CHAP.5 INTERRUPTS

- An interrupt is a microprocessor's action announcing an event. More precisely, an interrupt is an electrical signal transmitted to the Computing System (CS) announcing the issuing of a particular event.

The actions performed by the CS when an interrupt is issued are:

- 1. program suspension;
- 2. activation of a specialized routine, called *Interrupt Handling Routine (IHR)* representing the CS answer to that event;
- 3. possibly, resuming the suspended program's execution (depending on the interrupt's type).

The reasons for which interrupts are issued may be:

- a). *external* (keystrokes, starting or ending I/O operations);
- b). *internal* (trying dividing by 0, attempt of addressing an inexistent memory area, attempt of running an instruction having an inexistent code, overflow).

Usually, after handling an external interrupt the program resumes; after an internal one, it doesn't !

When an interrupt is issued, the CS must (in this order):

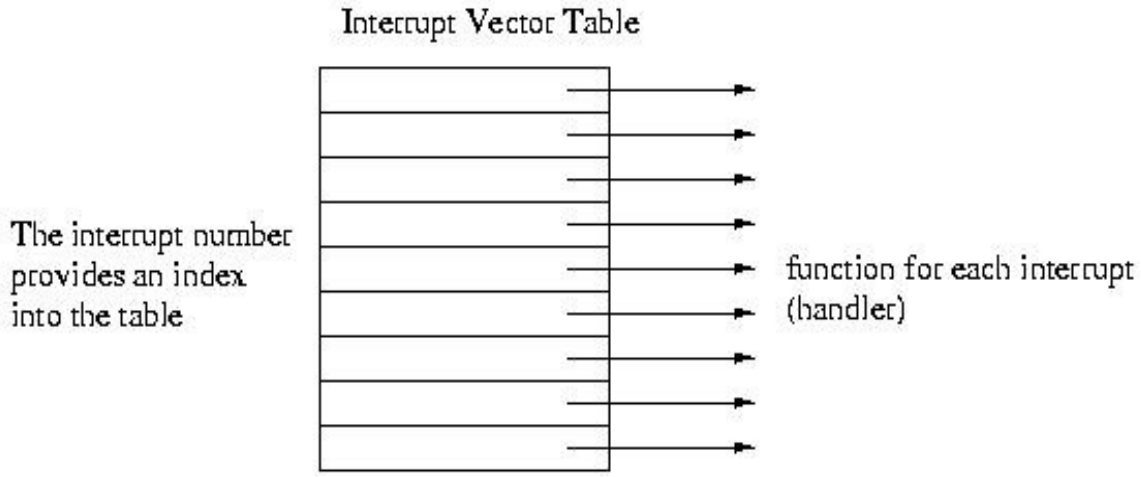
- 1) to determine the type of event that generated the interrupt (internal or external);
- 2) to find out the cause of that interrupt;
- 3) to get the address of the corresponding IHR (*Interrupt Handling Routine*);

- There are 3 cathegories of IHRs:
  - provided together with the CS;
  - written by the Operating System's designers;
  - user-defined IHR;

- For a fast access to the corresponding IHR, interrupts vectorizing is used: associating every interrupt with a fixed address doubleword memory location, where the far address of the corresponding IHR is stored.

- The IHR table (the interrupts array) is stored in memory at address 0000:0000. The first 256 x 4 = 1024 bytes contain these addresses (the area is called *the Interrupt Descriptor Table* - IDT). For INT k, the address of the IHR(k) is found at the address 0000 : k*4.

- This array with addresses is initialized at OS loading time. During CS run, these addresses may be accidentally or intentionally modified. The accidental one usually leads to CS blocking, a cold start-up being necessary afterwards. The intentional one is called *interrupt redirection (chapter 6)*.



5.2. INTERRUPTS CLASSIFICATION

The 80x86 architecture supports 3 types of events called in documentation interrupts:

- a). *hardware interrupts* – automatically generated, as a reaction to some external causes (peripheral devices signals usually linked to the I/O system); these are BIOS interrupts (**B**asic **I**nput **O**utput **S**ystem). The corresponding IHRs are loaded in memory from the ROM-BIOS files at CS startup.

- b). *exceptions* - automatically generated, as a reaction to some internal causes (zero divide, attempt of running an inexistent instruction - *inexistent code*, accessing a restricted memory area - *memory protection fault*).

- c). *software interrupts (traps)* – these suppose a programmer's initiated execution control transfer to the corresponding IHR (handler). The programmer is doing that by using the INT instruction. Those interrupts are called software interrupts because they are SOFTWARE invoked by an explicit instruction.

So we may give an alternate definition to this kind of interrupts: those interrupts that may be initiated ONLY by the programmer by INT are *software interrupts*.

5.2.1. Hardware interrupts

- In documentation we identify the hardware interrupts by refferal to IRQ (*Interrupt Request*), this being a native interrupt request sent at the level of a particular microprocessor device named PIC (*Programmable Interrupt Controller* – Intel 8259A). Example.: IRQ 0 = INT 8 , IRQ 1 = INT 9 etc.

- Details regarding the functioning of this device and about hardware interrupts can be found at: http://www.delorie.com/djgpp/doc/ug/interrupts/inthandlers1.html.

Most well-known and frequently used hardware interrupts are:

- **INT 8** is the clock's hardware interrupt (*system timer*). It is identified by IRQ 0.

- **INT 9** is the *keyboard interrupt*. It is identified by IRQ 1. This interrupt is generated by the keyboard at every key press and release. The actual actions performed by the corresponding IHR are: BIOS acts by reading the key scan-code (code associated to a key depending on its keyboard position), converting it to the corresponding ASCII code (see the corresponding Scan code/ASCII code pairs from documentation – Norton Guide), followed by storing this pair of attributes in the keyboard buffer, this being localized at the memory address 0000:041Ch. Starting from this address until the offset 043eh (on a length of 20h = 32 bytes) one may store 16 characters (for each character one scan code/ASCII code pair). Attention! – DO NOT disable interrupt 9, because the CS will not respond neither to CTRL + ALT + DELETE !!!. _

- **INT 0Bh** (IRQ 3) and **INT 0Ch** (IRQ 4) – interrupts for handling serial ports.
- **INT 0Dh** and **INT 0Fh** - interrupts serving parallel ports. Initially, interrupts INT 0Dh (IRQ 5) and INT 0Fh (IRQ 7) were designed for managing the LPT1 and LPT2 (*Line Printer*) parallel ports, but immediately after that, IBM designed a new printer interface (*printer interface card*) not compatible to these 2 interrupts. As a result, today these are not anymore used for printers but mostly for SCSI and sound cards.
- **INT 0Eh** → IRQ 6 - *Diskette Drive interrupt*
- **INT 76h** → IRQ 14 - *Hard Disk Controller*
- **INT 70h** - The Real-Time Clock Interrupt (IRQ 8). This one is activated 1024 times/second for managing the real time system clock.
- **INT 75h** - (*FPU – Floating Point Interrupt* - IRQ 13) - generated by the mathematical coprocessor at every *floating-point exception*.

5.2.2. **Exceptions**

Due to the fact that these also are BIOS interrupts, there are classifications which consider exceptions to be hardware interrupts (which are at their turn also BIOS interrupts). But this is not correct because after handling a hardware interrupt the program always resumes, while after exceptions handling, usually doesn't !!

INT 0 - *Zero Divide interrupt*. Interrupt 0 is generated every time when a so called *zero-divide condition* arises. INT 0 is issued in the following 3 distinct ways:

i). if there is a quotient overflow after performing a division (DIV or IDIV);

```
mov ax,600
mov bh,2
(i)div bh; performs ax/bh, quotient in AL (300 in AL ?!?) and the remainder in AH
```

300 doesn't fit in a byte, so INT 0 will be issued with the error message "*Divide by zero*". The recommended solution in such a case is performing a conversion by enlargement :

```
mov ax,600
mov dx,0  ; or cwd if a signed conversion is needed
mov bx,2
(i)div bx    ; performs dx:ax/bx, quotient in AX and remainder in DX
```

Now 300 fits AX and no INT 0 is issued anymore.

- ii). if attempting performing a division by zero:
- mov ax, 600
- mov bh,0
- div bh ; an INT 0 will be issued !

- iii). explicit issuing of this interrupt by software invoking it as INT 0:
- …….
- add ax,2
- INT 0 ; explicit issue of INT 0 by the programmer
- ……..
- The IHR of INT 0 displays "*Divide by zero*" and returns the control to the OS.

- **INT 1** (*Single Step*). This interrupt is initiated by the processor after each machine instruction, if TF = 1. It is used at debugging, the step by step execution being possible just because an INT 1 is automatically issued after each source code line.

- **INT 2** - *Non-Maskable Interrupt* - NMI. Interrupts may be disabled (masked) by the CLI instruction (*Clear Interrupts*). INT 2 is the only interrupt which cannot be disabled, it being generated every time a non-maskable condition arises (as for example a *memory parity error*).

- **INT 3** (*Breakpoint interrupt*). This interrupt is used by debuggers for establishing *breakpoints* during debugging.

- **INT 4** (*Overflow*). This interrupt is issued when an overflow arises during an arithmetic operation. More precisely, it is issued when an instruction INTO (*interrupt on overflow*) is run and OF = 1. If OF = 0, then INTO se results in a NOP (*no operation*).

  Writing a handler for INT 4 is a way of correctly managing arithmetic overflow conditions. By running an INTO instruction after an arithmetic operation exposed to overflow, one may accomplish an adequate management of overflows.

  Instruction INTO has the following effect:

  ```
  INTO ↔ if (OF=1)          PUSHF
         TF:=0; IF:=0
         CALL FAR 0000:0010h
  ```

- **INT 6** – (*invalid opcode*) – expresses that the microprocessor has encountered an illegal instruction code. INT 6 is issued when attempting to run an inexistent instruction code. For example:

  ```
        add ax,2 ; ok, run without any problems
  a db 199       ; Illegal instruction !!!! INT 6 is issued !
  ```

- **5.2.3. Software interrupts**

A software interrupt is invoked ONLY by the programmer by using the INT instruction (excepting interrupts 05h, 19h, 1Bh). The difference between invoking an interrupt by INT and the far call of a procedure with CALL is that INT puts also the flags onto the stack before the returning address, so that an IRET will be necessary for returning from a IHR (in contrast to RET, IRET pops out from the stack the flags register).

Software interrupts are classified in:

- BIOS interrupts (the IHR corresponding to those interrupts are loaded in memory at CS startup from the ROMBIOS files)
- DOS interrupts (the IHR corresponding to those interrupts are loaded in memory at CS startup from the BDOS files)
- Other interrupts do not have a previously defined goal or users may write their own IHR (user interrupts).

The main BIOS interrupts are:

- **05h** Issued when PrintScreen is pressed for sending the screen contents to the printer.
- **10h** Services related to working with the screen, text mode and graphic mode.
- **11h** Returns the list of the BIOS devices installed in the CS.
- **12h** Returns the size of the RAM memory.
- **13h** Services for working with the harddisk and diskette.
- **14h** Allows access to serial ports.
- **15h** Access functions to the extended memory, reading joystick-type devices etc.
- **16h** keyboard management services.
- **17h** printer management.
- **19h** operating system load services. The effect is equivalent to Ctrl-Alt-Del.
- **1Ah** system clock services : read (function 00h) and set (function 01h).
- **1Bh** initiated when <CTRL/BREAK> is pressed.
- **1Ch** called 18.2 times / second by INT 8. The corresponding IHR does not take any specific action, letting the user writing its own IHR. This is a user interrupt.

Main DOS interrupts

20h Program termination. As a result of this call the memory will be freed.
25h Absolute Disk Read. Allows physical disk read, starting from a certain sector to a memory area.
26h Absolute Disk Read. Allows physical disk write from a certain memory area to a given disk sector.
27h Terminate and stay resident. Ends the execution of the current program letting it or a part of it resident in memory.
28h DOS idle (Internal interrupt). *Undocumented DOS interrupt* used for time sharing**.**
2Eh DOS Execute Command. Undocumented. Runs a DOS command like given from the prompter.
2Fh Multiplex interrupt: multiplex of CS resources, extended memory management (XMS) if it is present, TSR programs control.
33h The mouse interrupt. This one groups all the functions necessary when working with the mouse.

22h DOS terminate address ; **23h** <CTRL/BREAK> exit address; **24h** DOS Critical error handler address.

Main DOS functions

The main DOS interrupt is **21h**. It encompasses the whole BDOS component of the DOS OS.
Memory management functions:
48h Allocate a memory block and returns a pointer to its beginning.
49h Frees a memory area.
4ah Adjusts memory block size (SETBLOCK).
Process management functions:
4Bh Load or Execute (EXEC). Loads a program to be run under the control of an existent program.
4Ch Quit with exit code (EXIT). Ends the execution of a program, returning an error level set in AL.
31h Terminate and stay resident. Ends the execution of a program letting it resident. An error level set in AL will be returned.
4Dh Get exit code of subprogram (WAIT). Used for obtaining the exit code of a child process run by a program which calls 4Bh.
26h Create PSP. It copies the PSP of the current program at a certain memory address, then updates the new PSP for being able to be used by a new program.
62h Get PSP segment. Obtains the PSP starting address.

- **Disk functions:**
- **19h** Get default disk number. Returns the code (0=A, 1=B, ...) of the implicit disk.
- **33h** Get boot drive. Returns the disk code used for OS loading.
- **1Bh** Get allocation table information for default drive. Provides information about the File Allocation Table (FAT).

- **Specific functions for files and directories:**
- **39h** Create a subdirectory (mkdir). Creates a new directory using the specified disk and path.
- **3Ah** Remove a directory entry (rmdir). Deletes a directory.
- **3Bh** Change the current directory (chdir).
- **47h** Get current directory. Returns an ASCIIZ string representing the path to the current directory.
- **56h** Rename a file.
- **4Eh** Find First. Search the first filename matching a given specification.
- **41h** Delete a file. Deletes a file from the specified directory.
- **3Fh** Read from file. Reads a certain nr of bytes from an opened file.
- **40h** Write to file. Writes a certain nr of bytes to an opened file.

- **Input / output with character peripheral devices:**
- **01h** Reads a character from the standard input and displays it to the standard output.
- **02h** Displays a character to the standard output.
- **09h** Displays a string to the standard output.
- **0Ah** Reads a string from the standard input until *Enter* is pressed.

- **Other functions:**
- **35h** Get interrupt vector. Obtains the address of a IHR (segment:offset).
- **25h** Set interrupt vector. Modifies the address of a IHR.
- **44h** IOCTL. Set of functions for I/O control of devices.
- **34h** Returns the nr of currently active processes.
- **52h** Assigns values to some DOS system variables (undocumented).

Some remarks on 8086 interrupts.

Event driven handlers- serve unusual events, internal or external.
Handlers activated by special interrupt call instructions – they offer same services as a subroutine library, callable by the user's programs

Some DOS functions of the 21h interrupt perform tasks run also by some other interrupts (ex: function 4ch of interrupt 21h – program termination, accomplished also by interrupt 20h)

Undocumented interrupts - reserved for being used by the OS designers

- some of them are effectively reserved
- some have intermediary tasks for other interrupts
- some have hidden tasks, not revealed for security reasons

OS designers reserve their right to use these interrupts for further OS developments.

Some interrupt numbers are classified as user interrupts (63-66h, F2h – FDh) being "oficially" unused for the moment. Users may write their own IHR and use these interrupt numbers for own purposes.

5.4. INTERRUPTS SPECIFIC INSTRUCTIONS

Instruction **INT** activates the handler corresponding to interrupt *n*.

- pushes the flags;
- pushes the returning FAR address;
- sets TF and IF flags to 0;
- calls the associated handler.

```
AdRTI dd        ...         ;we assume that AdRTI contains the handler's FAR address
    - - - - - - - -
        pushf                       ;push the flags
        push    cs                  ;save the segment of the returning address
        lea     ax,REV
        push    ax                  ;save the offset of the returning address
        xor     ax,ax
        push    ax          ;set all flags to 0 (for setting also TF and IF)
        popf
        jmp     AdRTI               ;jump to handler
REV: - - - - - - - -
```

Instruction **INTO** has the following effect:

- if OF = 1, then INTO is equiv to INT 4 (handler of INT 4 is called);
- if OF = 0, then equiv to NOP

Any instruction exposed to overflow is good to have a further INTO immediately after it:

```
        add   ax,b
        into        ;  do we have overflow ?...
```

CLI (Clear Interrupt) stops the acceptance of another interrupt. So, as long as IF=0 no interrupt will be handled by the CS. This instruction is typically used at the beginning of a handler to restrict issuing another interrupt while handling the current one.

STI (Set Interrupt) allows the processor to accept handling an interrupt. Usually it restores the effect induced by CLI.

Nonmaskable interrupts doesn't care about the IF flag value! The CS always reacts to a non-maskable interrupt !

IRET (no operands needed) performs a return from a handler procedure, being the last executed instruction. Its effect is converse to the INT instruction, namely:

- restores the FLAGS registers from the stack;
- transfers the control to the instruction which FAR address is found on top of the stack

The following sequence emulates IRET functioning:

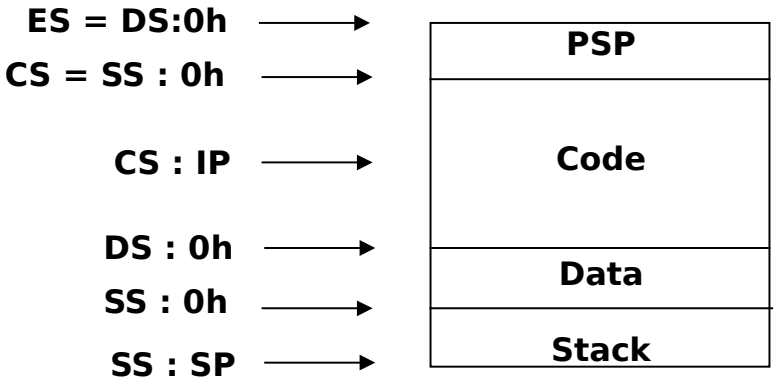
```
AdRev     dd        ?          ;temporarily stores the returning address
 - - - - - - - - - - - -
          popf                           ;restore the flags
          pop       word ptr AdRev       ;pops the offset of the returning address
          pop       word ptr AdRev+2     ;pops the segment of the returning address
          jmp       AdRev                ;indirect jump to the instruction following the
                                         ;interrupt request
```

Structure of PSP (Program Segment Prefix) – in memory !!

Offset	Length	Semantics
00h	2	Code of INT 20h - program termination
02h	2	Memory size in paragraphs (the end of memory occupied by the program)
04h	1	Reserved
05h	1	Code of INT 21h – DOS functions
06h	2	Available memory (nr of bytes) in the current segment
08h	2	Reserved
0Ah	4	FAR address of 22h IHR – provides the returning address from the current program
0Eh	4	FAR address of 23h IHR – used for restoring the old handler if the current program had redirected interrupt 23h (*CTRL+Break*)
12h	4	FAR address of 24h IHR - used for restoring the old handler if the current program had redirected interrupt 24h (*Critical error interrupt handler*)
16h	22	Reserved
2Ch	2	Segment address of DOS environment (MS-DOS system variables)
2Eh	46	Reserved
5Ch	32	FCB1 and FCB2, 16 bytes for each for accessing I/O standard files (kept for compatibility)
7Ch	4	Reserved
80h	1	Number of characters on command-line
81h	127	Command line – so it is possible to access command line parameters

5.5.2. Structure of an EXE program

An EXE program may have any number of code, data or stack segments. At any given run time moment we have an active code segment, an active data segment and an active stack segment.



Steps of loading a program into memory:

- **PSP is generated in memory**
- **the segments are placed in memory after PSP**
- **CS = address of the first active code segment**
- **IP = offset of the program's entry point (first instruction to be run from the CS segment)**
- **ES = DS = PSP starting address**
- **SS = stack segment address or SS = CS if there is no explicit stack segment defined**
- **SP = address of the stack's last word**

EXE Header (on disk!)

Any EXE file starts with a so called **EXE header** placed on disk in front of the actual file. Such a structure is necessary for being able to manage multiple segments, for correctly initialize the registers implicit values, for defining the current segments at program startup etc. The structure of the EXE header on disk is:

Offset	Size	Semantics
00h	2	EXE signature: 5A4Dh ('MZ' code – from “Mark Zbirkowski”)
02h	2	File size mod 512
04h	2	File size div 512
06h	**2**	**Total nr of relocatable items (NRI)**
08h	2	Header’s size in paragraphs (multiple of 16)
0Ah	2	Min. nr. of additionally required paragraphs (0000)
0Ch	2	Max. nr. of additionally required paragraphs(FFFE)
0Eh	2	Stack Segment address (paragraphs) from the program’s beginning
10h	2	SP initial value
12h	2	File’s control checksum
14h	2	IP initial value
16h	2	Code Segment address (paragraphs) from the program’s beginning
18h	**2**	**The offset of the relocation table (RT) (usual 1Ch)**
1Ah	2	Overlay
1Ch	?	Reserved
RT		**Offset1** | **Segment1** | **…** | **OffsetNRI** | **SegmentNRI**

Relocation

- **Relocatable items: values/operands specifying segment addresses ; ex:**

  mov ax, *data* or mov bx, *seg a*

- **The link-editor places in the instruction code (mov) the value which locates the corresponding segment.**

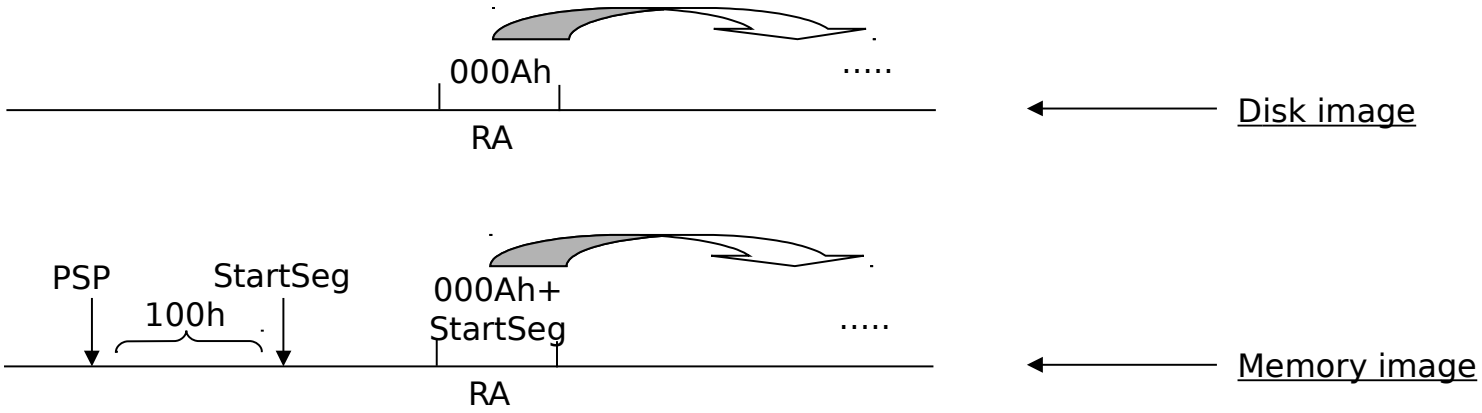
- **After loading the segments in memory, the segments starting address values must be adjusted by adding the "StartSeg" value.**

- **Relocation operation = updating the segment address values (operands) such that these correctly refer to the corresponding memory segment.**

- **For being able to perform the relocation process the EXE header offers:**
  - **at offset 06h – nr of relocatable items**
  - **at offset 18h – address of the relocation table (a FAR address for each segment name invocation)**

Loading in memory an EXE program:

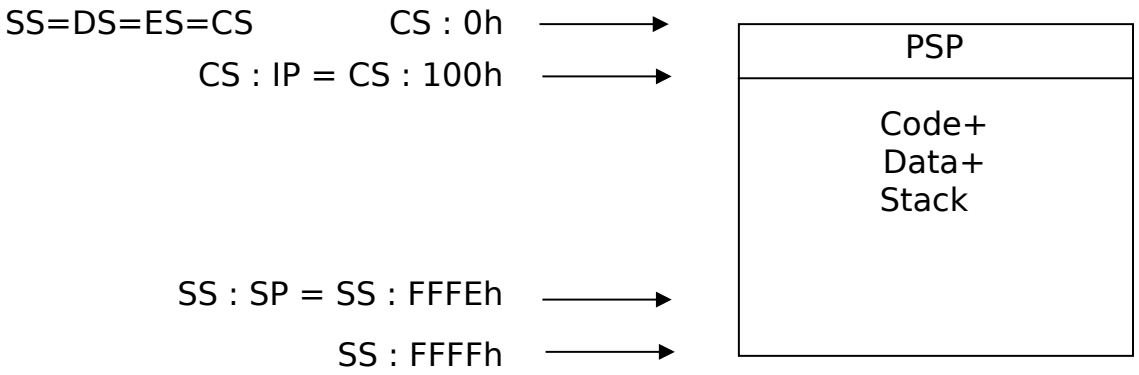
1. PSP (*Program Segment Prefix*) is created in memory - 256 bytes.
2. A starting address is chosen by the OS (loader). Usually this is the end of PSP:
   StartSeg := PSP address + 100h
3. The part of EXE file after the EXE header from disk is loaded into memory at StartSeg.
4. The relocation operation is performed: *for each relocation address RA (for each address where a relocatable item is present), to the contents of RA+StartSeg address the value StartSeg is added; in C notation this can be expressed as* [RA+StartSeg] += StartSeg



5. Depending on the values from the EXE header, the required additional memory is allocated (for heap), if possible.
6. The registers are initialized as follows:
   CS := CS relative (value from the EXE header) + StartSeg
   IP := IP initial (value from the EXE header)
   SS := SS relative (value from the EXE header) + StartSeg
   SP := SP initial value (value from the EXE header)
   ES := DS := PSP starting address

5.5.3. Structura unui program COM

A COM file has a very simple structure. It contains the binary image of the contents to be loaded in memory after the PSP for obtaining a runnable program. A COM file doesn't have a header on disk ! Only EXE files do… that is why they are called 'EXE headers' after all… isn't it ?



At a COM program loading time the PSP is created, all the segment registers are initialized as above and the control is given to the instruction found at offset 100h, that is to the first byte after the PSP. The stack pointer SP has the maximum value possible, that is FFFEh, coming from the fact that the program has 64 Kbytes and the stack grows to small addresses.

For a program to be a COM file several conditions have to be met :

- 1. The program has to be composed only from one single segment ; as a consequence the ASSUME directive has to be :

  assume cs:<name>,ds:<name>

  where <name> is the same for both CS and DS.

- 2. Immediately after the segment declaration, a directive :

  ORG 100h

  must be present indicating the fact that instructions and data start from address 100h. After ORG follows a labeled instruction which has to be the first executable instruction. The name of this label must be present in the END line of the program.

- 3. Data can be placed anywhere between instructions, the only condition being that instructions and data never to interfere (condition dependent only by programmer). For achieving this purpose the programmer must secure data areas by corresponding jump instructions.

- 4. Segment registers are automatically initialized, so the programmer must not load them explicitly anymore with some initial values .

- 5. COM programs must not contain relocatable items, that is operands representing segment names.

Assembling is done by ...>TASM C

Link editing is done by ...>TLINK C/T

An EXE program may be transposed in an equivalent COM file (if the above conditions are followed) by using the EXE2BIN DOS command:

...>EXE2BIN C.EXE C.COM

If the EXE program does not obey the rules imposed to a COM program and has for example referrals to relocatable items, TLINK will output the error:

“Fatal: Cannot generate COM file: segment relocatable items present”.