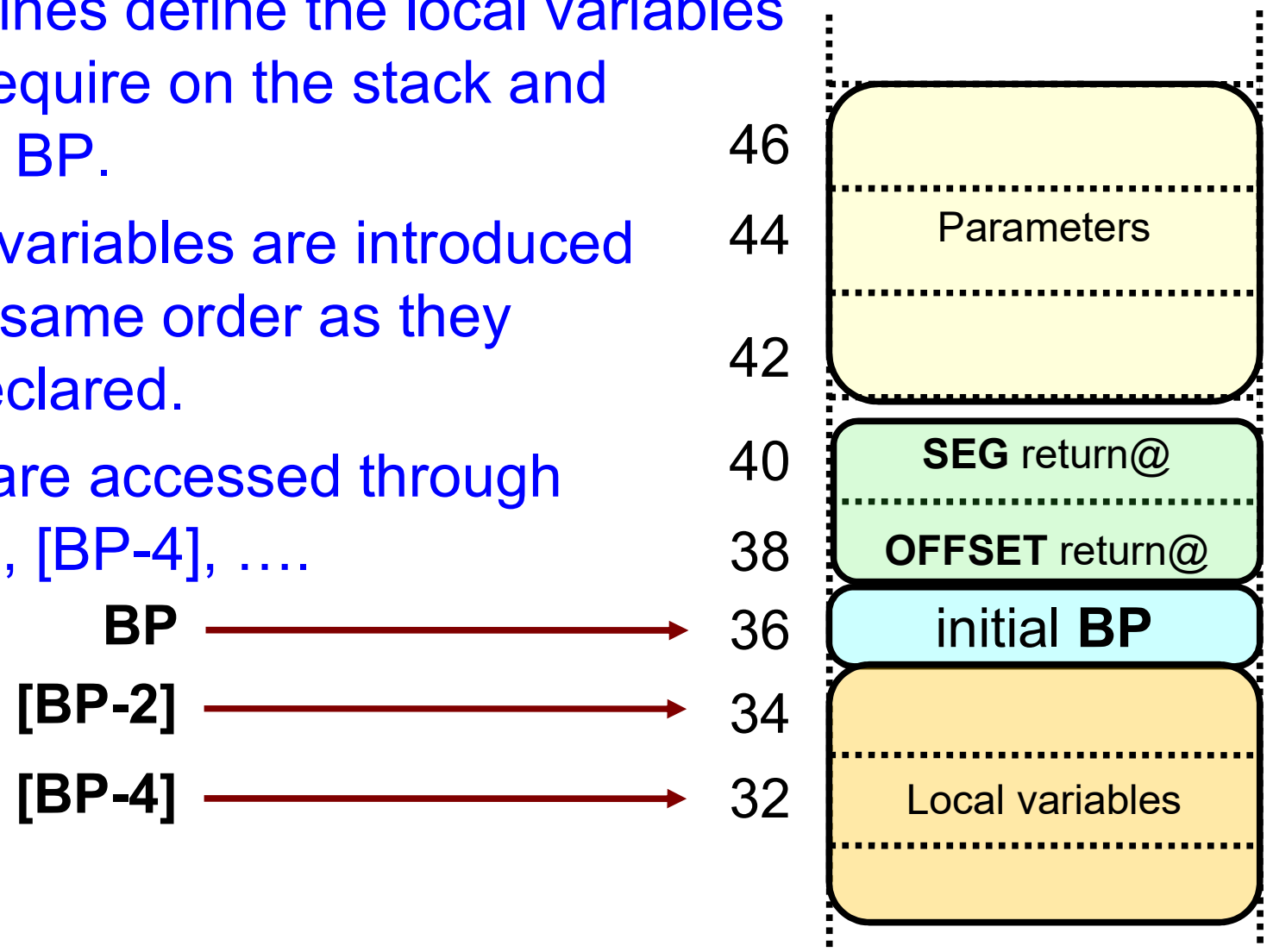
```
data SEGMENT
   mystring DB "This is an ASCIIZ string", 0
data ENDS

code SEGMENT

   mov ax,'a'                    /* Sought character is stacked */
   push ax

   mov ax, SEG mystring          /* String is stacked */
   push ax
   mov ax, OFFSET mystring
   push ax

   call FAR _strchr              /* Procedure call     */
   add sp, 6                     /* Stack balancing  */

   mov ds, dx                    /* Pointer returned through DX:AX   */
   mov si, ax

code ENDS
```

(3)

3.4. Conventions on nomenclature, parameter passing and return of results (XVII)

Definition of local variables

- C routines define the local variables they require on the stack and above BP.
- Local variables are introduced in the same order as they are declared.
- They are accessed through [BP-2], [BP-4], ….



3.4. Conventions on nomenclature, parameter passing and return of results (XVIII)

- Instruction asm allows inserting assembly code within a C program (*inline* assembly).

```
main
{
        int d1 = 5, d2 = 4, result;
        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 result, ax
                pop ax
                pop cx
        }
        printf ("result %d\n", result); }
```

BP must not be used, since it is used by the compiler to access local variables.