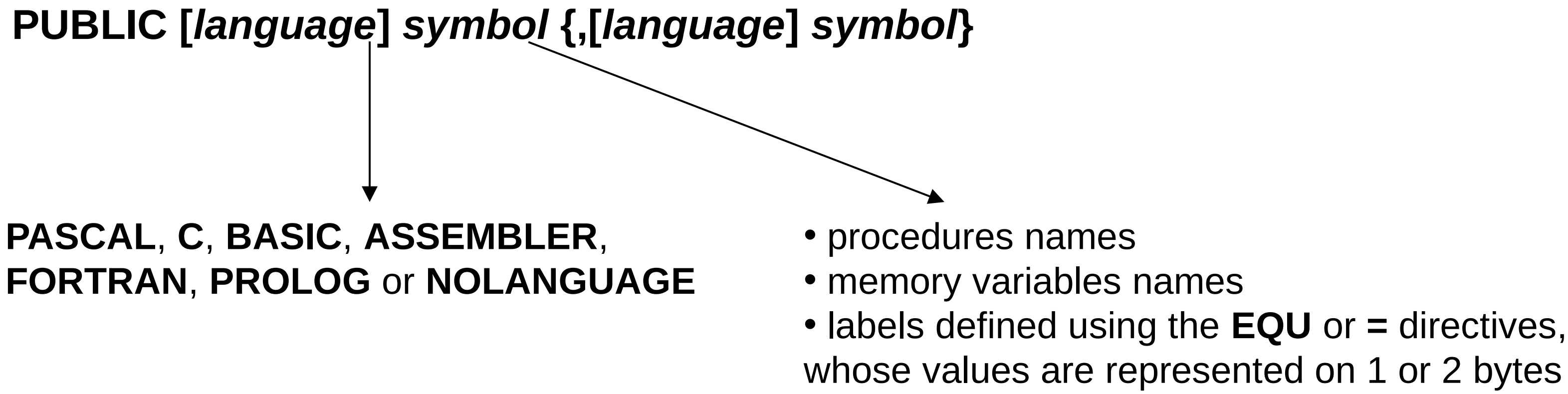
Multi-module programming

Requirements of an assembly language module when it is linked with another module

- <u>**PUBLIC directive**</u>

  - it exports to other modules the symbols defined in the current assembly language module



PUBLIC [*language*] *symbol* {,[*language*] *symbol*}

PASCAL, **C**, **BASIC**, **ASSEMBLER**, **FORTRAN**, **PROLOG** or **NOLANGUAGE**

• procedures names
• memory variables names
• labels defined using the **EQU** or **=** directives, whose values are represented on 1 or 2 bytes

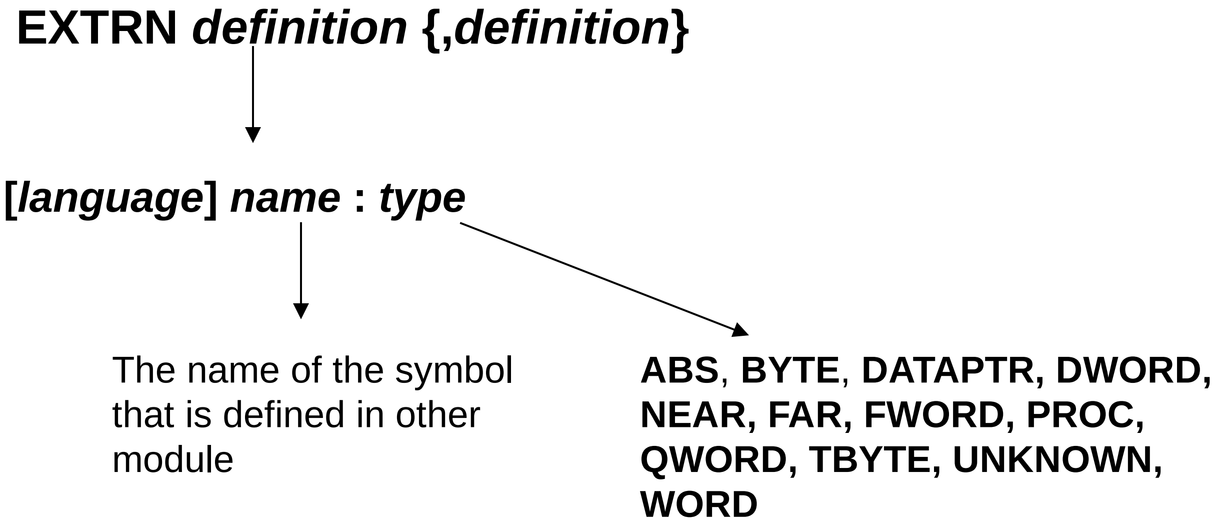
<u>*Ex:*</u> **PUBLIC C ProcA**

- It imposes to export the *ProcA* symbol to the other modules as _*ProcA*, according to the rules of the C language.

Requirements of an assembly language module when it is linked with another module

- **EXTRN directive**

- it makes visible in the current module the symbols defined in other modules



EXTRN *definition* {,*definition*}

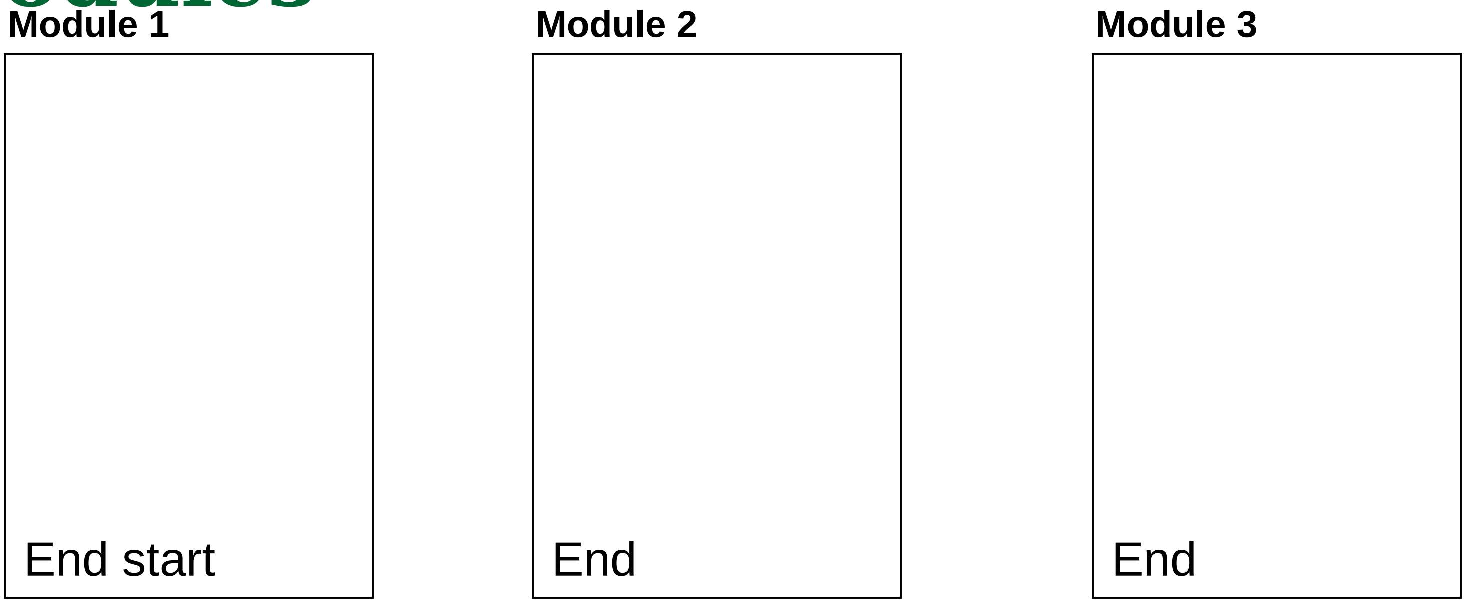
[*language*] *name* : *type*

The name of the symbol that is defined in other module

ABS, **BYTE**, **DATAPTR, DWORD, NEAR, FAR, FWORD, PROC, QWORD, TBYTE, UNKNOWN, WORD**

Ex: **EXTRN ProcA: near**

Linking several assembly language modules



Module 1

End start

Module 2

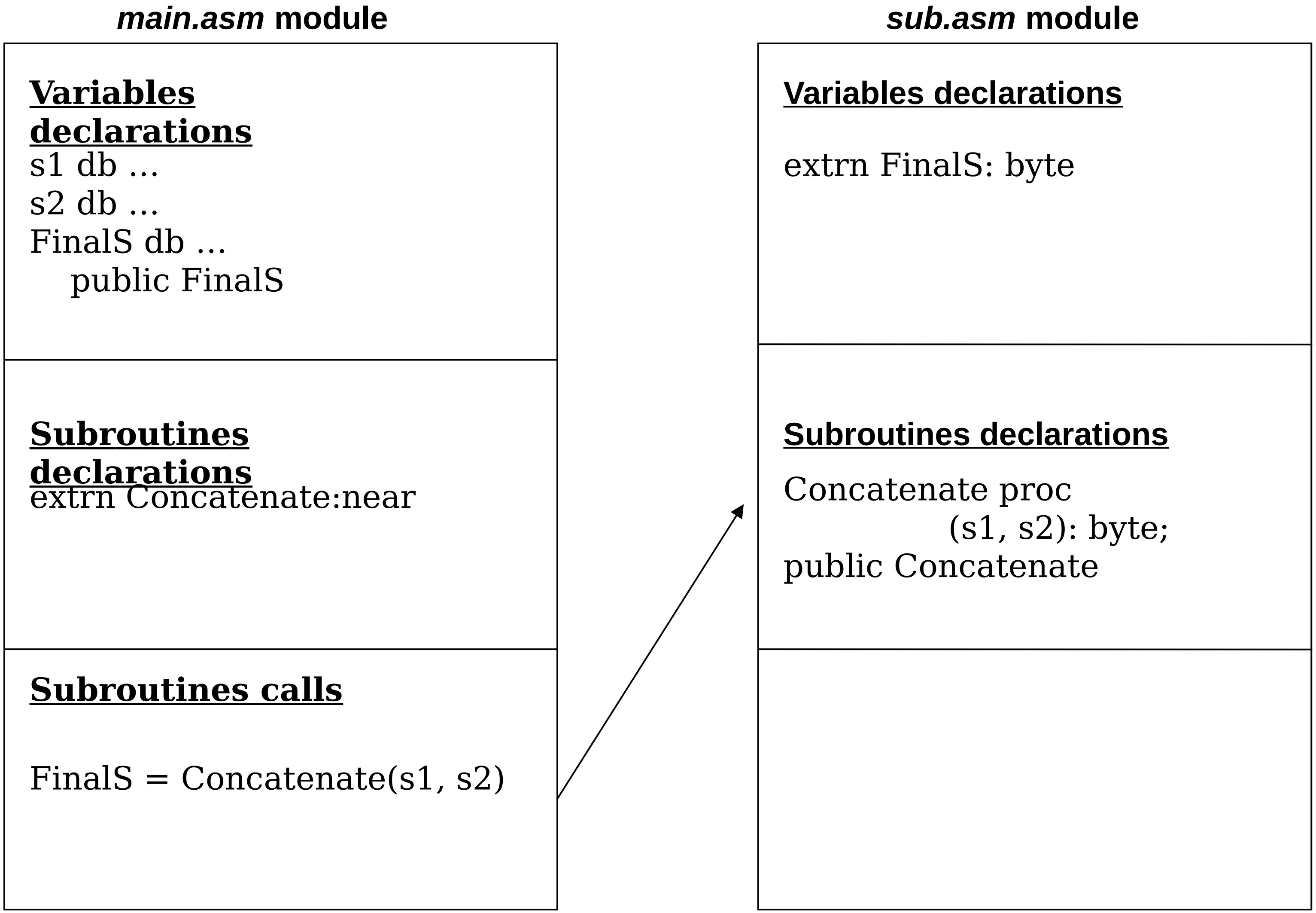
End

Module 3

End

- each module has an END directive at the end
- only the END directive of the module that contains the start instruction will specify the start address

Example



main.asm module

Variables declarations

s1 db …

s2 db …

FinalS db …

public FinalS

Subroutines declarations

extrn Concatenate:near

Subroutines calls

FinalS = Concatenate(s1, s2)

sub.asm module

Variables declarations

extrn FinalS: byte

Subroutines declarations

Concatenate proc (s1, s2): byte;

public Concatenate

***main.asm*:**

```
.MODEL SMALL
.STACK 200
.DATA
        s1 DB ‘Good ', 0
        s2 DB ‘morning!', '$', 0
        FinalS DB 50 DUP (?)
        PUBLIC FinalS
        ; could be replaced by GLOBAL FinalS:BYTE
.CODE
        EXTRN Concatenate:PROC
Start:
        mov ax, @data
        mov ds, ax                 ; it loads the ds register
        mov ax, OFFSET s1
        mov bx, OFFSET s2
        call Concatenate           ; FinalS:=s1+s2
        mov ah, 9
        mov dx, OFFSET FinalS
        int 21h                    ;it prints the obtained string
        mov ah, 4ch
        int 21h
                                   ; end of the program
END  Start
```

***sub.asm*:**

```
.MODEL SMALL
.DATA
EXTRN FinalS:BYTE
                ; could be replaced by GLOBAL FinalS:BYTE
.CODE
PUBLIC Concatenate
Concatenate PROC
     cld
     mov di, SEG FinalS
     mov es, di
     mov di, OFFSET FinalS
                  ;es:di <- the address of the final string
     mov si, ax
                  ;ds:si <- the address of the first string
     s1Loop:
          lodsb                     ; al <- the current character
          and al, al          ; it verifies if this is the final zero
          jz cont
          stosb     ; if not, it is placed in the destination string
     jmp  s1Loop
     cont:
     mov si, bx       ;ds:si <- the address of the other string
     s2Loop:
          lodsb
          stosb                     ; it loads the final zero as well
          and al, al
     jnz s2Loop
     ret                            ; return from the procedure
Concatenate ENDP
END
```

The main program ***main.asm*****:**

```
DATA SEGMENT
    S1 DB ‘Good', 0
    S2 DB ‘morning!', '$', 0
    FinalS DB 50 DUP (?)
    PUBLIC FinalS
DATA ENDS

CODE SEGMENT
    EXTRN Concatenate:PROC
(or  EXTRN Concatenare:FAR)
Start:
    mov ax, data
    mov ds, ax
    mov ax, OFFSET S1
    mov bx, OFFSET S2
    call FAR PTR Concatenate
(and then remains call Concatenate)
     ;FinalS:=S1+S2
    mov ah, 9
    mov dx, OFFSET FinalS
    int 21h             ; prints the obtained string
    mov ah, 4ch
    int 21h

END  Start
```

The secondary module ***sub.asm*****:**

```
ASSUME CS:CODE

EXTRN FinalS:BYTE
PUBLIC Concatenate

CODE SEGMENT
  Concatenate PROC
    cld
    mov di, SEG FinalS
    mov es, di
    mov di, OFFSET FinalS ;es:di <- final string address
    mov si, ax                      ; ds:si <- first string address
    Sir1Loop:
        lodsb                       ; al <- the current character
        and al, al                  ; checks if it is the final zero
        jz cont
        stosb    ; if not, it places it in the destination string
        jmp  Sir1Loop               ; resume operations
    cont:
    mov si, bx   ; ds:si <- the address of the second string
    Sir2Loop:
        lodsb
        stosb                       ; it loads the final zero as well
        and al, al
    jnz Sir2Loop
    ret                             ; return from the procedure
  Concatenate ENDP
END
```

The two modules will be separately assembled:

```
    TASM MAIN[.ASM]
    TASM SUB[.ASM]
```

the linkedit follows:

```
    TLINK MAIN[.OBJ]+SUB[.OBJ]
                 or
    TLINK MAIN[.OBJ] SUB[.OBJ]
```

It will result an executable program called ***main.exe*** which will print the message “Good morning!".

Linking assembly language modules with modules written in high level programming languages

- Requirements of the linkeditor
- Entering the procedure
- Keeping the values of some registers
- Passing and accessing parameters
- Allocating space for local data (optional)
- Returning a result (optional)
- Returning from the procedure

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

- The name of the segments are imposed by the high level programming languages

- Every symbol that is defined in the assembly language module and has to be visible in the module written in high-level programming language, has to be made visible using the **PUBLIC** directive

- Every symbol that is defined in the module written in high-level programming language and will be used in the assembly language module has to be declared as external in the assembly module using the **EXTRN directive**;

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

$L compilation directive (it can be placed anywhere inside the Pascal source text)

{$L name[**.obj**]**}**

The file ***name.obj*** has to fulfill the following conditions:

- All procedures and functions have to be placed inside a segment called **CODE** or **CSEG**, or inside a segment whose name ends with **_TEXT**;
- All initialized data have to be placed inside a segment called **CONST** or inside a segment whose name ends with **_DATA**;
- All uninitialized data have to be placed inside a segment called **DATA** or **DSEG**, or inside a segment whose name ends with **_BSS**;

Standard type declarations in Pascal have the following equivalences in assembly language:

Integer – WORD
Real – FWORD
Single – DWORD
Pointer – DWORD

Turbo Assembler * Turbo Pascal

Requirements of the linkeditor

- Entering the procedure
- Keeping the values of some registers
- Passing and accessing parameters
- Allocating space for local data (optional)
- Returning a result (optional)
- Returning from the procedure

Turbo Pascal

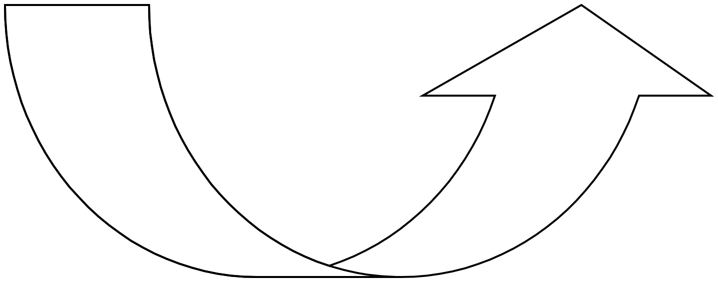
- A subroutine used in the Pascal program but defined in other module has to be declared using the **EXTERNAL** directive (only at the most exterior level of the program or unit):

```
Procedure AsmProc (a:Integer; b:Real); external;

Function AsmFunc (c:Word; d:Byte): Integer; external;
```

Turbo Assembler

The only objects that can be exported from an assembly language module to a Pascal program or unit are instructions labels or procedures names declared as **PUBLIC**.

```
CODE SEGMENT
AsmProc PROC NEAR
          PUBLIC AsmProc
          ...
AsmProc ENDP
AsmFunc PROC NEAR
          PUBLIC AsmFunc
          ...
AsmFunc ENDP
CODE ENDS
END
```



Turbo Assembler * Turbo Pascal

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

Turbo Assembler

- A TASM module can access every procedure, function, variable or constant with type declared at the most exterior level of a Pascal program or unit, including the unit libraries, using **EXTRN**

```
DATA SEGMENT
            ASSUME DS:DATA
            EXTRN A: BYTE
            EXTRN B: WORD
            EXTRN C: BYTE
            ...
DATA ENDS
CODE SEGMENT
            EXTRN ProcA:NEAR
            EXTRN FuncA:FAR
            ...
; the variables a, b, c can be used here and the
subroutines ProcA, FuncA can be called
CODE ENDS
```

Turbo Pascal

```
...
{global variables}
Var a: Byte;
    b: Word;
    c: ShortInt;
...
Procedure ProcA;
...
{$F+}
Function FuncA:Integer;
...
```

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

- when a procedure or a function is called, the caller puts first on the stack the return address and then it gives the control to the called subroutine using on this purpose a CALL instruction.

- the return address can be FAR or NEAR, depending on the memory model used when the module is compiled or assembled

- it is very important to return from the subroutine according to the call

- if, from a module written in high-level programming language (or assembly language module) one wants to call a subroutine written in assembly language (or high-level language) the link editor that links the two modules doesn’t verify if the type of the call (far or near) corresponds to the type of the return. The programmer has to take care of this.

- Requirements of the linkeditor
- Entering the procedure
- **Keeping the values of some registers**
- Passing and accessing parameters
- Allocating space for local data (optional)
- Returning a result (optional)
- Returning from the procedure

- high-level programming languages impose the fact that <u>some</u> registers, when returning from a subroutine, should keep the values they had when entering the routine

- for this purpose, if the assembly language subroutine changes some of them, their entry values have to be saved (possibly on the stack) and restored when quitting the subroutine

<u>Turbo Pascal – Turbo Assembler</u>

- when a function or a procedure is called in Turbo Pascal, the value of the following registers should remain unchanged: **SS, DS, BP, SP**

- when the subroutine is called:

SS points to the stack segment
DS points to the global data segment (called DATA)
BP points to the base of the stack
SP points to the top of the stack

- Requirements of the linkeditor
- Entering the procedure
- Keeping the values of some registers
- **Passing and accessing parameters**
- Allocating space for local data (optional)
- Returning a result (optional)
- Returning from the procedure

- **NEAR reference:** in the stack will be put the offset of the address (word)
- **FAR reference:** in the stack will be put 2 words: first the segment address and then the offset
- **value:** in the stack will be put the value of the parameter

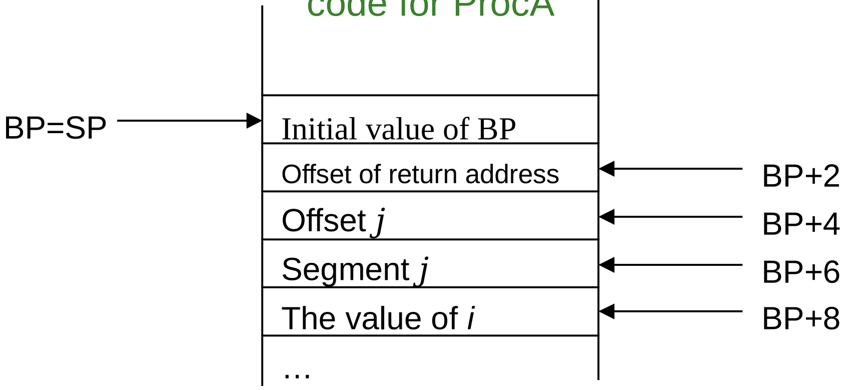
Turbo Pascal – Turbo Assembler

Procedure ProcA(i:integer; var j:integer); external;

- We suppose that we have a NEAR call

```
ProcA PROC NEAR
    PUBLIC ProcA
    j EQU DWORD PTR [BP+4]
    i EQU WORD PTR [BP+8]
    PUSH BP          ; entry code in ProcA
    MOV BP, SP
    ...
    MOV AX, i
; it loads in AX the value of the parameter i
    LES DI, j
    MOV ES:[DI], AX          ; j:=i
    …
```

The stack after the call code for ProcA



BP=SP

Initial value of BP

Offset of return address — BP+2

Offset *j* — BP+4

Segment *j* — BP+6

The value of *i* — BP+8

…

! Turbo Pascal – reference parameters = FAR references parameters

Turbo Assembler * Turbo Pascal

- Requirements of the linkeditor
- Entering the procedure
- Keeping the values of some registers
- Passing and accessing parameters
- Allocating space for local data (optional)
- Returning a result (optional)
- Returning from the procedure

Value parameters

TYPE	WHAT WILL WE HAVE ON THE STACK
Char	- Unsigned byte
Boolean	- byte (value 0 or 1)
type enumerare	- Unsigned byte, if the set has no more than 256 values - Unsigned word, otherwise
Real	- 6 byte on the stack (exception !)
Floating point	- 4, 6, 8, 10 bytes on the stack of the numeric coprocessor
Pointer	- 2 words on the stack
String	- pointer (far) to the value
type set	- The address of a set that has no more than 32 bytes
Array, Record	- The value on the stack, if it has no more than 1, 2 or 4 bytes - pointer to the value, otherwise

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

- if successive calls of the procedure do not have to keep their values, they will be allocated in the stack segment and they will be called volatile data.

- otherwise, they will be called static data and they will be allocated in a segment different than the stack segment, for example in the data segment (using the well-known directives DB, DW, DD ..)

- The allocation of *n* bytes (*n* – even number) for the local volatile data can be done using:

sub sp, n

“defining” and allocating 2 local variables for an external TP procedure (written in assembler)

```
push bp
mov bp, sp
sub sp, 4
minim EQU [bp-2]
maxim EQU [bp-4]
...
mov ax, 1
mov minim, ax
mov maxim, ax
…
```

Returning a result

- using the registers, if the returned value has no more than 4 bytes
    - **exception:** the real values, represented on 6 bytes, are returned using the registers (DX:BX:AX).

- if the returned value is longer, there are other methods for returning the result

Turbo Pascal – Turbo Assembler

- scalar result:
- 1 byte → in AL
- 2 bytes → in AX
- 4 bytes → in DX:AX (DX contains the **high** part)

- **real** result: in DX:BX:AX

- **floating point** result: in the registers of the numeric coprocessor

- **string** result: in a temporary area allocated by Turbo Pascal in the moment of the compilation of the program that contains the call of this function; a pointer to this area will be put on the stack before putting the parameters. This pointer is not part of the list of parameters, so it won’t affect the number of bytes that need to be extracted from the stack when returning from the function (see exit code);

- **pointer** results: the segment address will be put in DX and the offset in AX

- **Requirements of the linkeditor**
- **Entering the procedure**
- **Keeping the values of some registers**
- **Passing and accessing parameters**
- **Allocating space for local data (optional)**
- **Returning a result (optional)**
- **Returning from the procedure**

- restoring the values of the registers

- restoring the stack so that we have the return address on the top of the stack

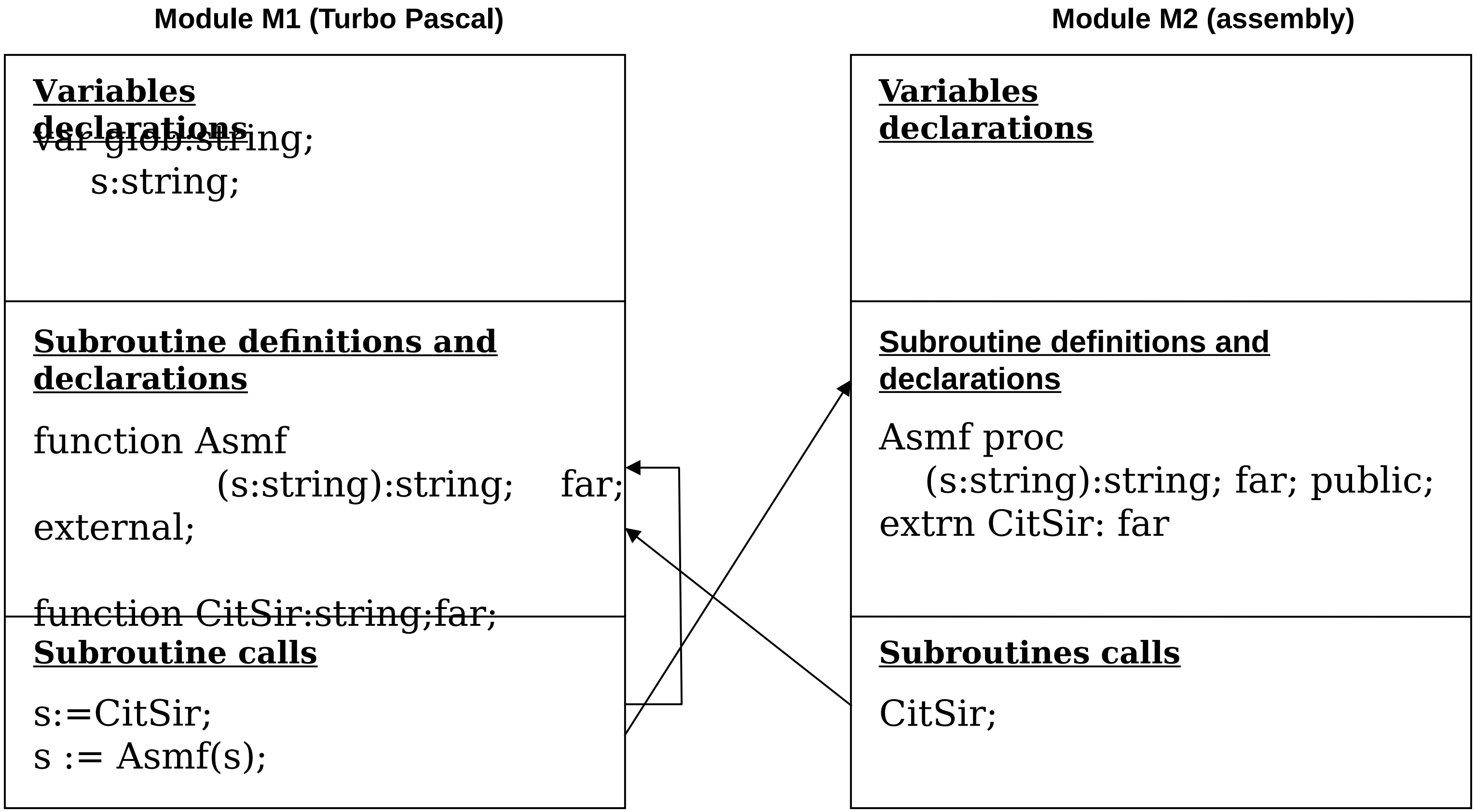
```
MOV SP, BP
POP BP
```

-If the high-level programming language requires that the called procedure should extract the parameters from the stack, this will be done using the instruction

```
ret n
```

where *n* is the number of bytes used for parameters

Example



Module M1 (Turbo Pascal)

Variables declarations

```
var glob:string;
    s:string;
```

Subroutine definitions and declarations

```
function Asmf
            (s:string):string;    far;
external;

function CitSir:string;far;
```

Subroutine calls

```
s:=CitSir;
s := Asmf(s);
```

Module M2 (assembly)

Variables declarations

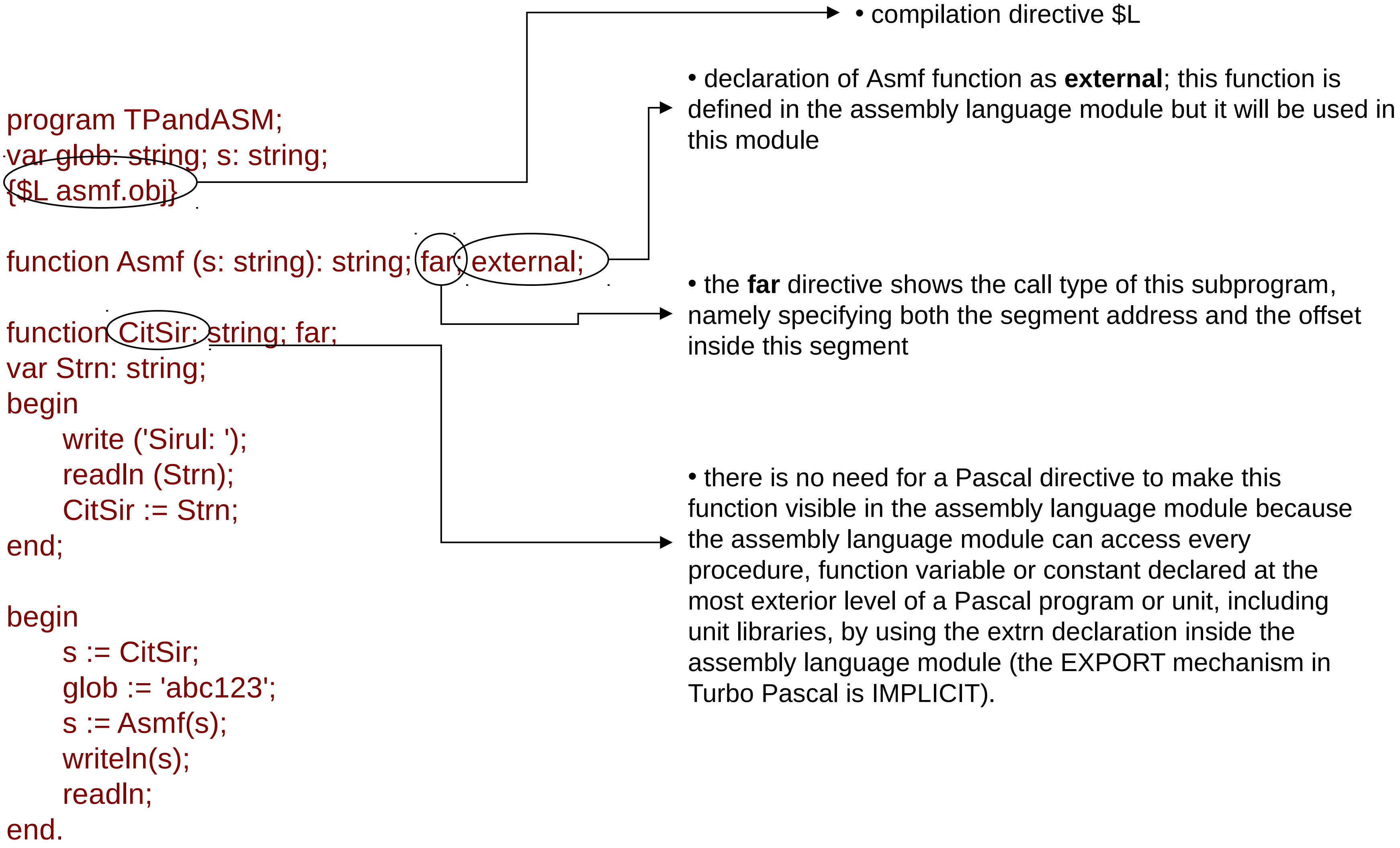
Subroutine definitions and declarations

```
Asmf proc
   (s:string):string; far; public;
extrn CitSir: far
```

Subroutines calls

```
CitSir;
```

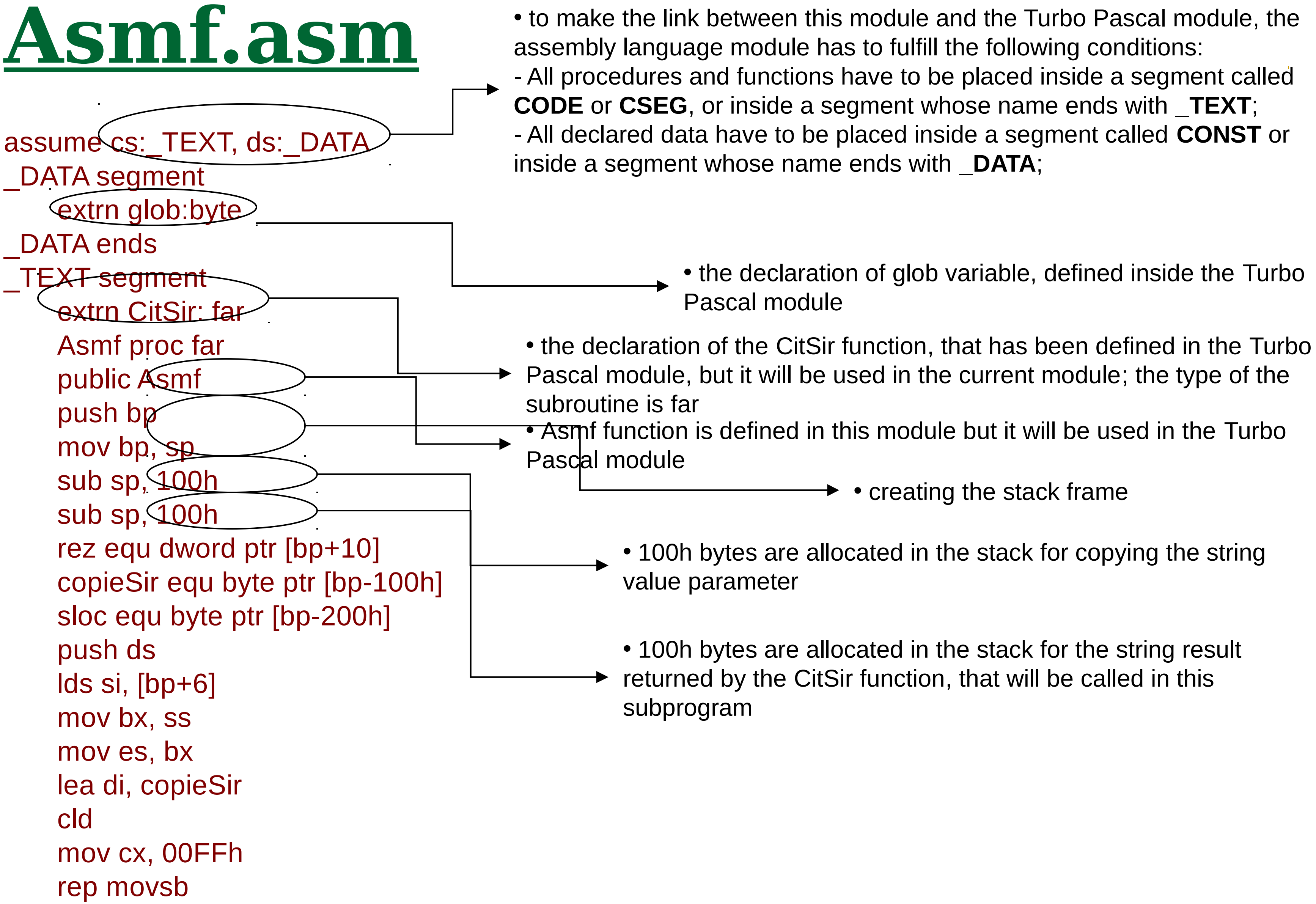
P.pas



```
program TPandASM;
var glob: string; s: string;
{$L asmf.obj}

function Asmf (s: string): string; far; external;

function CitSir: string; far;
var Strn: string;
begin
      write ('Sirul: ');
      readln (Strn);
      CitSir := Strn;
end;

begin
      s := CitSir;
      glob := 'abc123';
      s := Asmf(s);
      writeln(s);
      readln;
end.
```

- compilation directive $L
- declaration of Asmf function as **external**; this function is defined in the assembly language module but it will be used in this module
- the **far** directive shows the call type of this subprogram, namely specifying both the segment address and the offset inside this segment
- there is no need for a Pascal directive to make this function visible in the assembly language module because the assembly language module can access every procedure, function variable or constant declared at the most exterior level of a Pascal program or unit, including unit libraries, by using the extrn declaration inside the assembly language module (the EXPORT mechanism in Turbo Pascal is IMPLICIT).

Asmf.asm



```
assume cs:_TEXT, ds:_DATA
_DATA segment
    extrn glob:byte
_DATA ends
_TEXT segment
    extrn CitSir: far
    Asmf proc far
    public Asmf
    push bp
    mov bp, sp
    sub sp, 100h
    sub sp, 100h
    rez equ dword ptr [bp+10]
    copieSir equ byte ptr [bp-100h]
    sloc equ byte ptr [bp-200h]
    push ds
    lds si, [bp+6]
    mov bx, ss
    mov es, bx
    lea di, copieSir
    cld
    mov cx, 00FFh
    rep movsb
```

- to make the link between this module and the Turbo Pascal module, the assembly language module has to fulfill the following conditions:
  - All procedures and functions have to be placed inside a segment called **CODE** or **CSEG**, or inside a segment whose name ends with _**TEXT**;
  - All declared data have to be placed inside a segment called **CONST** or inside a segment whose name ends with _**DATA**;

- the declaration of glob variable, defined inside the Turbo Pascal module

- the declaration of the CitSir function, that has been defined in the Turbo Pascal module, but it will be used in the current module; the type of the subroutine is far

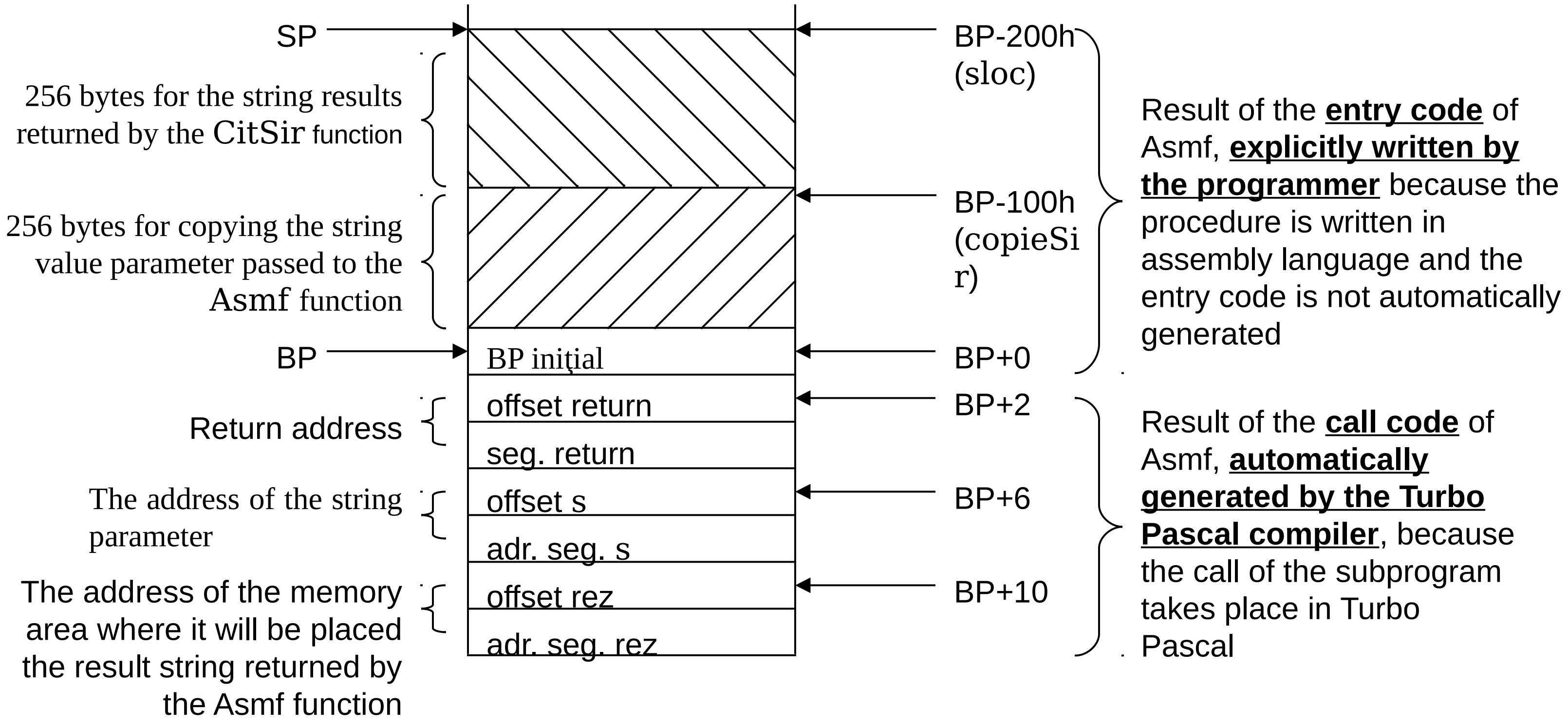
- Asmf function is defined in this module but it will be used in the Turbo Pascal module

- creating the stack frame

- 100h bytes are allocated in the stack for copying the string value parameter

- 100h bytes are allocated in the stack for the string result returned by the CitSir function, that will be called in this subprogram

Fig. 8.2. The stack after calling and entering Asmf function



SP

256 bytes for the string results returned by the CitSir function

256 bytes for copying the string value parameter passed to the Asmf function

BP

Return address

The address of the string parameter

The address of the memory area where it will be placed the result string returned by the Asmf function

BP inițial

offset return

seg. return

offset s

adr. seg. s

offset rez

adr. seg. rez

BP-200h (sloc)

BP-100h (copieSir)

BP+0

BP+2

BP+6

BP+10

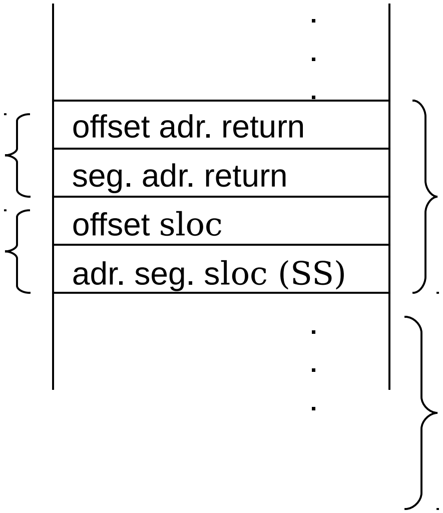
Result of the **entry code** of Asmf, **explicitly written by the programmer** because the procedure is written in assembly language and the entry code is not automatically generated

Result of the **call code** of Asmf, **automatically generated by the Turbo Pascal compiler**, because the call of the subprogram takes place in Turbo Pascal

Asmf.asm

```
push ss
lea ax, sloc
push ax
call CitSir
```

- the space for the resulting string is allocated inside the stack, therefore the segment address of the resulting string is SS
- we put on the stack this address and the offset
- the call of the CitSir function, which reads a string and puts it at the address ss:sloc



. . .
offset adr. return
seg. adr. return
offset sloc
adr. seg. sloc (SS)
. . .

Results of the call CitSir Explicitly written by the programmer

Results of the **call code** of CitSir function (no parameters for this function) **explicitly written by the programmer** because the function is called in assembly language module, and the calling code is therefore not automatically generated

The *Stackframe* of the called routine Asmf (see figure 8.2.)

Fig. 8.3. The stack after calling CitSir function

Asmf.asm

```
pop ax
pop ax
les di, rez
cld
mov bx, ss
mov ds, bx
lea si, copieSir
mov ch, 0
mov cl, byte ptr [si]
inc si
push di
inc di
cmp cx, 10
jb et1
mov cx, 10
et1:
    rep movsb
    push ss
    pop ds
    lea si, sloc
    mov cl, byte ptr [si]
    inc si
    cmp cx, 10
    jb et2
```

- extracting from the stack the address of the resulting string is the responsibility of the caller
- **add sp, 4** could also be used

- the number of elements of the string parameter is on the first byte of the string

- movsb instruction will be executed cx times; it will copy at the address es:di (where the resulting string is) a byte from the address ds:si (where the copy of the string parameter is)

- next, we will copy in the resulting string the first 10 characters from the string returned by the CitSir function (the sloc string)
- for this purpose, the address of the sloc string is loaded in ds:si

Asmf.asm

```
et2:
        rep movsb
        lea si, glob
        mov ax, _DATA
        mov ds, ax
        mov cl, byte ptr [si]
        inc si
        cmp cx, 10
        jb et3
        mov cx, 10
et3:
        rep movsb
        pop ax
        mov bx, di
        sub bx, ax
        dec bx
        les di, rez
        mov es:[di], bl
        pop ds

        mov sp, bp
        pop bp
        ret 4
asmf endp
_TEXT ends
end
```

- we restore the value that ds had when we entered the subroutine

- we restore the values of sp and bp (as part of the **exit code** of Asmf)
- the 200h bytes allocated for the string returned by CitSir and for the copy of the string parameter of Asmf will be therefore extracted from the stack

- returning from the function with extracting 4 bytes from the stack (the parameters: 2 bytes for the segment address and 2 bytes for the offset)

- extracting from the stack the parameters, as part of the **exit code** as well

Exit code from Asmf function, **explicitly written by the programmer**, because the function is written in assembly language and the exit code is not automatically generated