INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040 224172 TELETAX 040 224575 TELEX 460449 APH I

SMR/474 - 17

COLLEGE ON
"THE DESIGN OF REAL-TIME CONTROL SYSTEMS"
1 - 26 October

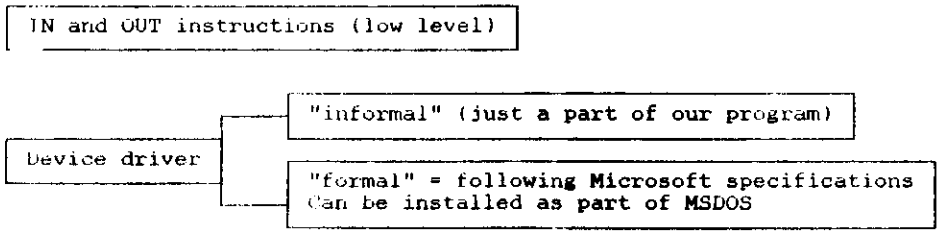
INTERFACING TO IBM-PCs.
Software

T. EGGARTER
Informatics Department
University of San Luis
San Luis
Argentina

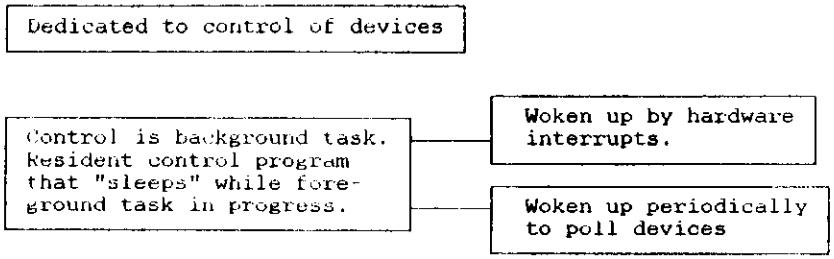
REAL TIME CONTROL USING A PC

DESIGN DECISIONS

a) How to talk with devices



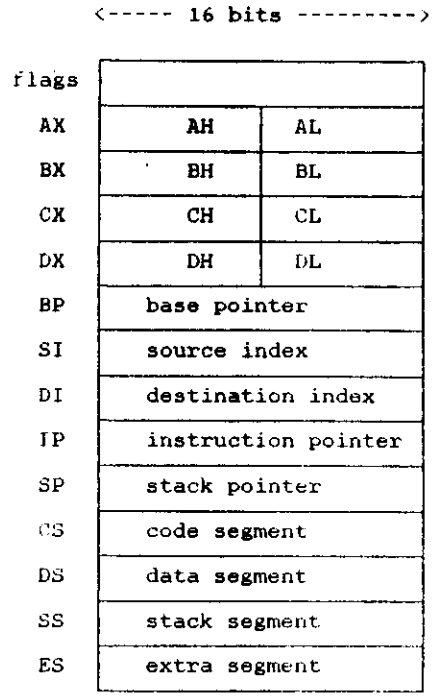
b) Use of PC



Design of appropriate software requires some familiarity with operation of PC in general and 8088 microprocessor in particular.

The first things a programmer will want to know about a microprocessor are:

a) What kind of data can it manipulate ? In other words, what are its registers ?



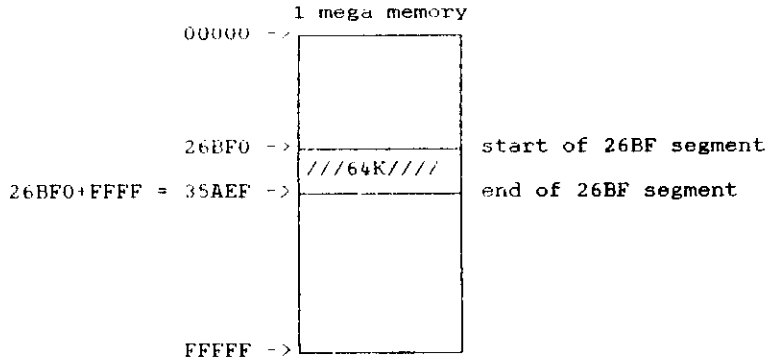
b) what can it do with these vales ? or, what is the instruction set ?

- There are around 100 instructions, many of these with several addressing modes.
- Their study is beyond the purpose of these lectures (see any text on 8088 assembly language).
- There are all the usual instructions MOV , CALL , ADD , JMP , ... available in 8 bit processors, plus others (for example string handling and integer multiplication and division).

SEGMENTED MEMORY ADDRESSING

A program is a set of instructions stored consecutively in memory . The IP keeps track of the address of the next instruction. But beeing only 16 bits long, the IP can sweep at most 64K memory locations. How, then, does the 8088 address 1Mb of memory ?.

Solution: the 8088 keeps a CS (code segment) register defining the 64K segment accessible by IP . For instance if CS contains the value 26BF , then the following memory area will be swept as IP varies:



Within this 64K segment IP defines an offset: IP = 0000 addresses the first location (26BF0) , IP = 0001 the next (26BF1) and so on until the last which corresponds to IP = FFFF .

A similar trick exists for data. The instruction MOV AX,[303] (move the value stored at location 303 into AX) uses DS (data segment) to define a 64K area where it will look for the value to load into AX. Within this segment the offset is 303.

And similarly for stack operations (PUSH , POP ..) : the values in SS (stack segment) and SP (stack pointer) are combined to give the desired address.

Notice that since there is a new 64K segment starting every 16 bytes, there are many different segment:offset pairs corresponding to the same memory location.

Consequences of segmentation:

- Programs with less than 64K of code can run without ever changing the value of CS .
- The relocation problem for such programs is trivial: to run it at another memory position just change the CS value.
- Same observation if all data fits in 64K or less: every piece of data can be referenced with only a 16 bit offset.
- From the above we see why the 64K barrier makes a difference for PC programs.
- In general instructions referring to memory addresses will come in two flavors: "near" (within the same segment) and "far" (segments need to be changed).
- For example CALL 55F0 is a call to a subroutine at address 55F0 within the current code segment (a near call). CALL 36A1:55F0 is an intersegment or far call. These two instructions have different lengths and push different return information on the stack.
- To return properly from the first call, the processor only has to get a 16 bit value of IP from the stack. In the second case it must get both an IP and a segment. Hence, there are two different returns with different opcodes.
- Pointers may also be near (16 bits, pointing to data within current data segment) and far (32 bits, one word for segment, another for offset).

COMMUNICATION BETWEEN THE 8088 AND THE "EXTERNAL WORLD"

- To exchange data with memory, the 8088 uses the MOV instruction.

- For any other device (keyboard, screen, disks, ...) it uses the IN and OUT instructions.

In both cases it is the 8088 who takes the initiative for the data transfer. Data is exchanged 8 bits at a time. An instruction like MOV AX,[BX+8] requires two memory accesses.

A third mechanism for interaction with the external world is the hardware interrupt. In this case it is an external device who takes the initiative and asks the 8088 for attention. The request passes first through another chip, the 8259A Programmable Interrupt Controller (PIC) which acts like a secretary: it analizes the request and decides if and when it should be passed to the 8088.

INTERRUPTS

Interrupts are numbered from 0 to 255 . For each possible interrupt there must be an appropriate service routine. The adresses of these service routines are stored in segment:offset form in a table starting at 0:0. It is called the INTERRUPT VECTOR TABLE.

When a request for, say, an INT 3 reaches the processor, it leaves the current task, gets the address of service routine 3 from this table, and starts executing it until it finds an IRET instruction (IRET = return from interrupt).

Although it is not an exact picture, you may think of an interrupt as a kind of long call whose destination address is found in the table. The call is triggered by an external request and therefore occurs at unpredictable times.

Apart from being requested by a device, an interrupt service routine can also be activated by a program. The corresponding instruction is INT n, for example INT 3 . This is called a "software interrupt". Its execution is immediate (there is no "secretary", the PIC isn't asked to approve the request).

Examples of hardware interrupts:

- Clock: uses interrupt 0 . Occurs approx. 18 times/sec.
- Keyboard: interrupt 1 each time a key is pressed or released.

Examples of software interrupts: later.

PRECAUTIONS WITH HARDWARE INTERRUPT SERVICING ROUTINES

Imagine a program executing the following code:

```
...
MOV AX,[1102]
OR AX,BX
...
```

and an interrupt arrives during the MOV instruction. The processor will jump to the service routine, execute it up to the first IRET, and then continue with the program at the OR instruction. Obviously the service routine must carefully save and restore all registers, or it will mess up the interrupted program.

Routines for software interrupts on the contrary may and usually do modify registers. No problem, they are called at well defined places of a program.

THE BASIC INPUT OUTPUT SYSTEM (BIOS)

Even trivial and rutinary tasks like reading a key need many IN and OUT instructions. IBM has provided standard routines in ROM for such tasks. The BIOS ROM area occupies adresses Fxxxx , or if you prefer, memory segment F000.

To use BIOS routines:

a) Load registers with parameters.

b) Execute an INT n with the appropriate n.

c) Recover data and arror information from registers.

All the BIOS routines (there are quite a few) are documented in the IBM manuals and in many books.

Examples:

```
INT 12H      ; no parameters, returns memory size (K) in AX.

MOV AH,4     ; parameter to read date
INT 1AH      ; returns CX=year, DH=month, DL=day
```

Advantages of this approach:

- Programmer doesn't have to learn how controller chips for devices work.
- Programs will work even if chips are changed in future PC models (IBM will just change the ROM routines accordingly).
- Relatively easy to modify PC's behavior. If you don't like how a device is treated, just write your own handler and change corresponding vector to point to it. (Vectors to BIOS routines are set up during boot phase but you may change them later if needed).

Bios routines do the most basic (low level) tasks. For example disks reads and writes by physical locations: you choose drive, head, track, sectors and memory buffer. There are no directories, no file structure.

EXPERIMENTING WITH BIOS CALLS

The following is a screen dump of a short session with the DEBUG program that comes with MSDOS. Advice: learn to use DEBUG, it really helps in understanding the machine.

Notice that program assembly begins at IP=100H . This is an MSDOS characteristic: the operating system reserves the first page of any loaded program for system information. It is called PSP (Program segment prefix).

```
C>debug
-a
2002:0100 int 12
2002:0102 mov ah,4
2002:0104 int 1a
2002:0106
-p

AX=0280  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2002  ES=2002  SS=2002  CS=2002  IP=0102   NV UP EI PL NZ NA PO NC
2002:0102 B404          MOV     AH,04
-p

AX=0480  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2002  ES=2002  SS=2002  CS=2002  IP=0104   NV UP EI PL NZ NA PO NC
2002:0104 CD1A          INT     1A
-p

AX=0019  BX=0000  CX=1990  DX=1016  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2002  ES=2002  SS=2002  CS=2002  IP=0106   NV UP EI PL NZ NA PO NC
2002:0106 C1            DB      C1
-q
```

CHANGING THE PC's BEHAVIOR

Suppose that you have built your own printer, but due to a slight miscalculation it only print every second letter it receives. You would like to change your PC so that it works correctly with it. Specifically, what the PC shold do is send every character twice in succession.

Notice: here we shall use some ugly tricks. Don't take this as an example to follow, we just want to illustrate what can be done.

From the documentation we see that INT 17H is the BIOS call to send a char to the printer. The char must be in AX with the printer number in DX. The INT 17H vector is in position 4*17H = 5C in the table. We enter debug and start looking:

```
C>debug
-d0:5c 5f
0000:0050                                              D2 EF 00 F0
```

Since the 8088 stores integer values in lowest-byte-first order the interrupt vector is F000:EFD2 .

Next we seek an unused entry in the vector table:

```
-d0:180 18f
0000:0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

There is one at address 0:180 . It corresponds to interrupt number 180H / 4 = 60H. We copy the original vector to this position:

```
-e0:180
0000:0180  00.d2   00.ef   00.00   00.f0   00.
```

Now an INT 60H will send a character to the printer: it activates the same BIOS service routine as INT 17H .

*** Warning ***: NEVER NEVER change interrupt vectors by hand as we do here. Imagine the consequences if we tried to mess around with the clock interrupt for example.

Next we assemble a few instructions:

```
-a
2002:0100 push ax
2002:0101 push dx
2002:0102 int 60
2002:0104 pop dx
2002:0105 pop ax
2002:0106 int 60
2002:0108 iret
2002:0109
-
```

This will be our service routine. It PUSHes the character and printer number on the stack, sends it to the printer (INT 60), then recovers the same character and printer number with POP , and sends it to the printer again. Voila !

Finally, we redirect the 17H vector (which everybody uses to access the printer) to our service routine:

```
-e0:5c
0000:005C  D2.00   EF.01   00.02   F0.20
-
```

Now a screen dump to the printer (Shift-PrtScr) looks like this:

```
--aa

22000022::00110000  ppuusshh  aaxx

22000022::00110011  ppuusshh  ddxx

22000022::00110022  iinntt  6600

22000022::00110044  ppoopp  ddxx

etc. etc.
```

After this we quickly turn off and reboot the PC. Leaving DEBUG would be quite dangerous. Can you tell why ?

MSDOS (or PCDOS as it is called by IBM) is the standard operating system for the PC . For our present purposes it is sufficient to know that it provides a wealth of "DOS Calls" to the programmer. In their use DOS calls are similar to BIOS calls: load registers with parameters and execute a given INT instruction.

The services provided by DOS are generally higher level than those of BIOS. Since DOS takes care of disk organization, it provides in particular all kinds of file oriented services. Many of these are accessed through INT 21H with various "function numbers" in AH .

Example:

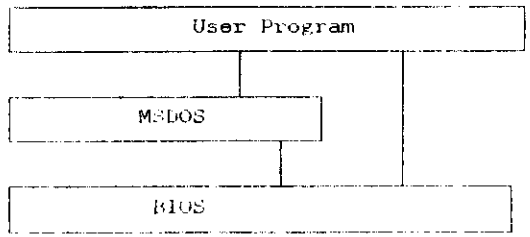
Open File:

```
   Input:  AH = 3DH
           AL = access mode (0=read , 1=write , 2=read/write)
        DS:DX = pointer to ASCIIZ path specification.

   Call : INT 21H

 returns : if successful, CF=0 and file handle in AX .
           if failure : CF=1 and error code in AX.
```

One of the DOS calls we shall use in the future is INT 21H . function 31H. This is the "Terminate and stay resident" call. It ends the execution of the calling program but does not release all the memory it occupies. Specifically:

```
   Input : AH = 31H
           AL = return code (as seen for instance by ERRORLEVEL if
                             program was called from batch file)
           DX = memory size in paragraphs to reserve.

   Call  : INT 21H

 returns : none (of course, the program does not continue)
```

Schematically the relation between a user program, MSDOS and BIOS may be viewed as follows:



The connecting lines mean that the upper box requests services from the lower box.

MSDOS DEVICES

Certain devices are handled by MSDOS as if they were files. For example the command

C> copy filea fileb

will copy the contents of filea to fileb. The completely analogous

C> copy filea prn

will send the contents of filea to the printer, while

C> copy filea con

will send the file to the screen.

This treatment of certain devices as files subsists in high level languages running under DOS. A Pascal or BASIC program that opens a file "PRN" and writes to it will effectively output data to the printer.

The nice thing is that MSDOS permits arbitrary user defined devices to "join the club" and be treated in the same way. This is handy for control problems. Imagine an instrument connected to the PC that can be read from any high level language by simple reading the file "MANOMETR", or controlling something by writing to a file "STEPMOTR".

Internally MSDOS keeps its devices in a linked list which it builds at bootup time. User installed devices are simply incorporated in the list.

The basic steps for installing a new device are:

1) Write and appropriate "driver" for the device. This is a program that establishes the actual communication with the device.

2) Put a "device=whatever.sys" line in confgig.sys. When you reboot, MSDOS will incorporate it in the device list together with CON, PRN, etc.

Of course everything has to be done according to strict rules.

- First, MSDOS admits "character" and "block" devices. Typical examples would be a terminal and a disk. We shall ONLY consider CHARACTER devices for simplicity.

- Second, your driver must consist of a header, a strategy routine and an interrupt routine. When accessing the device, MSDOS will call these routines in succession:

  a) it will execute a long call to the strategy routine passing in registers ES:BX a far pointer to a "request header" (don't confuse this with the device header!!!).

  In this step the request header is used by MSDOS to tell the driver what it wants to do.

  b) MSDOS calls the interrupt routine to do the required job. In this step the driver writes into the request header any information it wants to send back to MSDOS.

Notice: the name "interrupt routine" is misleading: it is called by MSDOS with a far call and ends with a far RET , not an IRET.

The format of the request header is:

REQUEST HEADER

offset	size	Function
0	1	Total length including possible data
1	1	never mind, not used in char devices
2	1	command code
3	2	status word
5	8	reserved for DOS
13	?	other data depending on the command

The command byte is used by DOS to signal what it wants. Examples:

command code	Desired function
0	INIT
4	INPUT
8	OUTPUT
9	OUTPUT WITH VERIFY
10	OUTPUT STATUS

There is a total of 13 different commands ranging from 0 to 12. For each of these, MSDOS expects the driver to do different things. For example with command 0 (INIT) it wants to know what part of the driver will stay in memory: the driver must put a far pointer to the end of its resident code in bytes 14 - 17 of the "other data" area. For an OUTPUT command it uses the "other data" area in a different way, and so on. We have no time to explore all the possibilities, look them up in a book when you need them.

Also, after each call to the interrupt routine, MSDOS expects to find the following bits in the status word:

STATUS WORD

bit	meaning
15	1 if error condition
14 .. 10	reserved
9	busy
8	done
7 .. 0	error number if bit 15 is 1

The driver header also has strict rules. Its length is 18 bytes. organized as follows:

DRIVER HEADER

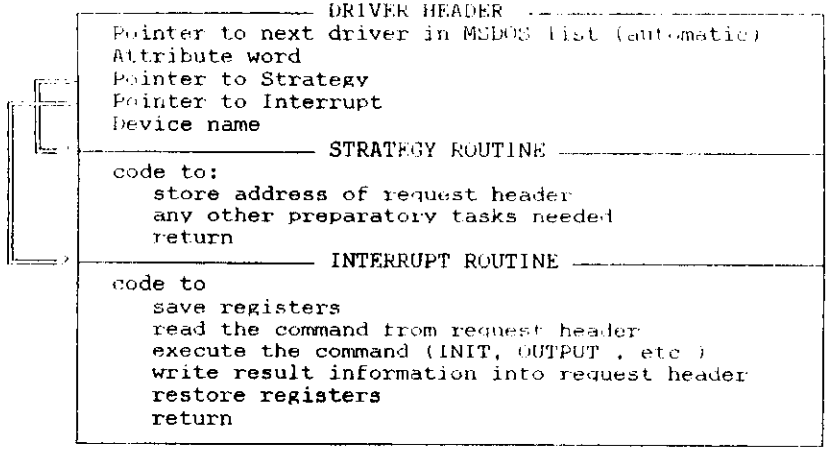
offset	size	contents
0	4	far pointer to next driver (MSDOS will fill it in if you put FFFFFFFF)
4	2	attributes of driver
6	2	offset of strategy routine
8	2	offset of interrupt routine
10	8	device name

The attribute word describes the device:

ATTRIBUTE WORD

bit	Meaning
15	1 if character device, 0 if block device
14 .. 3	look them up when needed
2	1 if current NUL device
1	1 if current stdout
0	1 if current stdin

Summarizing, the driver should look like this:



As a trivial example to see these things at work, here is a driver called NO_A.SYS which can be installed from by putting DEVICE=NO_A.SYS in the config.sys file.

It drives an output device called NOA (without the underscore). NOA is simply the screen, but with one little quirk: it does not print the letter A . When you install it, the computer acts like this:

```
C>type agreemen
   You should carefully read the following terms and conditions before opening
this diskette package. Opening this diskette package indicates your acceptance
of these terms and conditions. If you do not agree with them, you should
promptly return the package unopened, and your money will be refunded.

C>copy agreemen noa
   You should crefully red the following terms nd conditions before opening
this diskette pckge. Opening this diskette pckge indictes your cceptnce
of these terms nd conditions. If you do not gree with them, you should
promptly return the pckge unopened, nd your money will be retunded.
        1 File(s) copied

C>
```

The driver was produced from the following NO_A.ASM assembly program

```
CODE    SEGMENT
        ASSUME CS:CODE,DS:CODE

POINTER         DD      -1
ATTRIBUTE       DW      8000H     ; CHARACTER DRIVER
STRATEGY PTR    DW      STRATEGY
INTERRUPT PTR   DW      INTERRUPT
DRIVER NAME     DB      'NOA     '

OFFSET RH       DW      ?
SEGMENT RH      DW      ?

STRATEGY        PROC FAR
                MOV   OFFSET RH,BX
                MOV SEGMENT RH,ES
                RET
STRATEGY        ENDP

INTERRUPT       PROC FAR
                PUSH AX
                PUSH BX
                PUSH CX
                PUSH DX
                PUSH SI
                PUSH DI
                PUSH BP
                PUSH DS
                PUSH ES

                MOV BX,OFFSET RH
                MOV ES,SEGMENT RH

                MOV AL,ES:[BX+2]    ; GET COMMAND
                CMP AL,0
                JZ INIT
                CMP AL,8
                JZ OUTPUT
                CMP AL,9
                JZ OUTPUT
                JMP NORMAL EXIT

INIT:           MOV WORD PTR ES:[BX+14],OFFSET BOTTOM
                MOV   ES:[BX+16],CS
                JMP NORMAL EXIT
OUTPUT:         MOV CX,ES:[BX+18]
                MOV SI,ES:[BX+14]
                MOV DS,ES:[BX+16]
START LOOP:     MOV AL,[SI]
                CMP AL,'a'
                JZ NO OUTPUT
                CMP AL,'A'
                JZ NO OUTPUT
                MOV DL,AL
                MOV AH,2
                INT 21H
NO OUTPUT:      INC SI
                LOOP START LOOP

NORMAL EXIT:    MOV WORD PTR ES:[BX+3],0100H  ;STATUS
                POP ES
                POP DS
                POP BP
                POP DI
                POP SI
                POP DX
                POP CX
                POP BX
                POP AX
                RET
INTERRUPT       ENDP
BOTTOM          LABEL   WORD
CODE            ENDS
                END
```

The steps to arrive at a NO_A.SYS file are:

```
C>TASM NO A                    produces .OBJ file
C>LINK NO A                    produces .EXE file
C>EXE2BIN NO A.EXE NO_A.SYS    produces driver
```

To see the result, here is also a DEBUG disassembly of NO_A.SYS (loaded at offset 0 within its code segment):

```
-d0L20
200D:0000  FF FF FF FF 00 80 16 00-1F 00 4E 4F 41 20 20 20   ..........NOA
200D:0010  20 20 00 00 00 00 89 1E-12 00 8C 06 14 00 CB 50   ............P
-u16L70
200D:0016 891E1200      MOV     [0012],BX
200D:001A 8C061400      MOV     [0014],ES
200D:001E CB            RETF
200D:001F 50            PUSH    AX
200D:0020 53            PUSH    BX
200D:0021 51            PUSH    CX
200D:0022 52            PUSH    DX
200D:0023 56            PUSH    SI
200D:0024 57            PUSH    DI
200D:0025 55            PUSH    BP
200D:0026 1E            PUSH    DS
200D:0027 06            PUSH    ES
200D:0028 8B1E1200      MOV     BX,[0012]
200D:002C 8E061400      MOV     ES,[0014]
200D:0030 26            ES:
200D:0031 8A4702        MOV     AL,[BX+02]
200D:0034 3C00          CMP     AL,00
200D:0036 740B          JZ      0043
200D:0038 3C08          CMP     AL,08
200D:003A 7414          JZ      0050
200D:003C 3C09          CMP     AL,09
200D:003E 7410          JZ      0050
200D:0040 EB2D          JMP     006F
200D:0042 90            NOP
200D:0043 26            ES:
200D:0044 C7470E7F00    MOV     WORD PTR [BX+0E],007F
200D:0049 26            ES:
200D:004A 8C4F10        MOV     [BX+10],CS
200D:004D EB20          JMP     006F
200D:004F 90            NOP
200D:0050 26            ES:
200D:0051 8B4F12        MOV     CX,[BX+12]
200D:0054 26            ES:
200D:0055 8B770E        MOV     SI,[BX+0E]
200D:0058 26            ES:
200D:0059 8E5F10        MOV     DS,[BX+10]
200D:005C 8A04          MOV     AL,[SI]
200D:005E 3C61          CMP     AL,61
200D:0060 740A          JZ      006C
200D:0062 3C41          CMP     AL,41
200D:0064 7406          JZ      006C
200D:0066 8AD0          MOV     DL,AL
200D:0068 B402          MOV     AH,02
200D:006A CD21          INT     21
200D:006C 46            INC     SI
200D:006D E2ED          LOOP    005C
200D:006F 26            ES:
200D:0070 C747030001    MOV     WORD PTR [BX+03],0100
200D:0075 07            POP     ES
200D:0076 1F            POP     DS
200D:0077 5D            POP     BP
200D:0078 5F            POP     DI
200D:0079 5E            POP     SI
200D:007A 5A            POP     DX
200D:007B 59            POP     CX
200D:007C 58            POP     AX
200D:007D 58            POP     AX
200D:007E CB            RETF
```

As a final step, let us look directly at the MSDOS list of devices:

```
C>debug
-a
200C:0100 mov ah,52
200C:0102 int 21
200C:0104
-g 104

AX=5200  BX=0026  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=200C  ES=02BC  SS=200C  CS=200C  IP=0104   NV UP EI PL NZ NA PO NC
```

Function 52H is an undocumented DOS function which returns a "list of lists" address in ES:BX . 22H bytes later is the start of the first device driver. Let us look at it. (Remark : 26+22=48, even in hexa)

```
-des:48 L 12
02BC:0040                               00 00 07 0F 04 80 C6 14          ........
02BC:0050  CC 14 4E 55 4C 20 20 20-20 20                        ..NUL
```

The first device is NUL . This is always the case, no matter what you install later or in what order. Note that the device name is left justified and padded with blanks.

```
-dt07:0 L 12
0F07:0000  00 00 3A 0E 00 80 16 00-1F 00 4E 4F 41 20 20 20   ..:.......NOA
0F07:0010  20 20
-de3a:0 L 12
0E3A:0000  00 00 C7 0A 00 80 16 00-21 00 45 47 41 24 20 20   ........!.EGA$
0E3A:0010  20 20
-dac7:0 L 12
0AC7:0000  00 00 56 0A 00 C8 A2 00-AD 00 53 4D 41 52 54 41   ..V.......SMARTA
0AC7:0010  41 52                                             AR
```

The PC is strictly a monotasking machine. However there is a proliferation of programs which allow the user to do something else besides running his current application. There are calculators, spelling checkers, agendas, etc. etc. which can run "simultaneously" with other tasks. These are TSR (Terminate and Stay Resident) programs.

A TSR is loaded into memory by running it, just like any other program. It has an initialization phase in which it prepares a few things, and then exits without releasing the memory it is uses. From then on it just sits in memory until some specific event (frequently the touch of a "hot key") transfers control to it again.

Neither IBM norMicrosoft provide technical support for TSR programming. In this sense the subject is quite different from, say, drivers. In TSR whatever you do is at your own risk, the only way to learn is from other users and from your own experience.

To tell MSDOS that it has finished, a program must call an exit procedure. For example

Interrupt 21H, function 0 (old method) :

```
MOV AH,0      ;  function 0
INT 21H
```

or interrupt 21H, function 4CH (new method, recommended)

```
MOV AL,10H    ; value the program returns
MOV AH,4CH    ; function 4CH
INT 21H
```

Both methods end execution and consider that the the program's memory is free and can be used for something else. But there is another exit call:

```
MOV AL,10     ; returned value
MOV DX,110    ; memory to reserve in paragraphs
MOV AH,31H    ; function 31H
INT 21H
```

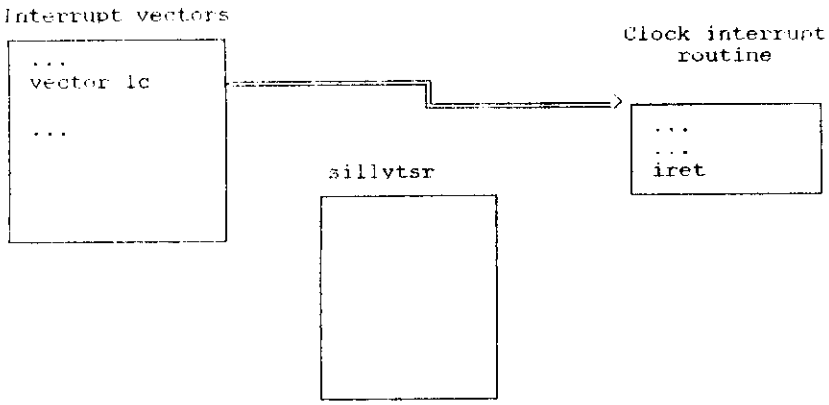
In this case execution also ends, but the amount of memory specified in DX remains protected, no other program can use it.

Let us write our first resident program: SILLYTSR.COM . It does nothing, it just sits in memory using up space. But it has the advantage of being simple and short.

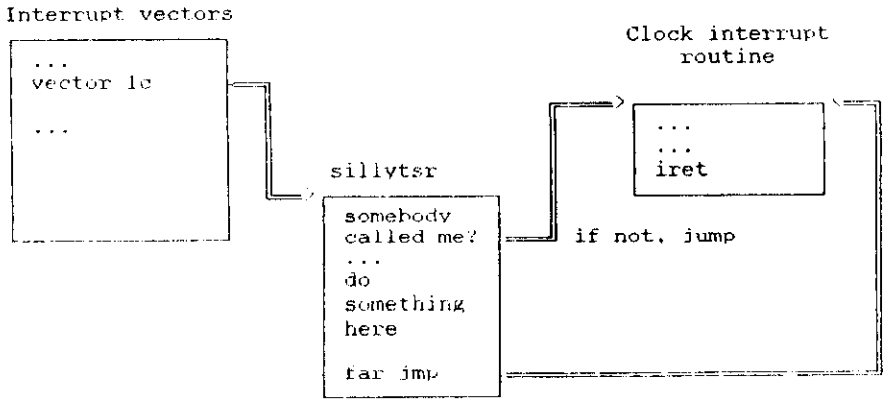
```
C>debug
-a
1FFD:0100 mov dx,100
1FFD:0103 mov ah,31
1FFD:0105 int 21
1FFD:0107
-nsillytsr.com
-rdx
DX 0000
:
-rcx
CX 0000
:8
-w
Writing 0008 bytes
-q
C>chkdsk
Volume DISK1 VOL1  created 23 Sep 1987 15:31

 22167552 bytes total disk space
    45056 bytes in 3 hidden files
   145408 bytes in 44 directories
 20787200 bytes in 831 user files
    40960 bytes in bad sectors
  1148928 bytes available on disk

   655360 bytes total memory
   540240 bytes free


C>sillytsr

C>chkdsk
Volume DISK1 VOL1  created 23 Sep 1987 15:31

 22167552 bytes total disk space
    45056 bytes in 3 hidden files
   145408 bytes in 44 directories
 20787200 bytes in 831 user files
    40960 bytes in bad sectors
  1148928 bytes available on disk

   655360 bytes total memory
   535888 bytes free


C>pcmap
PCMAP 1.0 (c) 1987, Ziff-Davis Publishing Corp.

Segment  Paragraphs    Program
142E        00E9        COMMAND.COM
1519        0004        (Free)
1524        0604        EGAPRTSC.COM
1C1B        010E        SILLYTSR.COM
1B34        00E8        (Unknown)
1D2B        82E3        PCMAP.COM

C>
```

A useful TSR program must have some mechanism to wake it up when needed. Ways of doing this:

a) Take an often accessed interrupt vector like the clock (interrupt 1c). The situation of our SILLYTSR program in memory is like this:



What our program could have done before exiting with the 31H function is to set up the vectors like this:

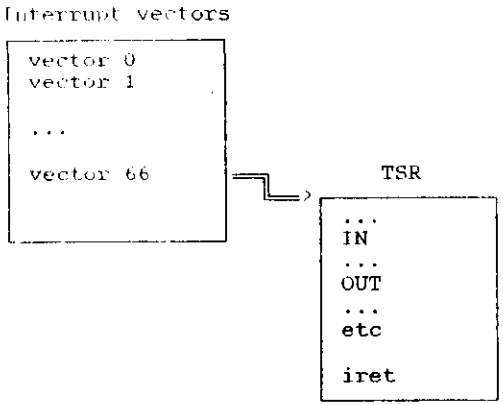


In this way the resident program gets into control periodically and can decide if it is its turn to do something or not.

For example, to read an instrument every second a TSR could keep a counter and perform the measurement every 18th time it is called.

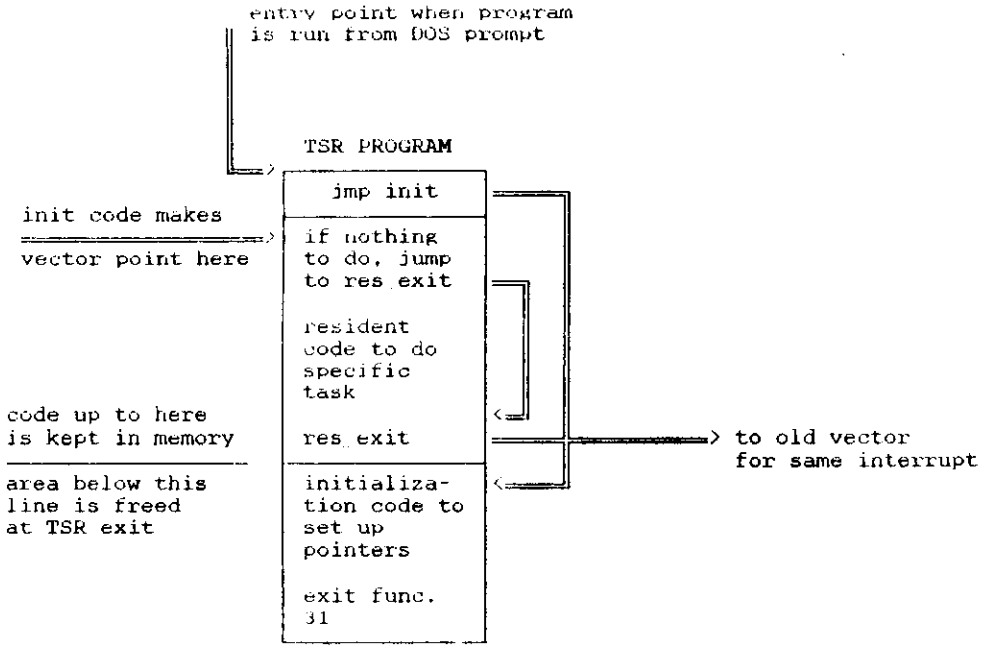
b) if there is a hardware interrupt associated with each call of the TSR then we would of course hook the program to the corresponding vector.

Example: an instrument which activates INT 66 (now unused) each time it requires attention. We would setup things this way



A particular case of hardware activated TSRs are those woken up by a "hot key". Each time a key is pressed or released it produces an INT 9. The TSR can sneak into the chain of events associated with vector 9 and have a look at the keystroke before any other program gets a chance to read (and thereby destroy) it.

A reasonable way to organize a TSR program is:



To illustrate the ideas, here is a little example. It just keeps putting a letter "Z" in the middle of the screen about once a second. However it is clear that useful things (like displaying a clock in a corner of the screen) can be done in the same way.

Notice: the left half is an actual debug session while the right part has explanatory comments.

```
C>debug
-a
1FFD:0100 jmp 128                  to initialization code
1FFD:0102 nop                      counter, every 15 clock ticks put "Z"
                                   RESIDENT PART STARTS AT NEXT LINE
1FFD:0103 push ds                  save to restore later
1FFD:0104 push cs
1FFD:0105 pop ds                   make ds=cs
1FFD:0106 inc byte [102]           inc counter
1FFD:010A and byte [102],0F        take lowest four bytes
1FFD:010F jnz 120                  and do nothing unless they are zero
1FFD:0111 push ax                  save to restore later
1FFD:0112 mov ax,b800              start screen memory. In B&W use b000
1FFD:0115 push ax
1FFD:0116 pop ds                   start screen memory in ds
1FFD:0117 mov ax,2f5a              5a="Z",2f=white letter on green bkgd
1FFD:011A mov [7c0],ax             center screen=offset 7c0 from start
1FFD:011D pop ax                   restore ax
1FFD:011E nop
1FFD:011F nop
1FFD:0120 pop ds                   restore ds
1FFD:0121 jmp 0:0                  this is res exit.
                                  (init will change 0:0 -> old vector)
1FFD:0126 nop
1FFD:0127 nop                      INIT CODE STARTS ON NEXT LINE
1FFD:0128 mov al,1c
1FFD:012A mov ah,35
1FFD:012C int 21                   get old 1c=clock vector
1FFD:012E mov [122],bx             change the 0:0 in res exit ..
1FFD:0132 mov [124],es             ...into old vector
1FFD:0136 mov dx,103
1FFD:0139 mov al,1c
1FFD:013B mov ah,25
1FFD:013D int 21                   install new 1c vector
1FFD:013F mov dx,20                keep 20 paragraphs (just in case)
1FFD:0142 mov ah,31
1FFD:0144 int 21                   and exit with TSR call
1FFD:0146
-rbx
BX 0000                            prepare registers to save
:0
-rcx
CX 0000
:50
-nztsr.com                         name the program
-w                                 save it
Writing 0050 bytes
-q
```

Convince yourself: enter the program and the run it from the DOS prompt with

```
C>ZTSR <enter>
```

Problems:

- When your TSR gets control, interrupts are disabled. A service routine should quickly reenable them, or the machine will loose clock ticks or even suffer some mayor disaster. How does a TSR perform longer tasks ? .

  Imagine a TSR in the clock vector. It can not simply enable interrupts and then do its work: it would "interrupt itself" and never complete anything.

  Possible solutions: a) keep a "TSR Active" flag in the program. Service the old clock routine whenever there is nothing to do or the flag is set. b) restore the old clock vector during the execution of the slow part of the TSR, put back the new one when finished.

- MSDOS is non-reentrant. We may have a problem if MSDOS is doing something, an interrupt activates the TSR which in turn requests a DOS service. Solution (undocumented, from gossip): there is an InDos flag in memory, set when MSDOS is doing something. To get its address use INT 21H, function 34H. At exit registers ES:BX give the address. The next byte (ES:BX+1) is used for the same purpose. To play it safe check both.

  Some "hot key" programs do the following. They attach themselves to the keyboard and clock routines. Then

```
      On keyboard interrupts : Set a "Service Required" flag

         On clock interrupts : if Service Required and not InDos {
                                     do job
                                     Clear Service Required
                               }
```

- I/O errors in the TSR should not send control back to DOS. this would kill the foreground program. Usually the critical error handler (INT 24H, the one that gives Abort, Retry or Ignore ?) is shortcircuited by an IRET while the TSR is active.

  What happens when the users presses Ctrl-C while in the TSR ? And the list of potential problems could be extended.

SUGGESTED REFERENCES

Handbook of Software and Hardware Interfacing for the IBM PCs, Jeffrey P. Royer, Prentice Hall.

Various books on the IBM PC by Peter Norton. These are "classics" in the field.

Advanced MS DOS Expert Techniques for Programmers, Carl Townsend, Howard W. Sams & Company. Shows how to write a TSR Program entirely in C .

BYTE March 87: There is an article by Brian Edginton "Installing Memory Resident Programs in C"