LOW LEVEL PROGRAMMING IN BORLAND PASCAL

- Machine code insertion in the source code of a Borland Pascal program

- Inserting instructions written in assembly language in a Pascal program (**the inline assembler of the** Borland Pascal 6.0 environment).

INSERTING MACHINE CODE IN THE SOURCE TEXT

- **The inline instruction**

inline (*inline_element* { / *inline_element* })

$$\left[\begin{pmatrix}<\\>\end{pmatrix}\right]\begin{pmatrix}\text{constant}\\ \text{var\_identifi}\mathit{er}\end{pmatrix}\left\{\begin{pmatrix}+\\-\end{pmatrix}\text{constant}\right\}$$

Example:

inline (<$1234/>$44)

generates three bytes of code: $34, $44, $00.

INSERTING MACHINE CODE IN THE SOURCE TEXT

The following inline instruction example generates machine code for copying a certain number of words to a specified address. The procedure *FillWords* below will store *Counter* words having the value *Data* in memory, starting with the first byte from the address contained in the *Dest* variable.

```
Procedure FillWords (var Dest; Counter, Data:Word);
begin
        inline(
                $C4/$BE/Dest/          { LES DI, Dest[BP]      }
                $8B/$8E/Contor/        { MOV CX, Contor[BP]    }
                $8B/$86/Data/          { MOV AX, Data[BP]      }
                $FC/                   { CLD                   }
                $F3/$AB);              { REP STOSW             }
end;
```

INSERTING MACHINE CODE IN THE SOURCE TEXT

- **The inline directive**

  - same syntax as the inline instruction
  - allows writing procedures and functions which when called are expanded to a given sequence of machine code instructions (**~ similar to macros in assembler**)
  - when an inline procedure or function is invoked, **the compiler generates the inline directive code ONLY, without generating any call code.**
  - parameters if present are put on the stack
  - because such procedures and functions are in fact macros, **no entry code and exit code is generated.**

```
Function LongMul(x,y:Integer):LongInt;
          inline(
            $5A/             { POP  DX  ; DX:=y              }
            $58/             { POP  AX  ; AX:=x              }
            $F7/$EA);        { IMUL DX  ; DX:AX := x*y       }
```

Correction to the example from page 311 in the coursebook!!!

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

- The Borland Pascal 6.0 inline assembler allows directly inserting assembly code into Pascal source code.

- **The asm instruction**

  asm *Asm_instr* { *Separator Asm_instr* } **end**

Examplu:

```
asm
    mov  ax, A; xchg ax, B; mov A, ax
end;
```

One asm instruction **has to preserve the integrity of the BP, SP, SS and DS registers**.

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

- **Assembler instructions**

[*Label* **:**] { *Prefix* } [*Mnemonic* [*Operand* { **,** *Operand* }]

- **Pascal labels** (defined in the declarations section of the Pascal program)

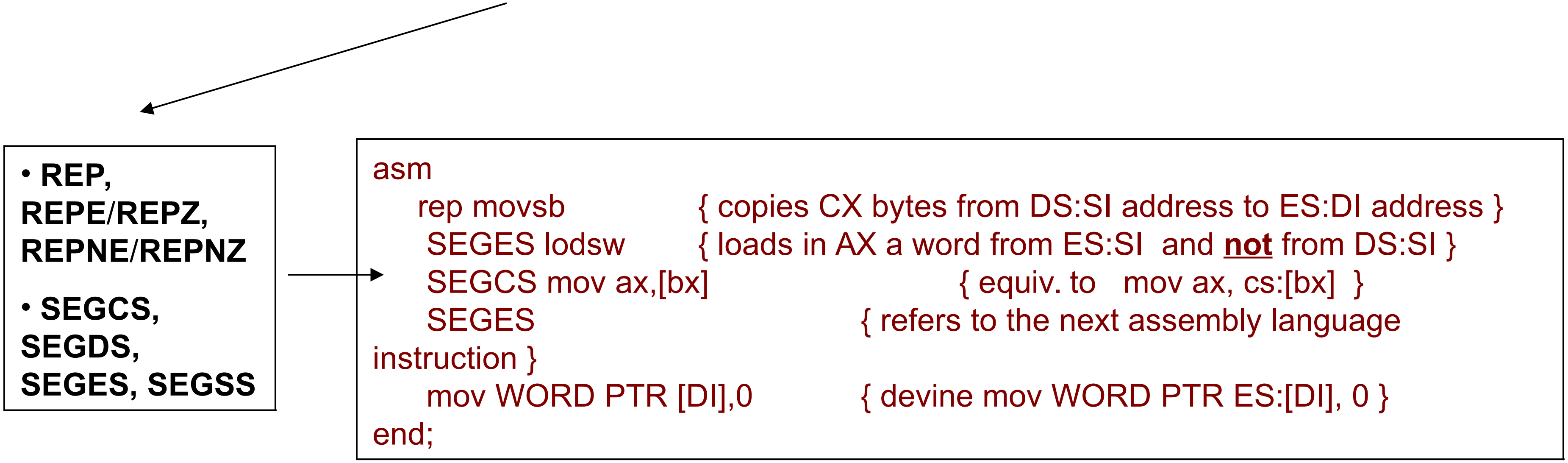
- **locale labels** (visibile only in the asm instruction in which they are defined; as a sintactic feature: they begin with the "@" symbol)

```
label Start, Stop;
...
begin
   asm
      Start:
             ...
             jz Stop
      @1:
             ...
             loop @1
   end;
   asm
       @1:
              ...
              jc  @2
              ...
              jmp @1
       @2:
   end;
   goto Start;
Stop:
end;
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

- **Assembler instructions**

[*Label* **:**] { *Prefix* } [*Mnemonic* [*Operand* { **,** *Operand* }]



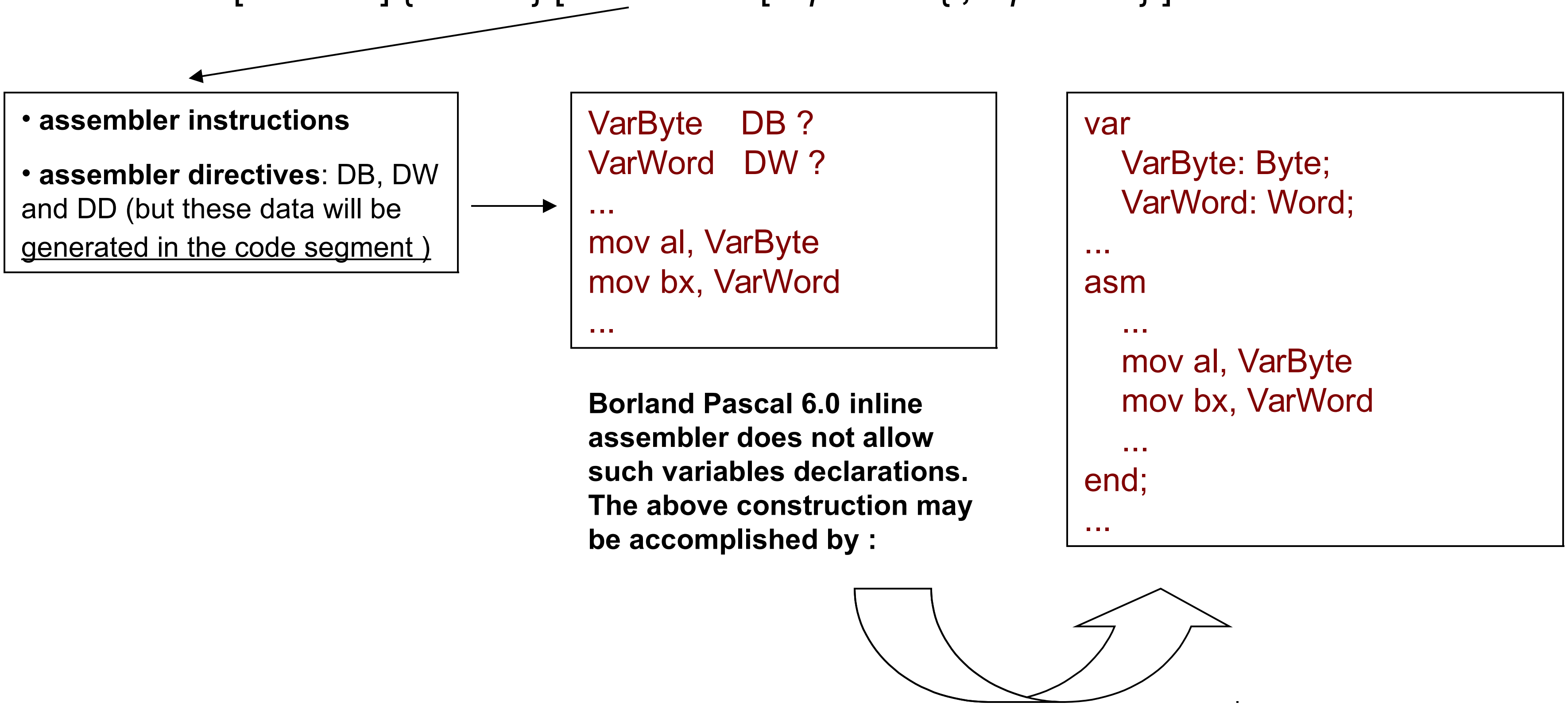
- **REP, REPE/REPZ, REPNE/REPNZ**
- **SEGCS, SEGDS, SEGES, SEGSS**

```
asm
     rep movsb              { copies CX bytes from DS:SI address to ES:DI address }
      SEGES lodsw           { loads in AX a word from ES:SI  and not from DS:SI }
      SEGCS mov ax,[bx]                            { equiv. to   mov ax, cs:[bx]  }
      SEGES                                  { refers to the next assembly language
instruction }
      mov WORD PTR [DI],0             { devine mov WORD PTR ES:[DI], 0 }
end;
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

- **Assembler instructions**

[*Label* **:**] { *Prefix* } [*Mnemonic* [*Operand* { **,** *Operand* }]



- **assembler instructions**
- **assembler directives**: DB, DW and DD (but these data will be generated in the code segment)

```
VarByte    DB ?
VarWord    DW ?
...
mov al, VarByte
mov bx, VarWord
...
```

Borland Pascal 6.0 inline assembler does not allow such variables declarations. The above construction may be accomplished by :

```
var
   VarByte: Byte;
   VarWord: Word;
...
asm
   ...
   mov al, VarByte
   mov bx, VarWord
   ...
end;
...
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

- **Assembler instructions**

[*Label* **:**] { *Prefix* } [*Mnemonic* [*Operand* { **,** *Operand* }]

• **expressions** – combinations of constants, registers, symbols and operators, reserved words: AH, CL, FAR, SEG, AL, CS, HIGH, SHL, AND, CX, LOW, SHR, AX, DH, MOD, SI, BH, DI, NEAR, SP, BL, DL, NOT, SS, BP, DS, OFFSET, ST, BX, DWORD, OR, TBYTE, BYTE, DX, PTR, TYPE, CH, ES, QWORD, WORD and XOR.

```
Var ch:Char;
...
asm
  mov ch,1; { refers to the CH register }

  mov &ch,1; { refers to the ch variable }
end;
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

Expressions

- evaluates all expressions as 32 bits integer values
- doesn’t allow real numbers or strings (only string **constants**)
- Hexadecimal constants may be written also in the Pascal syntax (preceeded by the "$“ symbol)
- Accessing a variable means its offset in the first place as in T.Assembler

```
Var x: Integer;
...
asm
  mov ax, x+4;

{ stores in AX the word value from address x+4 !}

  mov bx,x; { stores in BX the value of x }
end;
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

Expressions

- Identifiers allowed by the inline assembler are : **Pascal labels and constants, Pascal type names and variables, Pascal procedures and functions and the special symbols: @Code, @Data and @Result.**

```
asm
   mov ax, SEG @Data
   mov ds, ax
{stores in the DS register the segment address of the
current data segment }
end;
```

```
Procedure X;
var c: Integer;
...
asm
   mov ax,c { generates code similar to mov ax, [BP-2] }
end;
...
```

In inline assembler expressions **are not allowed** :

- standard procedures and functions;
- array names as *Mem*, *MemW*, *MemL*, *Port*, *PortW*;
- string constants longer than 4 bytes;
- real constanta and set constants;
- inline procedures and functions ;
- non-local labels ;
- the *@Result symbol* outside of a function

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

```
Function Suma(x,y:Integer):Integer;
begin
   asm
                  mov ax, x
                  add ax, y
                  mov @Result, ax
{puts the AX value in the place from where the caller will take the result returned by the function}
   end;
end;
```

```
Function Suma(var x,y:Integer):Integer;
begin
asm
  les bx, x
{ due to call-by-reference x is a far address - loaded here in ES (segment) and in BX (offset)}
  mov ax, es:[bx]
{ stores in AX the value found at address ES:[BX], namely the value of the parameter x }
  les bx, y
{ similar as above…….}
  add ax, es:[bx]
{ adds to AX the value from address ES:[BX], namely the value of parameter y }
  mov @Result, ax
{ transfers the value that must be returned from AX to the place where the caller will take the result returned by the
function }
end;
end;
```

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

Expression type

```
asm
  mov al,[100h]

{ puts in AL a byte from address  ds:[100H];
the associated type is inferred from the size of
AL register – it is 1 byte}

  mov bx,[100h]

{puts in AL a word from address  ds:[100H];
the associated type is inferred from the size of
BX register – it is 2 bytes}

end;
```

Sometimes, a memory reference doesn't have an explicit associated type, this being inferred from the type of the other operands

```
asm
    inc  WORD PTR [100h]
    imul BYTE PTR [100h]
end;
```

If the type cannot be inferred, the assembler requires an explicit type conversion.

THE INLINE ASSEMBLER OF BORLAND PASCAL 6.0

“assembler” procedures and functions

- Procedures (functions) labeled as “assembler” are procedures (functions) written entirely in inline assembling, without “begin ... end” part being necessary.

```
Function LongMul(x,y:Integer):LongInt; assembler;
asm
    mov  ax,x
    imul y
end;
```

- They are defined by the ***assembler*** directive
- When using the **assembler** directive the compiler performs some **optimizations** when generating the subroutine entry code :
  - **NO code is generated for copying value parameters to local variables if their size is > 4 bytes, these parameters must be handled as being passed by reference;**
  - **NO variable allocated for returning a function result, except for the string functions**
  - **NO stackframe generated for procedures and functions that have no parameters and no local variables.**

Functions return their result as follows:

- **Integer, Char, Boolean, enumeration:**
- **1 byte - in AL**
- **2 bytes - in AX**
- **4 bytes - in DX:AX**
- **Real - in DX:BX:AX**
- **Single, Double, Extended, Comp - in ST(0)**
- **Pointer - in DX:AX**
- **String - in the temporary location pointed by *@Result***

“assembler” procedures and functions

Example: Function that operates on strings built with inline assembler instructions; one variant without assembler directive and one with. The function returns a string representing the uppercase variant of the string given as parameter. The parameter is passed by value.

Varianta 1

```
Function UpperCase( Str:String): String;
begin
  asm
     cld
     lea si, Str
     les di, @Result
     SEGSS lodsb     ;due to call by value
     stosb
     xor ah, ah
     xchg  ax, cx
     jcxz @3
  @1:
     SEGSS lodsb
     cmp al, 'a'
     jb @2
     cmp al, 'z'
     ja @2
     add al, 'A'-'a‘      ; lower to upper…
  @2:
     stosb
     loop @1
  @3:
  end;
end;
```

Varianta2

```
Function UpperCase( Str:String): String; assembler;
asm
    push ds
    cld
    lds si, Str
    les di, @Result
    lodsb
    stosb
    xor ah, ah
    xchg ax, cx
    jcxz  @3
@1:
    lodsb
    cmp al, 'a'
    jb @2
    cmp al, 'z'
    ja @2
    add al, ‘A'-'a'
@2:
    stosb
    loop @1
@3:
    pop ds
end;
```

ACCESSING REGISTERS AND INVOKING INTERRUPTS IN Borland Pascal 6.0

```
type registers = record
           case Integer of
                      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : Word);
                      1: (AL, AH, BL, BH, CL, CH, DL, DH : Byte);
end;
```

defined in the **dos unit**

A Pascal program that displays a text on the screen, using for that purpose the DOS function 09h:

```
uses dos;
const mesaj: String= 'Hello, everybody ! $';
var
          reg: Registers;
begin
          reg.AH:= 9;                { loading AH with 9 }
          reg.DS:= Seg(mesaj);       { loading in DS:DX the far address of the string to be displayed }
          reg.DX:= Ofs(mesaj[1]);
          Intr($21,reg);                  { issuing INT 21h }
end;
```

procedure defined in the **dos** unit

SCRIEREA DE RUTINE DE TRATARE A ÎNTRERUPERILOR ÎN LIMBAJUL PASCAL

- salvarea lui CS şi a lui IP în stivă;
- salvarea în stivă a registrului de flag-uri;
- interzicerea apariției altor întreruperi;
- salt far la locația punctată de vectorul de întrerupere corespunzător.

<u>Proceduri interrupt în Pascal</u>

```
procedure MyInt(rFlags, rCS, rIP, rAX, rBX, rCX, rDX, rSI, rDS, rES, rBP:Word); interrupt;
begin
                ...
end;
```

- nu poate să fie apelată dintr-o altă procedură
- trebuie să fie declarată far
- parametrii corespund regiştrilor procesorului
- Indiferent de lista parametrilor unei proceduri interrupt, compilatorul produce (automat) cod la intrarea în rutina pentru salvarea tuturor regiştrilor pe stivă
- corespunzător, la ieşirea din rutină se restaurează automat aceşti regiştri şi se generează (tot automat) o instrucțiune iret

SCRIEREA DE RUTINE DE TRATARE A ÎNTRERUPERILOR ÎN LIMBAJUL PASCAL

Proceduri interrupt în Pascal

```
push ax
push bx
push cx
push dx
push si
push di
push ds
push es
push bp
mov  bp, sp
mov  ax, seg @Data
mov  ds, ax
```

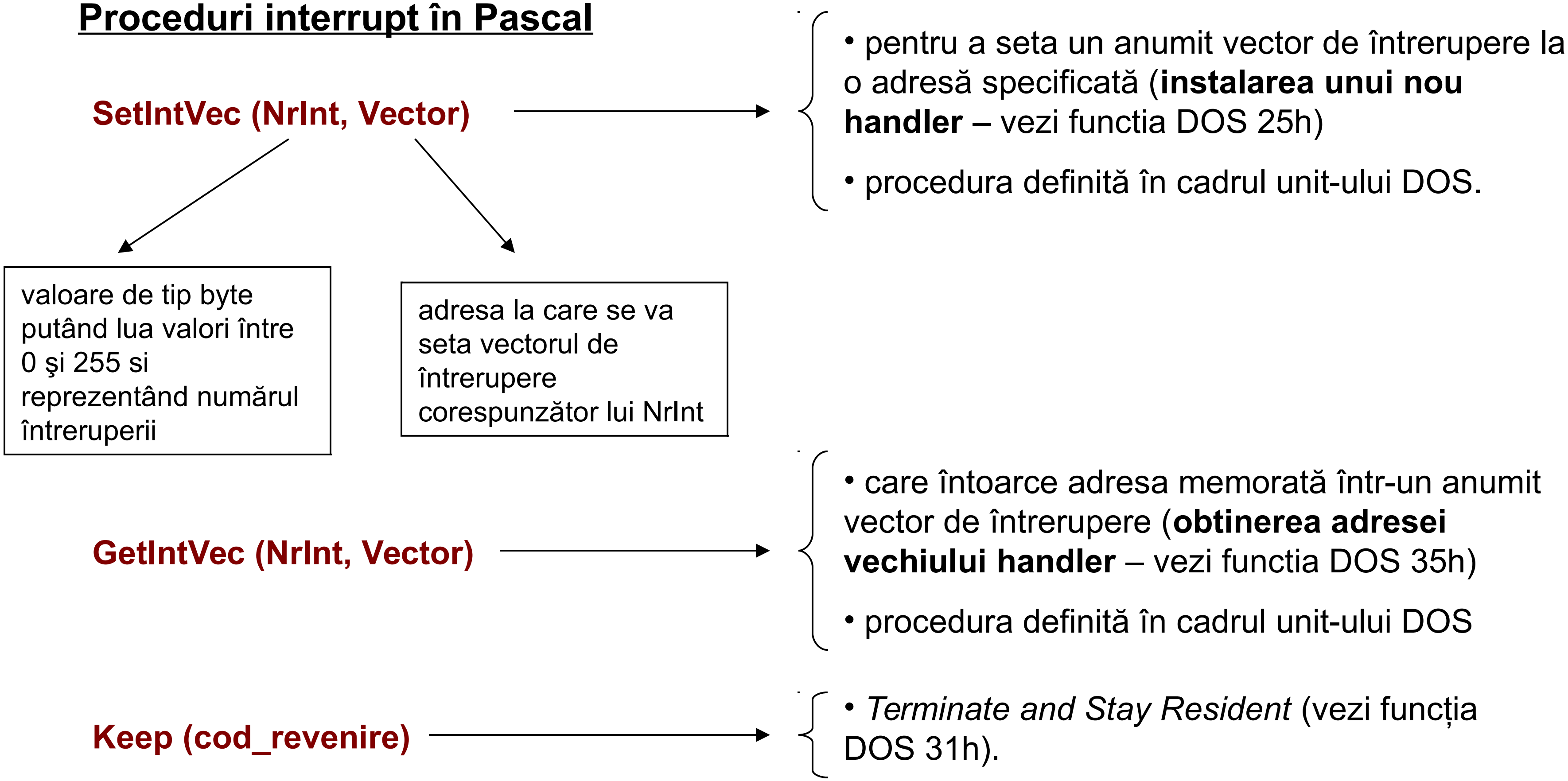
Cod de intrare

```
pop bp
pop es
pop ds
pop di
pop si
pop dx
pop cx
pop bx
pop ax
iret
```

Cod de iesire

SCRIEREA DE RUTINE DE TRATARE A ÎNTRERUPERILOR ÎN LIMBAJUL PASCAL

Proceduri interrupt în Pascal



SetIntVec (NrInt, Vector)

- pentru a seta un anumit vector de întrerupere la o adresă specificată (**instalarea unui nou handler** – vezi functia DOS 25h)
- procedura definită în cadrul unit-ului DOS.

valoare de tip byte putând lua valori între 0 şi 255 si reprezentând numărul întreruperii

adresa la care se va seta vectorul de întrerupere corespunzător lui NrInt

GetIntVec (NrInt, Vector)

- care întoarce adresa memorată într-un anumit vector de întrerupere (**obtinerea adresei vechiului handler** – vezi functia DOS 35h)
- procedura definită în cadrul unit-ului DOS

Keep (cod_revenire)

- *Terminate and Stay Resident* (vezi funcția DOS 31h).

SCRIEREA DE RUTINE DE TRATARE A ÎNTRERUPERILOR ÎN LIMBAJUL PASCAL

Proceduri interrupt în Pascal

Exemplu: program care modifică rutina de tratare a înteruperii 9, afişând la fiecare apăsare a tastei 'A' mesajul 'Ați apăsat tasta A'.

```
{$M $800,0,0 }                          { 2K stack, no heap }
uses Crt, Dos;
var c:char;
    OldHand : Procedure;
{$F+}
procedure MyHand; interrupt;
var i:Byte;
begin
             i := Port[$60];            { se citeşte un octet din portul $60 al controlerului tastaturii}
             inline ($9C);              { PUSHF - salvăm flagurile pe stivă }
             OldHand;
             if (i=65) then Writeln('Ati apasat tasta A')
end;
{$F-}
begin
             GetIntVec($9,@OldHand);
             SetIntVec($9,Addr(MyHand));
             Keep(0);                   { Terminate, stay resident }
end.
```