Assembly Language Programming
Lecture Notes

Volume 2 – Advanced Topics

Delivered by
Belal Hashmi

Compiled by
Junaid Haroon

Table of Contents

1
Software Interrupts

1.1. INTERRUPTS

Interrupts in reality are events that occurred outside the processor and the processor must be informed about them. Interrupts are asynchronous and unpredictable. Asynchronous means that the interrupts occur, independent of the working of the processor, i.e. independent of the intstruction currently executing. Synchronous events are those that occur side by side with another activity. Interrupts must be asynchronous as they are generated by the external world which is unaware of the happenings inside the processor. True interrupts that occur in real time are asynchronous with the execution. Also it is unpredictable at which time an interrupt will come. The two concepts of being unpredictable and asynchronous are overlapping. Unpredicable means the time at which an interrupt will come cannot be predicted, while asynchronous menas that the interrupt has nothing to do with the currently executing instruction and the current state of the processor.

The 8088 processor divides interrupts into two classes. Software interrupts and hardware interrupts. Hardware interrupts are the real interrupts generated by the external world as discussed above. Software interrupts on the contrary are not generated from outside the processor. They just provide an extended far call mechanism. Far all allows us to jump anywhere in the whole megabyte of memory. To return from the target we place both the segment and offset on the stack. Software interrupts show a similar behavior. It however pushes one more thing before both the segment and offset and that is the FLAGS register. Just like the far call loads new values in CS and IP, the interrupt call loads new values in CS, IP, and FLAGS. Therefore the only way to retain the value of original FLAGS register is to push and pop as part of interrupt call and return instructions. Pushing and popping inside the routine will not work as the routine started with an already tampered value.

The discussion of real time interrupts is deferred till the next chapter. They play the critical part in control applications where external hardware must be control and events and changes therein must be appropriately responded by the processor. To generate an interrupt the INT instruction is used. The routine that executes in response to an INT instruction is called the interrupt service routine (ISR) or the interrupt handler. Taking example from real time interrupts the routine to instruct an external hardware to close the valve of a boiler in response to an interrupt from the pressure sensor is an interrupt routine.

The software interrupt mechanism in 8088 uses vectored interrupts meaning that the address of the interrupt routine is not directly mentioned in an interrupt call, rather the address is lookup up from a table. 8088 provides a mechanism for mapping interrupts to interrupt handlers. Introducing a new entry in this mapping table is called hooking an interrupt.

Syntax of the INT instruction is very simple. It takes a single byte argument varying from 0-255. This is the interrupt number informing the processor, which interrupt is currently of interest. This number correlates to the interrupt handler routine by a routing or vectoring mechanism. A few interrupt numbers in the start are reserved and we generally do not use them. They are related to the processor working. For example INT 0 is the divide by zero interrupt. A list of all reserved interrupts is given later. Such interrupts are programmed in the hardware to generate the designated interrupt when the specified condition arises. The remaining interrupts are provided by the processor for our use. Some of these were reserved by the IBM PC designers to interface user programs with system software like DOS and BIOS. This was the logical choice for them as interrupts provided a very flexible architecture. The remaining interrupts are totally free for use in user software.

The correlation process from the interrupt number to the interrupt handler uses a table called interrupt vector table. Its loation is fixed to physical memory address zero. Each entry of the table is four bytes long containing the segment and offset of the interrupt routine for the corresponding interrupt number. The first two bytes in the entry contain the offset and the next two bytes contain the segment. The little endian rule of putting the more significant part (segment) at a higher address is seen here as well. Mathematically offset of the interrupt n will be at nx4 while the segment will be at nx4+2. One entry in this table is called a vector. If the vector is changed for interrupt 0 then INT 0 will take execution to the new handler whose address is now placed at those four bytes. INT 1 vector occupies location 4, 5, 6, and 7 and similarly vector for INT 2 occupies locations 8, 9, 10, and 11. As the table is located in RAM it can be changed anytime. Immediately after changing it the interrupt mapping is changed and now the interrupt will result in execution of the new routine. This indirection gives the mechanism extreme flexibility.

The operation of interrupt is same whether it is the result of an INT instruction (software interrupt) or it is generated by an external hardware which passes the interrupt number by a different mechanism. The currently executing instruction is completed, the current value of FLAGS is pushed on the stack, then the current code segment is pushed, then the offset of the next instruction is pushed. After this it automatically clears the trap flag and the interrupt flag to disallow further interrupts until the current routine finishes. After this it loads the word at nx4 in IP and the word at nx4+2 in CS if interrupt n was generated. As soon as these values are loaded in CS and IP execution goes to the start of the interrupt handler. When the handler finishes its work it uses the IRET instruction to return to the caller. IRET pops IP, then CS, and then FLAGS. The original value of IF and TF is restored which reenables further interrupts. IF and TF will be discussed in detail in the discussion of real time interrupts. We have discussed three things till now.

1. The INT and IRET instruction format and syntax
2. The formation of IVT (interrupt vector table)
3. Operation of the processor when an interrupt in generated

Just as discussed in the subroutines chapter, the processor will not match interrupt calls to interrupt returns. If a RETF is used in the end of an ISR the processor will still return to the caller but the FLAGS will remain on the stack which will destroy the expectations of the caller with the stack. If we know what we are doing we may use such different combination of instructions. Generally we will use IRET to return from an interrupt routine. Apart from indirection the software interrupt mechanism is similar to CALL and RET. Indirection is the major difference.

The operation of INT can be written as:

- sp ← sp+2
- [sp] ← flag
- sp ← sp+2
- if ← 0
- tf ← 0
- [sp] ← cs
- sp ← sp+2
- [sp] ← ip

- ip ← [0:N*4]
- cs ← [0:N*4+2]

The operation of IRET can be written as:

- ip ← [sp]
- sp ← sp-2
- cs ← [sp]
- sp ← sp-2
- flag ← [sp]
- sp ← sp-2

The above is the microcode description of INT and IRET. To obey an assembly language instruction the processor breaks it down into small opertions. By reading the microcode of an instruction its working can be completely understood.

The interrupt mechanism we have studied is an extended far call mechanism. It pushes FLAGS in addition to CS and IP and it loads CS and IP with a special mechanism of indirection. It is just like the table of contents that is located at a fixed position and allows going directly to chapter 3, to chapter 4 etc. If this association is changed in the table of contents the direction of the reader changes. For example if Chapter 2 starts at page 220 while 240 is written in the table of contents, the reader will go to page 240 and not 220. The table of contents entry is a vector to point to map the chapter number to page number. IVT has 256 chapters and the interrupt mechanism looks up the appropriate chapter number to reach the desired page to find the interrupt routine.

Another important similarlity is that table of contents is always placed at the start of the book, a well known place. Its physical position is fixed. If some publishers put it at some place, others at another place, the reader will be unable to find the desired chapter. Similarly in 8088 the physical memory address zero is fixed for the IVT and it occupies exactly a kilobyte of memory as the 256x4=1K where 256 is the number of possible interrupt vectors while the size of one vector is 4 bytes.

Interrupts introduce temporary breakage in the program flow, sometimes programmed (software interrupts) and unprogrammed at other times (hardware interrupts). By hooking interrupts various system functionalities can be controlled. The interrupts reserved by the processor and having special functions in 8088 are listed below:

- INT 0, Division by zero
  Menaing the quotient did not fit in the destination register. This is a bit different as this interrupt does not return to the next instruction, rather it returns to the same instruction that generated it, a DIV instruction ofcourse. Here INT 0 is automatically generated by a DIV when a specific situation arises, there is no INT 0 instruction.
- INT 1, Trap, Single step Interrupt
  This interrupt is used in debugging with the trap flag. If the trap flag is set the Single Step Interrupt is generated after every instruction. By hooking this interrupt a debugger can get control after every instruction and display the registers etc. 8088 was the first processor that has this ability to support debugging.
- INT 2, NMI-Non Maskable Interrupt
  Real interrupts come from outside the proessor. INT 0 is not real as it is generated from inside. For real interrupts there are two pins in the processor, the INT pin and the NMI pin. The processor can be directed to listen or not to listen to the INT pin. Consider a recording studio, when the recording is going on, doors are closed so that no interruption occurs, and when there is a break, the doors are opened so that if someone is waiting outside can come it. However if there is an urgency like fire outside then the door must be broken and the recording must not be catered for. For such situations is the NMI pin

which informs about fatal hardware failures in the system and is tied to interrupt 2. INT pin can be masked but NMI cannot be masked.

- INT 3, Debug Interrupt
  The only special thing about this interrupt is that it has a single byte opcode and not a two byte combination where the second byte tells the interrupt number. This allows it to replace any instruction whatsoever. It is also used by the debugger and will be discussed in detail with the debugger working.
- INT 4, Arithmetic Overflow, change of sign bit
  The overflow flag is set if the sign bit unexpectedly changes as a result of a mathemcial or logical instruction. However the overflow flag signals a real overflow only if the numbers in question are treated as signed numbers. So this interrupt is not automatically generated but as a result of a special instruction INTO (interrupt on overflow) if the overflow flag is set. Otherwise the INTO instruction behaves like a NOP (no operation).

These are the five interrupts reserved by Intel and are generally not used in our operations.

1.2. HOOKING AN INTERRUPT

To hook an interrupt we change the vector corresponding to that interrupt. As soon as the interrupt vector changes, that interrupt will be routed to the new handler. Our first example is with the divide by zero interrupt. The normal system defined behavior in response to divide by zero is to display an error message and terminate the program. We will change it to display our own message.

Example 8.1

```
; hooking divide by zero interrupt
[org 0x0100]
              jmp  start

message:      db   'You divided something by zero.', 0

;;;;; COPY LINES 028-050 FROM EXAMPLE 7.4 (strlen) ;;;;;
;;;;; COPY LINES 005-024 FROM EXAMPLE 7.1 (clrscr) ;;;;;
;;;;; COPY LINES 050-090 FROM EXAMPLE 7.4 (printstr) ;;;;;

; divide by zero interrupt handler
myisrfor0:    push ax                 ; push all regs
              push bx
              push cx
              push dx
              push si
              push di
              push bp
              push ds
              push es

              push cs
              pop  ds                 ; point ds to our data segment

              call clrscr             ; clear the screen
              mov  ax, 30
              push ax                 ; push x position
              mov  ax, 20
              push ax                 ; push y position
              mov  ax, 0x71           ; white on blue attribute
              push ax                 ; push attribute
              mov  ax, message
              push ax                 ; push offset of message
              call printstr           ; print message

              pop  es
              pop  ds
              pop  bp
```

```
120                 pop  di
121                 pop  si
123                 pop  dx
124                 pop  cx
125                 pop  bx
126                 pop  ax
127                 iret                    ; return from interrupt
128
129    ; subroutine to generate a divide by zero interrupt
130    genint0:     mov  ax, 0x8432         ; load a big number in ax
131                 mov  bl, 2              ; use a very small divisor
132                 div  bl                 ; interrupt 0 will be generated
133                 ret
134
135    start:       xor  ax, ax
136                 mov  es, ax             ; load zero in es
137                 mov  word [es:0*4], myisrfor0 ; store offset at n*4
138                 mov  [es:0*4+2], cs     ; store segment at n*4+2
139                 call  genint0           ; generate interrupt 0
140
141                 mov ax, 0x4c00          ; terminate program
142                 int 0x21
```

93-101	We often push all registers in an interrupt service routine just to be sure that no unintentional modification to any register is made. Since any code may be interrupted an unintentional modification will be hard to debug
103-104	Since interrupt can be called from anywhere we are not sure about the value in DS so we reset it to our code segment.

When this program is executed our desired message will be shown instead of the default message and the computer will hang thereafter. The first thing to observe is that there is no INT 0 call anywhere in the code. INT 0 was invoked automatically by an internal mechanism of the processor as a result of the DIV instruction producing a result that cannot fit in the destination register. Just by changing the vector we have changed the response of the system to divide overflow situations.

However the system stuck instead of returning to the next instruction. This is because divide overflow is a special type of interrupt that returns to the same instruction instead of the next instruction. This is why the defaut handler forcefully terminates the program instead of returning. Now the IRET will take control back to the DIV instruction which will again generate the same interrupt. So the computer is stuck in an infinite loop.

1.3. BIOS AND DOS INTERRUPTS

In IBM PC there are certain interrupts designated for user programs to communicate with system software to access various standard services like access to the floppy drive, hard drive, vga, clock etc. If the programmer does not use these services he has to understand the hardware details like which particular controller is used and how it works. To avoid this and provide interoperability a software interface to basic hardware devices is provided except in very early computers. Since the manufacturer knows the hardware it burns the software to control its hardware in ROM. Such software is called firmware and access to this firmware is provided through specified interrupts.

This basic interface to the hardware is called BIOS (basic input output services). When the computer is switched on, BIOS gets the control at a specified address. The messages at boot time on the screen giving BIOS version, detecting different hardware are from this code. BIOS has the responsibility of testing the basic hardware including video, keyboard, floppy drive, hard drive etc and a special program to bootstrap. Bootstrap means to load OS from hard disk and from there OS takes control and proceeds to load its components and display a command prompt in the end. There are two important programs; BIOS and OS. OS services are high level and build upon the BIOS services. BIOS services are very low level. A level further lower is only directly controlling the hardware. BIOS services provide a hardware independent layer above the hardware and OS services provide another higher level layer over the BIOS services. We have practiced direct hardware access with the video device directly without using BIOS or DOS. The layer of BIOS provides services like display a character, clear the screen, etc. All these layers are optional in that we can skip to whatever lower layer we want.

The most logical way to provide access to firmware is to use the interrupt mechanism. Specific services are provided at specific interrupts. CALL could also have been used but in that case every manufacturer would be required to place specific routines at specific addresses, which is not a flexible mechanism. Interrupts provide standard interrupt number for the caller and flexibility to place the interrupt routine anywhere in the memory for the manufacturer. Now for the programmer it is decided that video services will be provided at INT 10 but the actual address of the video services can and do vary on computers from different manufacturers. Any computer that is IBM compatible must make the video services accessible through INT 10. Similarly keyboard services are available at INT 16 and this is standard in every IBM compatible. Manufacturers place the code wherever they want and the services are exported through this interrupt.

BIOS exports its various services through different interrupts. Keyboard services are exported through INT 16, parallel port services through INT 17 and similarly others through different interrupts. DOS has a single entry point through INT 21 just like a pin hole camer, this single entry points leads to a number of DOS services. So how one interrupt provides a number of different services. A concept of service number is used here which is a defecto standard in providing multiple services through an interrupt. INT 10 is for video services and each of character printing service, screen clearing service, cursor movement service etc. has a service number associated to it. So we say INT 10 service 0 is used for this purpose and INT 10 service 1 is used for that purpose etc. Service numbers for different standard services are also fixed for every IBM compatible. The concept of exported services through interrupts is expanded with the service numbering scheme.

The service number is usually given in the AH register. Sometimes these 256 services seem less. For example DOS exports thousands of services. So will be often see an extension to a level further with subservices. For examples INT 10 character generator services are all provided through a single service number and the services are distinguished with a subservice number.

The finally selected service would need some arguments for it to work. In interrupts arguments are usually not given through stack, rather registers are used. The BIOS and DOS specifications list which register contains which argument for a particular service of a particular interrupt.

We will touch some important BIOS and DOS services and not cover it completely neither is it possible to cover it in this space. A very comprehensive reference of interrupts is the Ralph Brown List. It is just a reference and not to be studied from end to end. All interrupts cannot be rememberd and there is no need to remember them.

The service number is almost always in AH while the subservice number is in AL or BL and sometimes in other registers. The documentation of the service we are using will list which register should hold what when the interrupt is invoked for that particular service.

Our first target using BIOS is video so let us proceed to our first program that uses INT 10 service 13 to print a string on the screen. BIOS will work even if the video memory is not at B8000 (a very old video card) since BIOS knows everything about the hardware and is hardware specific.

Example 8.2

```
001  ; print string using bios service
002  [org 0x0100]
003                jmp  start
004  message:      db   'Hello World'
005
006  start:        mov  ah, 0x13           ; service 13 - print string
007                mov  al, 1              ; subservice 01 – update cursor
008                mov  bh, 0              ; output on page 0
009                mov  bl, 7              ; normal attrib
010                mov  dx, 0x0A03         ; row 10 column 3
011                mov  cx, 11             ; length of string
012                push cs
013                pop  es                 ; segment of string
014                mov  bp, message        ; offset of string
015                int  0x10               ; call BIOS video service
016
017                mov  ax, 0x4c00         ; terminate program
018                int  0x21
```

007	The subservice are versions of printstring that update and do not update the cursor after printing the string etc.
008	Text video screen is in the form of pages which can be upto 32. At one time one page is visible which is by default the zeroth page unless we change it.

When we execute it the string is printed and the cursor is updated as well. With direct access to video memory we had no control over the cursor. To control cursor a different mechanism to access the hardware was needed.

Our next example uses the keyboard service to read a key. The combination of keyboard and video services is used in almost every program that we see and use. We will wait for four key presses; clear the screen after the first, and draw different strings after the next key presses and exiting after the last. We will use INT 16 service 1 for this purpose. This is a blocking service so it does not return until a key has been pressed. We also used the blinking attribute in this example.

Example 8.3

```
001      ; print string and keyboard wait using BIOS services
002      [org 0x100]
003                    jmp  start
004
005      msg1:         db   'hello world', 0
006      msg2:         db   'hello world again', 0
007      msg3:         db   'hello world again and again', 0
008
009-028  ;;;;; COPY LINES 005-024 FROM EXAMPLE 7.1 (clrscr) ;;;;;
029-069  ;;;;; COPY LINES 050-090 FROM EXAMPLE 7.4 (printstr) ;;;;;
070-092  ;;;;; COPY LINES 028-050 FROM EXAMPLE 7.4 (strlen) ;;;;;
093
094      start:        mov  ah, 0x10           ; service 10 – vga attributes
095                    mov  al, 03             ; subservice 3 – toggle blinking
096                    mov  bl, 01             ; enable blinking bit
097                    int  0x10               ; call BIOS video service
098
099                    mov  ah, 0              ; service 0 – get keystroke
100                    int  0x16               ; call BIOS keyboard service
101
102                    call clrscr             ; clear the screen
103
104                    mov  ah, 0              ; service 0 – get keystroke
105                    int  0x16               ; call BIOS keyboard service
106
107                    mov  ax, 0
108                    push ax                 ; push x position
109                    mov  ax, 0
110                    push ax                 ; push y position
111                    mov  ax, 1              ; blue on black
```

```
112                push ax                 ; push attribute
113                mov  ax, msg1
114                push ax                 ; push offset of string
115                call printstr           ; print the string
116
117                mov  ah, 0              ; service 0 – get keystroke
118                int  0x16               ; call BIOS keyboard service
119
120                mov  ax, 0
121                push ax                 ; push x position
123                mov  ax, 0
124                push ax                 ; push y position
125                mov  ax, 0x71           ; blue on white
126                push ax                 ; push attribute
127                mov  ax, msg1
128                push ax                 ; push offset of string
129                call printstr           ; print the string
130
131                mov  ah, 0              ; service 0 – get keystroke
132                int  0x16               ; call BIOS keyboard service
133
134                mov  ax, 0
135                push ax                 ; push x position
136                mov  ax, 0
137                push ax                 ; push y position
138                mov  ax, 0xF4           ; red on white blinking
139                push ax                 ; push attribute
140                mov  ax, msg1
141                push ax                 ; push offset of string
142                call printstr           ; print the string
143
144                mov  ah, 0              ; service 0 – get keystroke
145                int  0x16               ; call BIOS keyboard service
146
147                mov  ax, 0x4c00         ; terminate program
148                int  0x21
```

099-100 This service has no parameters so only the service number is initialized in AH. This is the only service so there is no subservice number as well. The ASCII code of the char pressed is returned in AL after this service.

EXERCISES

1. Write a TSR that forces a program to exit when it tries to become a TSR using INT 21h/Service 31h by converting its call into INT 21h/Service 4Ch.
2. Write a function to clear the screen whose only parameter is always zero. The function is hooked at interrupt 80h and may also be called directly both as a near call and as a far call. The function should detect how it is called and return appropriately. It is provided that the direction flag will be set before the function is called.
3. Write a function that takes three parameters, the interrupt number (N) and the segment and offset of an interrupt handler XISR. The arguments are pushed in the order N, XISR's offset and XISR's segment. It is known that the first two instructions of XISR are PUSHF and CALL 0:0 followed by the rest of the interrupt handler. PUSHF instruction is of one byte and far call is of 5 bytes with the first byte being the op-code, the next two containing the target offset and the last two containing the target segment. The function should hook XISR at interrupt N and chain it to the interrupt handler previously hooked at N by manipulating the call 0:0 instruction placed near the start of XISR.
4. Write a TSR that provide the circular queue services via interrupt 0x80 using the code written in Exercise 5.XX. The interrupt procedure should call one of qcreate, qdestroy, qempty, qadd,

qremove, and uninstall based on the value in AH. The uninstall function should restore the old interrupt 0x80 handler and remove the TSR from memory.

2
Real Time Interrupts and Hardware Interfacing

2.1. HARDWARE INTERRUPTS

The same mechanism as discussed in the previous chapter is used for real interrupts that are generated by external hardware. However there is a single pin outside the processor called the INT pin that is used by external hardware to generate interrupts. The detailed operation that happens outside the process when an interrupt is generated is complex and only a simplified view will be discussed here; the view that is relevant to an assembly language programmer. There are many external devices that need the processor's attention like the keyboard, hard disk, floppy disk, sound card. All of them need real time interrupts at some point in their operation. For example if a program is busy in some calculations for three minutes the key strokes that are hit meanwhile should not be wasted. Therefore when a key is pressed, the INT signal is sent, an interrupt generated and the interrupt handler stores the key for later use. Similarly when the printer is busy printing we cannot send it more data. As soon as it gets free from the previous job it interrupts the processor to inform that it is free now. There are many other examples where the processor needs to be informed of an external event. If the processor actively monitors all devices instead of being automatically interrupted then it there won't be any time to do meaningful work.

Since there are many devices generating interruptsand there is only one pin going inside the processor and one pin cannot be technically drived by more than one source a controller is used in between called the Programmable Interrupt Controller (PIC). It has eight input signals and one output signal. It assigns priorities to its eight input pins from 0 to 7 so that if more than one interrupt comes at the same times, the highest priority one is forwarded and the rest are held till that is serviced. The rest are forwarded one by one according to priority after the highest priority one is completed. The original IBM XT computer had one PIC so there were 8 possible interrupt sources. However IBM AT and later computers have two PIC totaling 16 possible interrupt sources. They are arrange is a special cascade master slave arrangement so that only one output signal comes towards the processor. However we will concentrate on the first interrupt controller only.

The priority can be understood with the following example. Consider eight parallel switches which are all closed and connected to form the output signal. When a signal comes on one of the switches, it is passed on to the output and this switch and all below it are opened so that no further signals can pass through it. The higher priority switches are still closed and the signal on them can be forwarded. When the proessor signals that it is finished with the processing the switches are closed again and any waiting interrupts may be forwarded. The way the processor signals ending of the interrupt service routine is by using a special mechanism discussed later.

The eight input signals to the PIC are called Interrupt Requests (IRQ). The eight lines are called IRQ 0 to IRQ 7. These are the input lines of the 8451.* For example IRQ 0 is drived by a timer device. The timer device keeps

* 8451 is the technical number of the PIC.

generating interrupts with a specified frequency. IRQ 1 is drived by the keyboard when generates an interrupts when a key is pressed or released. IRQ 2 is the cascading interrupt connected to the output of the second 8451 in the machine. IRQ 3 is connected to serial port COM 2 while IRQ 4 is connected to serial port COM 1. IRQ 5 is used by the sound card or the network card or the modem. An IRQ conflict means that two devices in the system want to use the same IRQ line. IRQ 6 is used by the floppy disk drive while IRQ 7 is used by the parallel port.

Each IRQ is mapped to a specific interrupt in the system. This is called the IRQ to INT mapping. IRQ 0 to IRQ 7 are consecutively mapped on interrupts 8 to F. This mapping is done by the PIC and not the processor. The actual mechanism fetches one instruction from the PIC whenever the INT pin is signaled instead of the memory. We can program the PIC to generate a different set of interrupts on the same interrupt requests. From the perspective of an assembly language programmer an IRQ 0 is translated into an INT 8 without any such instruction in the program and that's all. Therefore an IRQ 0, the highest priority interrupt, is generated by the timer chip at a precise frequency and the handler at INT 8 is invoked which updates the system time. A key press generates IRQ 1 and the INT 9 handler is invoked which stores this key. To handler the timer and keyboard interrupts one can replace the vectors corresponding to interrupt 8 and 9 respectively. For example if the timer interrupt is replaced and the floppy is accessed by some program, the floppy motor and its light will remain on for ever as in the normal case it is turned off by the timer interrupt after two seconds in anticipation that another floppy access might be needed otherwise the time motor takes to speed up will be needed again.[†]

We have seen that an interrupt request from a device enters the PIC as an IRQ, from there it reaches the INT pin of the processor, the proessor receives the interrupt number from the PIC, generates the designated interrupt, and finally the interrupt handler gain control and can do whatever is desired. At the end of servicing the interrupt the handler should inform the PIC that it is completed so that lower priority interrupts can be sent from the PIC. This signal is called an End Of Interrupt (EOI) signal and is sent through the I/O ports of the interrupt controller.

2.2. I/O PORTS

There are hundreds of peripheral devices in the system, PIC is one example. The processor needs to communicate with them, give and take data from them, otherwise their presence is meaningless. Memory has a totally different purpose. It contains the program to be executed and its data. It does not control any hardware. For communicating with peripheral devices the processor uses I/O ports. There are only two operations with the external world possible, read or write. Similarly with I/O ports the processor can read or write an I/O port. When an I/O port is read or written to, the operation is not as simple as it happens in memory. Some hardware changes it functionality or performs some operation as a result.

IBM PC has separate memory address space and peripheral address space. Some processors use memory mapped I/O in which case designated memory cells work as ports for specific devices. In case of Intel a special pin on the control bus signals whether the current read or write is from the memory address space or from the peripheral address space. The same address and data buses are used to select a port and to read or write data from that port. However with I/O only the lower 16 bits of the address bus are used meaning that there are a total of 65536 possible I/O ports. Now keyboard has special

[†] The programs discussed from now onwards in the book must be executed in pure DOS and not in a DOS window so that we are in total control of the PIC and other devices.

I/O ports designated to it, PIC has others, DMA, sound card, network card, each has some ports.

If the two address spaces are differentiated in hardware, they must also have special instructions to select the other address space. We have the IN and OUT instructions to read or write from the peripheral address space. When MOV is given the processor selects the memory address space, when IN is given the processor selects the peripheral address space.

IN and OUT instructions

The IN and OUT instructions have a byte form and a word form but the byte form is almost always used. The source register in OUT and destination register in IN is AL or AX depending on which form is used. The port number can be directly given in the instruction if it fits in a byte otherwise it has to be given in the DX register. Port numbers for specific devices are fixed by the IBM standard. For example 20 and 21 are for PIC, 60 to 64 for Keybaord, 378 for the parallel port etc. A few example of IN and OUT are below:

```
in al, 0x21

mov dx, 0x378
in al, dx

out 0x21, al

mov dx, 0x378
out dx, al
```

PIC Ports

Programmable interrupt controller has two ports 20 and 21. Port 20 is the control port while port 20 is the interrupt mask register which can be used for selectively enabling or disabling interrupts. Each of the bits at port 21 corresponds to one of the IRQ lines. We first write a small program to disable the keyboard using this port. As we know that the keyboard IRQ is 1, we place a 1 bit at its corresponding position. A 0 bit will enable an interrupt and a 1 bit disables it. As soon as we write it on the port keyboard interrupts will stop arriving and the keyboard will effectively be disabled. Even Ctrl-Alt-Del would not work; the reset power button has to be used.

Example 9.1

```
; disable keyboard interrupt in PIC mask register
[org 0x0100]
              in   al, 0x21           ; read interrupt mask register
              or   al, 2              ; set bit for IRQ2
              out  0x21, al           ; write back mask register

              mov  ax, 0x4c00         ; terminate program
              int  0x21
```

After this three line mini program is executed the computer will not understand anything else. Its ears are closed. No keystrokes are making their way to the processor. Ports always make something happen on the system. A properly designed system can launch a missile on writing a bit on some port. Memory is simple in that it is all that it is. In ports every bit has a meaning that changes something in the system.

As we previously discussed every interrupt handler invoked because of an IRQ must signal an EOI otherwise lower priority interrupts will remain disabled.

Keyboard Controller

We will go in further details of the keyboard and its relation to the computer. We will not discuss how the keyboard communicates with the keyboard controller in the computer rather we will discuss how the keyboard controller communicates with the processor. Keyboard is a collection of labeled buttons and every button is designated a number (not the ASCII code). This number is sent to the processor whenever the key is pressed. From this number called the scan code the processor understands which key was pressed. For each key the scan code comes twice, once for the key press and once for the key release. Both are scan codes and differ in one bit only. The lower seven bits contain the key number while the most significant bit is clear in the press code and set in the release code. The IBM PC standard gives a table of the scan codes of all keys.

If we press Shift-A resulting in a capital A on the screen, the controller has sent the press code of Shift, the press code of A, the release code of A, the release code of Shift and the interrupt handler has understood that this sequence should result in the ASCII code of ‘A’. The ‘A’ key always produces the same scan code whether or not shift is pressed. It is the interrupt handler’s job to remember that the press code of Shift has come and release code has not yet come and therefore to change the meaning of the following key presses. Even the capslock key works the same way.

An interesting thing is that the two shift keys on the left and right side of the keyboard produce different scan codes. The standard way implemented in BIOS is to treat that similarly. That’s why we always think of them as identical. If we leave BIOS and talk directly with the hardware we can differentiate between left and right shift keys with their scan code. Now this scan code is available from the keyboard data port which is 60. The keyboard generates IRQ 1 whenever a key is pressed so if we hook INT 9 and inside it read port 60 we can tell which of the shift keys was hit. Our first program will do precisely this. It will output an L if the left shift key was pressed and R if the right one was pressed. The hooking method is the same as done in the previous chapter.

Example 9.2

```
; differentiate left and right shift keys with scancodes
[org 0x0100]
              jmp  start

; keyboard interrupt service routine
kbisr:        push ax
              push es

              mov  ax, 0xb800
              mov  es, ax             ; point es to video memory

              in   al, 0x60           ; read a char from keyboard port
              cmp  al, 0x2a           ; is the key left shift
              jne  nextcmp            ; no, try next comparison

              mov  byte [es:0], 'L'   ; yes, print L at top left
              jmp  nomatch            ; leave interrupt routine

nextcmp:      cmp  al, 0x36           ; is the key right shift
              jne  nomatch            ; no, leave interrupt routine

              mov  byte [es:0], 'R'   ; yes, print R at top left

nomatch:      mov  al, 0x20
              out  0x20, al           ; send EOI to PIC

              pop  es
              pop  ax
              iret

start:        xor  ax, ax
              mov  es, ax             ; point es to IVT base
              cli                     ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs     ; store segment at n*4+2
              sti                     ; enable interrupts
```

```
037
038     l1:           jmp  l1                ; infinite loop
```

033-036	CLI clears the interrupt flag to disable the interrupt system completely. The processor closes its ears and does not care about the state of the INT pin. Interrupt hooking is done in two instructions, placing the segment and placing the offset. If an interrupt comes inbetween and the vector is ina an indeterminate state, the sytem will go to a junk address and eventually crash. So we stop all interruptions while changing a real time interrupt vector. We set the interrupt flag afterwards to reenable interrupts.
038	The program hangs in an infinite loop. The only activity can be caused by a real time interrupt. The kbisr routine is not called from anywhere; it is only automatically invoked as a result of IRQ 1.

When the program is executed the left and right shift keys can be distinguished with the L or R on the screen. As no action was taken for the rest of the keys, they are effectively disabled and the computer has to be rebooted. To check that the keyboard is actually disabled we change the program and add the INT 16 service 0 at the end to wait for an Esc keypress. As soon as Esc is pressed we want to terminate our program.

Example 9.3

```
; attempt to terminate program with Esc that hooks keyboard interrupt
[org 0x0100]
              jmp  start

;;;;; COPY LINES 005-029 FROM EXAMPLE 9.2 (kbisr) ;;;;;

start:        xor  ax, ax
              mov  es, ax            ; point es to IVT base
              cli                    ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs    ; store segment at n*4+2
              sti                    ; enable interrupts

l1:           mov  ah, 0             ; service 0 – get keystroke
              int  0x16              ; call BIOS keyboard service

              cmp al, 27             ; is the Esc key pressed
              jne l1                 ; if no, check for next key

              mov ax, 0x4c00         ; terminate program
              int 0x21
```

When the program is executed the behavior is same. Esc does not work. This is because the original IRQ 1 handler was written by BIOS that read the scan code, converted into an ASCII code and stored in the keyboard buffer. The BIOS INT 16 read the key from there and gives in AL. When we hooked the keyboard interrupt BIOS is no longer in control, it has no information, it will always see the empty buffer and INT 16 will never return.

Interrupt Chaining

We can transfer control to the original BIOS ISR in the end of our routine. This way the normal functioning of INT 16 can work as well. We can retrieve the address of the BIOS routine by saving the values in vector 9 before hooking our routine. In the end of our routine we will jump to this address using a special indirect form of the JMP FAR instruction.

Example 9.4

```
001 ; another attempt to terminate program with Esc that hooks
002 ; keyboard interrupt
003 [org 0x100]
004               jmp  start
005
006 oldisr:       dd   0                  ; space for saving old isr
007
008 ; keyboard interrupt service routine
009 kbisr:        push ax
010               push es
011
012               mov  ax, 0xb800
013               mov  es, ax             ; point es to video memory
014
015               in   al, 0x60           ; read a char from keyboard port
016               cmp  al, 0x2a           ; is the key left shift
017               jne  nextcmp            ; no, try next comparison
018
019               mov  byte [es:0], 'L'   ; yes, print L at top left
020               jmp  nomatch            ; leave interrupt routine
021
022 nextcmp:      cmp  al, 0x36           ; is the key right shift
023               jne  nomatch            ; no, leave interrupt routine
024
025               mov  byte [es:0], 'R'   ; yes, print R at top left
026
027 nomatch:      ; mov  al, 0x20
028               ; out  0x20, al
029
030               pop es
031               pop ax
032               jmp far [cs:oldisr]     ; call the original ISR
033               ; iret
034
035 start:        xor  ax, ax
036               mov  es, ax             ; point es to IVT base
037               mov ax, [es:9*4]
038               mov [oldisr], ax        ; save offset of old routine
039               mov ax, [es:9*4+2]
040               mov [oldisr+2], ax      ; save segment of old routine
041               cli                     ; disable interrupts
042               mov  word [es:9*4], kbisr ; store offset at n*4
043               mov  [es:9*4+2], cs     ; store segment at n*4+2
044               sti                     ; enable interrupts
045
046 l1:           mov  ah, 0              ; service 0 – get keystroke
047               int  0x16               ; call BIOS keyboard service
048
049               cmp al, 27              ; is the Esc key pressed
050               jne l1                  ; if no, check for next key
051
052               mov ax, 0x4c00          ; terminate program
053               int 0x21
```

027-028	EOI is no longer needed as the original BIOS routine will have it at its end.
033	IRET has been removed and an unconditional jump is introduced. At time of JMP the stack has the exact formation as was when the interrupt came. So the original BIOS routine's IRET will take control to the interrupted program. We have been careful in restoring every register we modified and retained the stack in the same form as it was at the time of entry into the routine.

When the program is executed L and R are printed as desired and Esc terminates the program as well. Normal commands like DIR work now and shift keys still show L and R as our routine did even after the termination of our program. Now start some application like the editor, it open well but as soon as a key is pressed the computer crashes.

Actually our hookin and chaining was fine. When Esc was pressed we signaled DOS that our program has terminated. DOS will take all our memory as a result. The routine is still in memory and functioning but the memory is free according to DOS. As soon as we load EDIT the same memory is allocated to EDIT and our routine as overwritten. Now when a key is pressed our routine's address is in the vector but at that address some new code is placed that is not intended to be an interrupt handler. That may be data or some part of the EDIT program. This results in crashing the computer.

Unhooking Interrupt

We now add the interrupt restoring part to our program. This code resets the interrupt vector to the value it had before the start of our program.

Example 9.5

```
; terminate program with Esc that hooks keyboard interrupt
[org 0x100]
              jmp  start

oldisr:       dd   0                  ; space for saving old isr

;;;;; COPY LINES 005-029 FROM EXAMPLE 9.4 (kbisr) ;;;;;

start:        xor  ax, ax
              mov  es, ax             ; point es to IVT base
              mov ax, [es:9*4]
              mov [oldisr], ax        ; save offset of old routine
              mov ax, [es:9*4+2]
              mov [oldisr+2], ax      ; save segment of old routine
              cli                     ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs     ; store segment at n*4+2
              sti                     ; enable interrupts

l1:           mov  ah, 0              ; service 0 – get keystroke
              int  0x16               ; call BIOS keyboard service

              cmp al, 27              ; is the Esc key pressed
              jne l1                  ; if no, check for next key

              mov ax, [oldisr]        ; read old offset in ax
              mov bx, [oldisr+2]      ; read old segment in bx
              cli                     ; disable interrupts
              mov [es:9*4], ax        ; restore old offset from ax
              mov [es:9*4+2], bx      ; restore old segment from bx
              sti                     ; enable interrupts

              mov ax, 0x4c00          ; terminate program
              int 0x21
```

2.3. TERMINATE AND STAY RESIDENT

We change the display to show L only while the left shift is pressed and R only while the right shift is pressed to show the use of the release codes. We also changed that shift keys are not forwarded to BIOS. The effect will be visible with A and Shift-A both producing small 'a' but capslock will work.

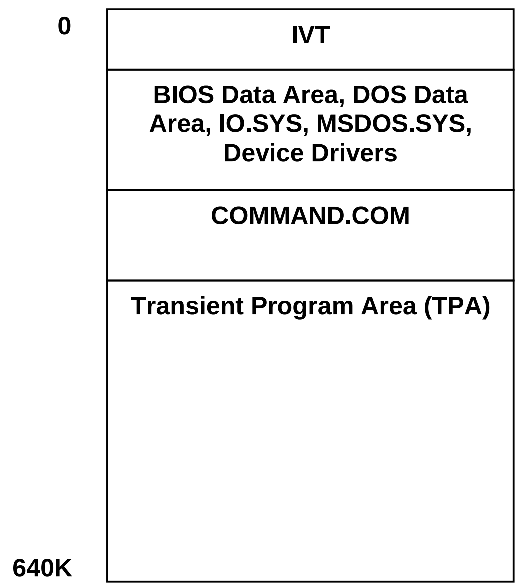
There is one major difference from all the programs we have been writing till now. The termination is done using INT 21 service 31 instead of INT 21 service 4C. The effect is that even after termination the program is there and is legally there.

Example 9.6

```
; TSR to show status of shift keys on top left of screen
[org 0x0100]
              jmp  start

oldisr:       dd   0                  ; space for saving old isr

```

```
; keyboard interrupt service routine
kbisr:        push ax
              push es

              mov  ax, 0xb800
              mov  es, ax             ; point es to video memory

              in   al, 0x60           ; read a char from keyboard port
              cmp  al, 0x2a           ; has the left shift pressed
              jne  nextcmp            ; no, try next comparison

              mov  byte [es:0], 'L'   ; yes, print L at first column
              jmp  exit               ; leave interrupt routine

nextcmp:      cmp  al, 0x36           ; has the right shift pressed
              jne  nextcmp2           ; no, try next comparison

              mov  byte [es:0], 'R'   ; yes, print R at second column
              jmp  exit               ; leave interrupt routine

nextcmp2:     cmp  al, 0xaa           ; has the left shift released
              jne  nextcmp3           ; no, try next comparison

              mov  byte [es:0], ' '   ; yes, clear the first column
              jmp  exit               ; leave interrupt routine

nextcmp3:     cmp  al, 0xb6           ; has the right shift released
              jne  nomatch            ; no, chain to old ISR

              mov  byte [es:2], ' '   ; yes, clear the second column
              jmp  exit               ; leave interrupt routine

nomatch:      pop  es
              pop  ax
              jmp  far [cs:oldisr]    ; call the original ISR

exit:         mov  al, 0x20
              out  0x20, al           ; send EOI to PIC

              pop  es
              pop  ax
              iret                    ; return from interrupt

start:        xor  ax, ax
              mov  es, ax             ; point es to IVT base
              mov  ax, [es:9*4]
              mov  [oldisr], ax       ; save offset of old routine
              mov  ax, [es:9*4+2]
              mov  [oldisr+2], ax     ; save segment of old routine
              cli                     ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs     ; store segment at n*4+2
              sti                     ; enable interrupts

              mov  dx, start          ; end of resident portion
              add  dx, 15             ; round up to next para
              mov  cl, 4
              shr  dx, cl             ; number of paras
              mov  ax, 0x3100         ; terminate and stay resident
              int  0x21
```

When this program is executed the command prompt immediately comes. DIR can be seen. EDIT can run and keypresses do not result in a crash. And with all that left and right shift keys shown L and R on top left of the screen while they are pressed but the shift keys do not work as usual since we did not forwarded the key to BIOS. This is selective chaining.



To understand Terminate and Stay Resident (TSR) programs the DOS memory formation and allocation procedure must be understood. At physical address zero is the interrupt vector table. Above it are the BIOS data area, DOS data area, IO.SYS, MSDOS.SYS and other device drivers. In the end there is COMMAND.COM command interpreter. The remaining space is called the transient program area as programs are loaded and executed in this area and the space reclaimed on their exit. A freemem pointer in DOS points where the free memory begins. When DOS loads a program the freemem pointer is moved to the end of memory, all the available space is allocated to it, and when it exits the freemem pointer comes back to its original place thereby reclaiming all space. This action is initiated by the DOS service 4C.

The second method to legally terminate a program and give control back to DOS is using the service 31. Control is still taken back but the memory releasing part is modified. A portion of the allocated memory can be retained. So the differene in the two methods is that the freemem pointer goes back to the original place or a designated number of bytes ahead of that old position. Remember that our program crashed because the interrupt routine was overwritten. If we can tell DOS not to reclaim the memory of the interrupt routine, then it will not crash. In the last program we have told DOS to make a number of bytes resident. It becomes a part of the operation system, an extension to it. Just like DOSKEY[‡] is an extension to the operation system.

The number of paragraps to reserve is given in the DX register. Paragraph is a unit just like byte, word, and double word. A paragraph is 16 bytes. Therefore we can reserve in multiple of 16 bytes. We write TSRs in such a way that the initialization code and data is located at the end as it is not necessary to make it resident and therefore to save space.

To calculate the number of paragraphs a label is placed after the last line that is to be made resident. The value of that label is the number of bytes needed to be made resident. A simple division by 16 will not give the correct number of paras as we want our answer to be rounded up and not down. For example 100 bytes should need 7 pages but division gives 6 and a remainder of 4. A standard technique to get rounded up integer division is to add divisor-1 to the dividend and then divide. So we add 15 to the number of bytes and then divide by 16. We use shifting for division as the divisor is a power of 2. We use a form of SHR that places the count in the CL register so that we can shift by 4 in just two instructions instead of 4 if we shift one by one.

In our program anything after start label is not needed after the program has become a TSR. We can observe that our program has become a part of DOS by giving the following command.

```
mem /c
```

‡ DOSKEY is a TSR that shows the previous commands on the command prompt with up and down arrows and allows editing of the command

This command displays all currently loaded drivers and the current state of memory. We will be able to see our program in the list of DOS drivers.

2.4. PROGRAMMABLE INTERVAL TIMER

Another very important peripheral device is the Programmable Interval Timer (PIT), the chip numbered 8254. This chip has a precise input frequency of 1.19318 MHz. This frequency is fixed regardless of the processor clock. Inside the chip is a 16bit divisor which divides this input frequency and the output is connected to the IRQ 0 line of the PIC. The special number 0 if placed in the divisor means a divisor of 65536 and not 0. The standard divisor is 0 unless we change it. Therefore by default IRQ 0 is generated 1193180/65536=18.2 times per second. This is called the timer tick. There is an interval of about 55ms between two timer ticks. The system time is maintained with the timer interrupt. This is the highest priority interrupt and breaks whatever is executing. Time can be maintained with this interrupt as this frequency is very precise and is part of the IBM standard.

When writing a TSR we give control back to DOS so TSR activation, reactivation and action is solely interrupt based, whether this is a hardware interrupt or a software one. Control is never given back; it must be caught, just like we caught control by hooking the keyboard interrupt. Our next example will hook the timer interrupt and display a tick count on the screen.

Example 9.7

```
; display a tick count on the top right of screen
[org 0x0100]
              jmp  start

tickcount:    dw   0

; subroutine to print a number at top left of screen
; takes the number to be printed as its parameter
printnum:     push bp
              mov  bp, sp
              push es
              push ax
              push bx
              push cx
              push dx
              push di

              mov  ax, 0xb800
              mov  es, ax            ; point es to video base
              mov  ax, [bp+4]        ; load number in ax
              mov  bx, 10            ; use base 10 for division
              mov  cx, 0             ; initialize count of digits

nextdigit:    mov  dx, 0             ; zero upper half of dividend
              div  bx                ; divide by 10
              add  dl, 0x30          ; convert digit into ascii value
              push dx                ; save ascii value on stack
              inc  cx                ; increment count of values
              cmp  ax, 0             ; is the quotient zero
              jnz  nextdigit         ; if no divide it again

              mov  di, 140           ; point di to 70th column

nextpos:      pop  dx                ; remove a digit from the stack
              mov  dh, 0x07          ; use normal attribute
              mov  [es:di], dx       ; print char on screen
              add  di, 2             ; move to next screen location
              loop nextpos           ; repeat for all digits on stack

              pop  di
              pop  dx
              pop  cx
              pop  bx
              pop  ax
```

```
                pop  es
                pop  bp
                ret  2

; timer interrupt service routine
timer:          push ax

                inc  word [cs:tickcount]; increment tick count
                push word [cs:tickcount]
                call printnum           ; print tick count

                mov  al, 0x20
                out  0x20, al           ; end of interrupt

                pop  ax
                iret                    ; return from interrupt

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                cli                     ; disable interrupts
                mov  word [es:8*4], timer; store offset at n*4
                mov  [es:8*4+2], cs     ; store segment at n*4+2
                sti                     ; enable interrupts

                mov  dx, start          ; end of resident portion
                add  dx, 15             ; round up to next para
                mov  cl, 4
                shr  dx, cl             ; number of paras
                mov  ax, 0x3100         ; terminate and stay resident
                int  0x21
```

When we execute the program the counter starts on the screen. Whatever we do, take directory, open EDIT, the debugger etc. the counter remains running on the screen. No one is giving control to the program; the program is getting executed as a result of timer generating INT 8 after every 55ms.

Our next example will hook both the keyboard and timer interrupts. When the shift key is pressed the tick count starts incrementing and as soon as the shift key is released the tick count stops. Both interrupt handlers are communicating through a common variable. The keyboard interrupt sets this variable while the timer interrupts modifies its behavior according to this variable.

Example 9.8

```
; display a tick count while the left shift key is down
[org 0x0100]
                jmp  start

seconds:        dw   0
timerflag:      dw   0
oldkb:          dd   0

;;;;; COPY LINES 007-047 FROM EXAMPLE 9.7 (printnum) ;;;;;

; keyboard interrupt service routine
kbisr:          push ax

                in   al, 0x60           ; read char from keyboard port
                cmp  al, 0x2a           ; has the left shift pressed
                jne  nextcmp            ; no, try next comparison

                cmp  word [cs:timerflag], 1; is the flag already set
                je   exit               ; yes, leave the ISR

                mov  word [cs:timerflag], 1; set flag to start printing
                jmp  exit               ; leave the ISR

nextcmp:        cmp  al, 0xaa           ; has the left shift released
                jne  nomatch            ; no, chain to old ISR

                mov  word [cs:timerflag], 0; reset flag to stop printing
```

```
068                 jmp  exit               ; leave the interrupt routine
069
070  nomatch:       pop  ax
071                 jmp  far [cs:oldkb]     ; call original ISR
072
073  exit:          mov  al, 0x20
074                 out  0x20, al           ; send EOI to PIC
075
076                 pop  ax
077                 iret                    ; return from interrupt
078
079  ; timer interrupt service routine
080  timer:         push ax
081
082                 cmp  word [cs:timerflag], 1 ; is the printing flag set
083                 jne  skipall            ; no, leave the ISR
084
085                 inc  word [cs:seconds]  ; increment tick count
086                 push word [cs:seconds]
087                 call printnum           ; print tick count
088
089  skipall:       mov  al, 0x20
090                 out  0x20, al           ; send EOI to PIC
091
092                 pop  ax
093                 iret                    ; return from interrupt
094
095  start:         xor  ax, ax
096                 mov  es, ax             ; point es to IVT base
097                 mov  ax, [es:9*4]
098                 mov  [oldkb], ax        ; save offset of old routine
099                 mov  ax, [es:9*4+2]
100                 mov  [oldkb+2], ax      ; save segment of old routine
101                 cli                     ; disable interrupts
102                 mov  word [es:9*4], kbisr ; store offset at n*4
103                 mov  [es:9*4+2], cs     ; store segment at n*4+2
104                 mov  word [es:8*4], timer ; store offset at n*4
105                 mov  [es:8*4+2], cs     ; store segment at n*4+
106                 sti                     ; enable interrupts
107
108                 mov  dx, start          ; end of resident portion
109                 add  dx, 15             ; round up to next para
110                 mov  cl, 4
111                 shr  dx, cl             ; number of paras
112                 mov  ax, 0x3100         ; terminate and stay resident
113                 int  0x21
```

006	This flag is one when the timer interrupt should increment and zero when it should not.
058-059	As the keyboard controller repeatedly generates the press code if the release code does not come in a specified time, we have placed a check to not repeatedly set it to one.
058	Another way to access TSR data is using the CS override instead of initializing DS. It is common mistake not to initialize DS and also not put in CS override in a real time interrupt handler.

When we execute the program and the shift key is presseed, the counter starts incrementing. When the key is released the counter stops. When it is pressed again the counter resumes counting. As this is made as a TSR any other program can be loaded and will work properly alongside the TSR.

2.5. PARALLEL PORT

Computers can control external hardware through various external ports like the parallel port, the serial port, and the new additions USB and FireWire. Using this, computers can be used to control almost anything. For our examples we will use the parallel port. The parallel port has two views, the connector that the external world sees and the parallel port controller ports through which the processor communicates with the device connected to the parallel port.

The parallel port connector is a 25pin connector called DB-25. Different pins of this connector have different meanings. Some are meaningful only with the printer[§]. This is a bidirectional port so there are some pins to take data from the processor to the parallel port and others to take data from the parallel port to the processor. Important pins for our use are the data pins from pin 2 to pin 9 that take data from the processor to the device. Pin 10, the ACK pin, is normally used by the printer to acknowledge the receipt of data and show the willingness to receive more data. Signalling this pin generates IRQ 7 if enabled in the PIC and in the parallel port controller. Pin 18-25 are ground and must be connected to the external circuit ground to provide the common reference point otherwise they won't understand each other voltage levels. Like the datum point in a graph this is the datum point of an electrical circuit. The remaining pins are not of our concern in these examples.

This is the external view of the parallel port. The processor cannot see these pins. The processor uses the I/O ports of the parallel port controller. The first parallel port LPT1[**] has ports designated from 378 to 37A. The first port 378 is the data port. If we use the OUT instruction on this port, 1 bits result in a 5V signal on the corresponding pin and a 0 bits result in a 0V signal on the corresponding pin.

Port 37A is the control port. Our interest is with bit 4 of this port which enables the IRQ 7 triggering by the ACK pin. We have attached a circuit that connects 8 LEDs with the parallel port pins. The following examples sends the scancode of the key pressed to the parallel port so that it is visible on LEDs.

Example 9.9

```
; show scancode on external LEDs connected through parallel port
[org 0x0100]
              jmp  start

oldisr:       dd   0                  ; space for saving old ISR

; keyboard interrupt service routine
kbisr:        push ax
              push dx

              in   al, 0x60           ; read char from keyboard port
              mov  dx, 0x378
              out  dx, al             ; write char to parallel port

              pop  ax
              pop  dx
              jmp  far [cs:oldisr]    ; call original ISR

start:        xor  ax, ax
              mov  es, ax             ; point es to IVT base
              mov  ax, [es:9*4]
              mov  [oldisr], ax       ; save offset of old routine
              mov  ax, [es:9*4+2]
              mov  [oldisr+2], ax     ; save segment of old routine
              cli                     ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs     ; store segment at n*4+2
              sti                     ; enable interrupts

              mov  dx, start          ; end of resident portion
              add  dx, 15             ; round up to next para
```

[§] The parallel port is most commonly used with the printer. However some new printers have started using the USB port.

[**] Older computer had more than one parallel port named LPT2 and having ports from 278-27A.

```
                mov  cl, 4
                shr  dx, cl             ; number of paras
                mov  ax, 0x3100         ; terminate and stay resident
                int  0x21
```

The following example uses the same LED circuit and sends data such that LEDs switch on and off turn by turn so that it looks like light is moving back and forth.

Example 9.10

```
; show lights moving back and forth on external LEDs
[org 0x0100]
                jmp  start

signal:         db   1                  ; current state of lights
direction:      db   0                  ; current direction of motion

; timer interrupt service routine
timer:          push ax
                push dx
                push ds

                push cs
                pop  ds                 ; initialize ds to data segment

                cmp  byte [direction], 1; are moving in right direction
                je   moveright          ; yes, go to shift right code

                shl  byte [signal], 1   ; shift left state of lights
                jnc  output             ; no jump to change direction

                mov  byte [direction], 1; change direction to right
                mov  byte [signal], 0x80; turn on left most light
                jmp  output             ; proceed to send signal

moveright:      shr  byte [signal], 1   ; shift right state of lights
                jnc  output             ; no jump to change direction

                mov  byte [direction], 0; change direction to left
                mov  byte [signal], 1   ; turn on right most light

output:         mov  al, [signal]       ; load lights state in al
                mov  dx, 0x378          ; parallel port data port
                out  dx, al             ; send light state of port

                mov  al, 0x20
                out  0x20, al           ; send EOI on PIC

                pop  ds
                pop  dx
                pop  ax
                iret                    ; return from interrupt

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                cli                     ; disable interrupts
                mov  word [es:8*4], timer ; store offset at n*4
                mov  [es:8*4+2], cs     ; store segment at n*4+2
                sti                     ; enable interrupts

                mov  dx, start          ; end of resident portion
                add  dx, 15             ; round up to next para
                mov  cl, 4
                shr  dx, cl             ; number of paras
                mov  ax, 0x3100         ; terminate and stay resident
                int  0x21
```

We will now use the parallel port to control a slightly complicated circuit. This time we will also use the parallel port interrupt. We are using a 220 V bulb with AC input. AC current is 50Hz sine wave. We have made our circuit such that it triggers the parallel port interrupt whenever the since wave crosses zero. We have control of passing the AC current to the bulb. We control it such that in every cycle only a fixed percentage of time the current passes on to the bulb. Using this we can control the intensity or glow of the bulb.

Our first example will slowly turn on the bulb by increasing the power provided using the mechanism just described.

Example 9.11

```
; slowly turn on a bulb by gradually increasing the power provided
[org 0x0100]
              jmp  start

flag:         db   0                  ; next time turn on or turn off
stop:         db   0                  ; flag to terminate the program
divider:      dw   11000              ; divider for minimum intensity
oldtimer:     dd   0                  ; space for saving old isr

; timer interrupt service routine
timer:        push ax
              push dx

              cmp  byte [cs:flag], 0  ; are we here to turn off
              je   switchoff          ; yes, go to turn off code

switchon:     mov  al, 1
              mov  dx, 0x378
              out  dx, al             ; no, turn the bulb on

              mov  ax, 0x0100
              out  0x40, al           ; set timer divisor LSB to 0
              mov  al, ah
              out  0x40, al           ; set timer divisor MSB to 1
              mov  byte [cs:flag], 0  ; flag next timer to switch off
              jmp  exit               ; leave the interrupt routine

switchoff:    xor  ax, ax
              mov  dx, 0x378
              out  dx, al             ; turn the bulb off

exit:         mov  al, 0x20
              out  0x20, al           ; send EOI to PIC

              pop  dx
              pop  ax
              iret                    ; return from interrupt

; parallel port interrupt service routine
parallel:     push ax

              mov  al, 0x30           ; set timer to one shot mode
              out  0x43, al

              cmp  word [cs:divider], 100; is the current divisor 100
              je   stopit             ; yes, stop

              sub  word [cs:divider], 10; decrease the divisor by 10
              mov  ax, [cs:divider]
              out  0x40, al           ; load divisor LSB in timer
              mov  al, ah
              out  0x40, al           ; load divisor MSB in timer
              mov  byte [cs:flag], 1  ; flag next timer to switch on

              mov  al, 0x20
              out  0x20, al           ; send EOI to PIC
              pop  ax
              iret                    ; return from interrupt

stopit:       mov  byte [stop], 1     ; flag to terminate the program
              mov  al, 0x20
              out  0x20, al           ; send EOI to PIC
              pop  ax
              iret                    ; return from interrupt

```

```
start:          xor  ax, ax
                mov  es, ax              ; point es to IVT base
                mov  ax, [es:0x08*4]
                mov  [oldtimer], ax      ; save offset of old routine
                mov  ax, [es:0x08*4+2]
                mov  [oldtimer+2], ax    ; save segment of old routine
                cli                      ; disable interrupts
                mov  word [es:0x08*4], timer ; store offset at n*4
                mov  [es:0x08*4+2], cs   ; store segment at n*4+2
                mov  word [es:0x0F*4], parallel ; store offset at n*4
                mov  [es:0x0F*4+2], cs   ; store segment at n*4+2
                sti                      ; enable interrupts

                mov dx, 0x37A
                in al, dx                ; parallel port control register
                or al, 0x10              ; turn interrupt enable bit on
                out dx, al               ; write back register

                in al, 0x21              ; read interrupt mask register
                and al, 0x7F             ; enable IRQ7 for parallel port
                out 0x21, al             ; write back register

recheck:        cmp byte [stop], 1       ; is the termination flag set
                jne recheck              ; no, check again

                mov dx, 0x37A
                in al, dx                ; parallel port control register
                and al, 0xEF             ; turn interrupt enable bit off
                out dx, al               ; write back register

                in al, 0x21              ; read interrupt mask register
                or al, 0x80              ; disable IRQ7 for parallel port
                out 0x21, al             ; write back regsiter

                cli                      ; disable interrupts
                mov ax, [oldtimer]       ; read old timer ISR offset
                mov [es:0x08*4], ax      ; restore old timer ISR offset
                mov ax, [oldtimer+2]     ; read old timer ISR segment
                mov [es:0x08*4+2], ax    ; restore old timer ISR segment
                sti                      ; enable interrupts

                mov ax, 0x4c00           ; terminate program
                int 0x21
```

The next example is simply the opposite of the previous. It slowly turns the bulb off from maximum glow to no glow.

Example 9.12

```
; slowly turn off a bulb by gradually decreasing the power provided
[org 0x0100]
                jmp  start

flag:           db   0                   ; next time turn on or turn off
stop:           db   0                   ; flag to terminate the program
divider:        dw   0                   ; divider for maximum intensity
oldtimer:       dd   0                   ; space for saving old isr

;;;;; COPY LINES 009-036 FROM EXAMPLE 9.11 (timer) ;;;;;

; parallel port interrupt service routine
parallel:       push ax

                mov  al, 0x30            ; set timer to one shot mode
                out  0x43, al

                cmp  word [cs:divider], 11000; current divisor is 11000
                je   stopit              ; yes, stop

                add  word [cs:divider], 10; increase the divisor by 10
                mov  ax, [cs:divider]
                out  0x40, al            ; load divisor LSB in timer
                mov  al, ah
                out  0x40, al            ; load divisor MSB in timer
```

```
                mov  byte [cs:flag], 1  ; flag next timer to switch on

                mov  al, 0x20
                out  0x20, al           ; send EOI to PIC
                pop  ax
                iret                    ; return from interrupt

stopit:         mov  byte [stop], 1     ; flag to terminate the program
                mov  al, 0x20
                out  0x20, al           ; send EOI to PIC
                pop  ax
                iret                    ; return from interrupt

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                mov  ax, [es:0x08*4]
                mov  [oldtimer], ax     ; save offset of old routine
                mov  ax, [es:0x08*4+2]
                mov  [oldtimer+2], ax   ; save segment of old routine
                cli                     ; disable interrupts
                mov  word [es:0x08*4], timer ; store offset at n*4
                mov  [es:0x08*4+2], cs  ; store segment at n*4+2
                mov  word [es:0x0F*4], parallel ; store offset at n*4
                mov  [es:0x0F*4+2], cs  ; store segment at n*4+2
                sti                     ; enable interrupts

                mov  dx, 0x37A
                in   al, dx             ; parallel port control register
                or   al, 0x10           ; turn interrupt enable bit on
                out  dx, al             ; write back register

                in   al, 0x21           ; read interrupt mask register
                and  al, 0x7F           ; enable IRQ7 for parallel port
                out  0x21, al           ; write back register

recheck:        cmp  byte [stop], 1     ; is the termination flag set
                jne  recheck            ; no, check again

                mov  dx, 0x37A
                in   al, dx             ; parallel port control register
                and  al, 0xEF           ; turn interrupt enable bit off
                out  dx, al             ; write back register

                in   al, 0x21           ; read interrupt mask register
                or   al, 0x80           ; disable IRQ7 for parallel port
                out  0x21, al           ; write back regsiter

                cli                     ; disable interrupts
                mov  ax, [oldtimer]     ; read old timer ISR offset
                mov  [es:0x08*4], ax    ; restore old timer ISR offset
                mov  ax, [oldtimer+2]   ; read old timer ISR segment
                mov  [es:0x08*4+2], ax  ; restore old timer ISR segment
                sti                     ; enable interrupts

                mov  ax, 0x4c00         ; terminate program
                int  0x21
```

This example is a mix of the previous two. Here we can increase the bulb intensity with F11 and decrease it with F12.

Example 9.13

```
; control external bulb intensity with F11 and F12
[org 0x0100]
                jmp  start

flag:           db   0                  ; next time turn on or turn off
divider:        dw   100                ; initial timer divider
oldkb:          dd   0                  ; space for saving old ISR

;;;;; COPY LINES 009-036 FROM EXAMPLE 9.11 (timer) ;;;;;

; keyboard interrupt service routine
kbisr:          push ax
```

```
                in   al, 0x60
                cmp  al, 0x57
                jne  nextcmp
                cmp  word [cs:divider], 11000
                je   exitkb
                add  word [cs:divider], 100
                jmp  exitkb

nextcmp:        cmp  al, 0x58
                jne  chain
                cmp  word [cs:divider], 100
                je   exitkb
                sub  word [cs:divider], 100
                jmp  exitkb

exitkb:         mov  al, 0x20
                out  0x20, al

                pop  ax
                iret

chain:          pop  ax
                jmp  far [cs:oldkb]

; parallel port interrupt service routine
parallel:       push ax

                mov  al, 0x30           ; set timer to one shot mode
                out  0x43, al

                mov  ax, [cs:divider]
                out  0x40, al           ; load divisor LSB in timer
                mov  al, ah
                out  0x40, al           ; load divisor MSB in timer
                mov  byte [cs:flag], 1  ; flag next timer to switch on

                mov  al, 0x20
                out  0x20, al           ; send EOI to PIC
                pop  ax
                iret                    ; return from interrupt

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                mov  ax, [es:0x09*4]
                mov  [oldkb], ax        ; save offset of old routine
                mov  ax, [es:0x09*4+2]
                mov  [oldkb+2], ax      ; save segment of old routine
                cli                     ; disable interrupts
                mov word [es:0x08*4], timer ; store offset at n*4
                mov [es:0x08*4+2], cs   ; store segment at n*4+2
                mov word [es:0x09*4], kbisr ; store offset at n*4
                mov [es:0x09*4+2], cs   ; store segment at n*4+2
                mov word [es:0x0F*4], parallel ; store offset at n*4
                mov [es:0x0F*4+2], cs   ; store segment at n*4+2
                sti                     ; enable interrupts

                mov  dx, 0x37A
                in   al, dx             ; parallel port control register
                or   al, 0x10           ; turn interrupt enable bit on
                out  dx, al             ; write back register

                in   al, 0x21           ; read interrupt mask register
                and  al, 0x7F           ; enable IRQ7 for parallel port
                out  0x21, al           ; write back register

                mov  dx, start          ; end of resident portion
                add  dx, 15             ; round up to next para
                mov  cl, 4
                shr  dx, cl             ; number of paras
                mov  ax, 0x3100         ; terminate and stay resident
                int  0x21
```

EXERCISES

1. Suggest a reason for the following. The statements are all true.
    a. We should disable interrupts while hooking interrupt 8h. I.e. while placing its segment and offset in the interrupt vector table.
    b. We need not do this for interrupt 80h.
    c. We need not do this when hooking interrupt 8h from inside the interrupt handler of interrupt 80h.
    d. We should disable interrupts while we are changing the stack (SS and SP).
    e. EOI is not sent from an interrupt handler which does interrupt chaining.
    f. If no EOI is sent from interrupt 9h and no chaining is done, interrupt 8h still comes if the interrupt flag is on.
    g. After getting the size in bytes by putting a label at the end of a COM TSR, 0fh is added before dividing by 10h.
    h. Interrupts are disabled but divide by zero interrupt still comes.
2. If no hardware interrupts are coming, what are all possible reasons?
3. Write a program to make an asterisks travel the border of the screen, from upper left to upper right to lower right to lower left and back to upper left indefinitely, making each movement after one second.
4. [Musical Arrow] Write a TSR to make an arrow travel the border of the screen from top left to top right to bottom right to bottom left and back to top left at the speed of 36.4 locations per second. The arrow should not destroy the data beneath it and should be restored as soon as the arrow moves forward.
   The arrow head should point in the direction of movement using the characters > V < and ^. The journey should be accompanied by a different tone from the pc speaker for each side of the screen. Do interrupt chaining so that running the TSR 10 times produces 10 arrows travelling at different locations.
   HINT: At the start you will need to reprogram channel 0 for 36.4 interrupts per second, double the normal. You will have to reprogram channel 2 at every direction change, though you can enable the speaker once at the very start.
5. In the above TSR hook the keyboard interrupt as well and check if 'q' is pressed. If not chain to the old interrupt, if yes restore everything and remove the TSR from memory. The effect should be that pressing 'q' removes one moving arrow. If you do interrupt chaining when pressing 'q' as well, it will remove all arrows at once.
6. Write a TSR to rotate the screen (scroll up and copy the old top most line to the bottom) while F10 is pressed. The screen will keep rotating while F10 is pressed at 18.2 rows per second. As soon as F10 is released the rotation should stop and the original screen restored. A secondary buffer of only 160 bytes (one line of screen) can be used.
7. Write a TSR that hooks software interrupt 0x80 and the timer interrupt. The software interrupt is called by other programs with the address of a far function in ES:DI and the number of timer ticks after which to call back that function in CX. The interrupt records this information and returns to the caller. The function will actually be called by the timer interrupt after the desired number of ticks. The maximum number of functions and their ticks can be fixed to 8.
8. Write a TSR to clear the screen when CTRL key is pressed and restore it when it is released.
9. Write a TSR to disable all writes to the hard disk when F10 is pressed and re-enable when pressed again like a toggle.

HINT: To write to the hard disk programs call the BIOS service INT 0x13 with AH=3.

10. Write a keyboard interrupt handler that disables the timer interrupt (no timer interrupt should come) while Q is pressed. It should be re-enabled as soon as Q is released.
11. Write a TSR to calculate the current typing speed of the user. Current typing speed is the number of characters typed by the user in the last five seconds. The speed should be represented by printing asterisks at the right border (80th column) of the screen starting from the upper right to the lower right corner (growing downwards). Draw n asterisks if the user typed n characters in the last five seconds. The count should be updated every second.
12. Write a TSR to show a clock in the upper right corner of the screen in the format HH:MM:SS.DD where HH is hours in 24 hour format, MM is minutes, SS is seconds and DD is hundredth of second. The clock should beep twice for one second each time with half a second interval in between at the start of every minute at a frequency of your choice.

    HINT: IBM PC uses a Real Time Clock (RTC) chip to keep track of time while switched off. It provides clock and calendar functions through its two I/O ports 70h and 71h. It is used as follows:

```
              mov  al, <command>
              out  0x70, al        ; command byte written at first port
              jmp  D1              ; waste one instruction time
D1:           in   al, 0x71        ; result of command is in AL now
```

    Following are few commands

    00 Get current second
    02 Get current minute
    04 Get current hour

    All numbers returned by RTC are in BCD. E.g. if it is 6:30 the second and third command will return 0x30 and 0x06 respectively in al.

3
Debug Interrupts

3.1. DEBUGGER USING SINGLE STEP INTERRUPT

The use of the trap flag has been deferred till now. The three flags not used for mathematical operations are the direction flag, the interrupt flag and the trap flag. The direction and interrupt flags have been previously discussed.

If the interrupt flag is set, the after every instruction a type 1 interrupt will be automatically generated. When the IVT and reserved interrupts were discussed this was named as the single step interrupt. This is like the divide by zero interrupt which was never explicitly invoked but it came itself. The single step interrupt behaves in the same manner.

The debugger is made using this interrupt. It allows one instruction to be executed and then return control to us. It has its display code and its code to wait for the key in the INT 1 handler. Therefore after every instruction the values of all registers are shown and the debugger waits for a key. Another interrupt used by the debugger is the break point interrupt INT 3. Apart from single stepping debugger has the breakpoint feature. INt 3 is used for this feature. INT 3 has a single byte opcode so it can replace any instruction. To put a breakpoint the instruction is replaced with INT 3 opcode and restored in the INT 3 handler. The INT 3 opcode is agained placed by a single step interrupt that is set up for this purpose after the replaced instruction has been executed.

There is no instruction to set or clear the trap flag like there are instructions for the interrupt and direction flags. We use twop special instructions PUSHF and POPF to push and pop the flag from the stack. We use PUSHF to place flags on the stack, change TF in this image on the stack and then reload into the flags register with POPF. The single step interrupt will come after the first instruction after POPF. The interrupt mechanism automatically clears IF and TF otherwise there would an infinite recursion of the single step interrupt. The TF is set in the flags on the stack so another interrupt will comes after one more instruction is executed after the return of the interrupt.

The following example is a very elementary debugger using the trap flag and the single step interrupt.

Example 10.1

```
; single stepping using the trap flag and single step interrupt
[org 0x0100]
              jmp start

flag:         db   0                  ; flag whether a key pressed
oldisr:       dd   0                  ; space for saving old ISR
names:        db   'FL =CS =IP =BP =AX =BX =CX =DX =SI =DI =DS =ES ='

;;;;; COPY LINES 008-025 FROM EXAMPLE 6.2 (clrscr) ;;;;;

; subroutine to print a number on screen
; takes the row no, column no, and number to be printed as parameters
printnum:     push bp
              mov  bp, sp
              push es
              push ax
              push bx
              push cx
```

```
                push dx
                push di

                mov  di, 80             ; load di with columns per row
                mov  ax, [bp+8]         ; load ax with row number
                mul  di                 ; multiply with columns per row
                mov  di, ax             ; save result in di
                add  di, [bp+6]         ; add column number
                shl  di, 1              ; turn into byte count
                add  di, 8              ; to end of number location

                mov  ax, 0xb800
                mov  es, ax             ; point es to video base
                mov  ax, [bp+4]         ; load number in ax
                mov  bx, 16             ; use base 16 for division
                mov  cx, 0              ; initialize count of digits

nextdigit:      mov  dx, 0              ; zero upper half of dividend
                div  bx                 ; divide by 10
                add  dl, 0x30           ; convert digit into ascii value
                cmp  dl, 0x39           ; is the digit an alphabet
                jbe  skipalpha          ; no, skip addition
                add  dl, 7              ; yes, make in alphabet code
skipalpha:      mov  dh, 0x07           ; attach normal attribute
                mov [es:di], dx         ; print char on screen
                sub  di, 2              ; to previous screen location
                loop nextdigit          ; if no divide it again

                pop  di
                pop  dx
                pop  cx
                pop  bx
                pop  ax
                pop  es
                pop  bp
                ret  6

; subroutine to print a string
; takes row no, column no, address of string, and its length
; as parameters
printstr:       push bp
                mov bp, sp
                push es
                push ax
                push bx
                push cx
                push dx
                push si
                push di

                mov  ax, 0xb800
                mov  es, ax             ; point es to video base

                mov  di, 80             ; load di with columns per row
                mov  ax, [bp+8]         ; load ax with row number
                mul  di                 ; multiply with columns per row
                mov  di, ax             ; save result in di
                add  di, [bp+6]         ; add column number
                shl  di, 1              ; turn into byte count

                mov  si, [bp+6]         ; string to be printed
                mov  cx, [bp+4]         ; length of string
                mov  ah, 0x07           ; normal attribute is fixed

nextchar:       mov  al, [si]           ; load next char of string
                mov  [es:di], ax        ; show next char on screen
                add  di, 2              ; move to next screen location
                add  si, 1              ; move to next char
                loop nextchar           ; repeat the operation cx times

                pop  di
                pop  si
                pop  dx
                pop  cx
                pop  bx
                pop  ax
```

```
                pop  es
                pop  bp
                ret  8

; keyboard interrupt service routine
kbisr:          push ax

                in   al, 0x60           ; read a char from keyboard port
                test al, 0x80           ; is it a press code
                jnz  skipflag           ; no, leave the interrupt
                add  byte [cs:flag], al ; yes, set flag to proceed

skipflag:       mov  al, 0x20
                out  0x20, al
                pop  ax
                iret

; single step interrupt service routine
trapisr:        push bp
                mov bp, sp              ; to read cs, ip and flags
                push ax
                push bx
                push cx
                push dx
                push si
                push di
                push ds
                push es

                sti                     ; waiting for keyboard interrupt
                push cs
                pop  ds                 ; initialize ds to data segment

                mov  byte [flag], 0     ; set flag to wait for key
                call clrscr             ; clear the screen

                mov  si, 6              ; first register is at bp+6
                mov  cx, 12             ; total 12 registers to print
                mov  ax, 0              ; start from row 0
                mov  bx, 5              ; print at column 5

l3:             push ax                 ; row number
                push bx                 ; column number
                mov  dx, [bp+si]
                push dx                 ; number to be printed
                call printnum           ; print the number
                sub  si, 2              ; point to next register
                inc  ax                 ; next row number
                loop l3                 ; repeat for the 12 registers

                mov  ax, 0              ; start from row 0
                mov  bx, 0              ; start from column 0
                mov  cx, 12             ; total 12 register names
                mov  si, 4              ; each name length is 4 chars
                mov  dx, names          ; offset of first name in dx

l1:             push ax                 ; row number
                push bx                 ; column number
                push dx                 ; offset of string
                push si                 ; length of string
                call printstr           ; print the string
                add  dx, 4              ; point to start of next string
                inc  ax                 ; new row number
                loop l1                 ; repeat for 12 register names

keywait:        cmp  byte [flag], 0     ; has a key been pressed
                je   keywait            ; no, check again

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

```
                pop  bp
                iret

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base
                mov ax, [es:9*4]
                mov [oldisr], ax        ; save offset of old routine
                mov ax, [es:9*4+2]
                mov [oldisr+2], ax      ; save segment of old routine
                mov word [es:1*4], trapisr ; store offset at n*4
                mov [es:1*4+2], cs      ; store segment at n*4+2
                cli                     ; disable interrupts
                mov  word [es:9*4], kbisr ; store offset at n*4
                mov  [es:9*4+2], cs     ; store segment at n*4+2
                sti                     ; enable interrupts

                pushf                   ; save flags on stack
                pop  ax                 ; copy flags in ax
                or   ax, 0x100          ; set bit corresponding to TF
                push ax                 ; save ax on stack
                popf                    ; reload into flags register

; the trap flag bit is on now, INT 1 will come after next instruction
; sample code to check the working of our elementary debugger
                mov  ax, 0
                mov  bx, 0x10
                mov  cx, 0x20
                mov  dx, 0x40

l2:             inc  ax
                add  bx, 2
                dec  cx
                sub  dx, 2
                jmp  l2
```

3.2. DEBUGGER USING BREAKPOINT INTERRUPT

We now write a debugger using INT 3. This debugger stops at the same point everytime where the breakpoint has been set up unlike the previous one which stopped at every instruction. The single step interrupt in this example is used only to restore the breakpoint interrupt which was removed by the breakpoint interrupt handler temporarily so that the original instruction can be executed.

Example 10.2

```
; elementary debugger using breakpoint interrupt
[org 0x0100]
                jmp start

flag:           db   0                  ; flag whether a key pressed
oldisr:         dd   0                  ; space for saving old ISR
names:          db   'FL =CS =IP =BP =AX =BX =CX =DX =SI =DI =DS =ES ='
opcode:         db 0
opcodepos:      dw 0

;;;;; COPY LINES 008-025 FROM EXAMPLE 6.2 (clrscr) ;;;;;
;;;;; COPY LINES 028-071 FROM EXAMPLE 10.1 (printnum) ;;;;;
;;;;; COPY LINES 073-114 FROM EXAMPLE 10.1 (printstr) ;;;;;
;;;;; COPY LINES 116-128 FROM EXAMPLE 10.1 (kbisr) ;;;;;

; single step interrupt service routine
trapisr:        push bp
                mov  bp, sp
                push ax
                push di
                push ds
                push es

                push cs
                pop  ds                 ; initialize ds to data segment
```

```
                mov  ax, [bp+4]
                mov  es, ax             ; load interrupted segment in es
                mov  di, [opcodepos]    ; load saved opcode position
                mov  byte [es:di], 0xCC ; reset the opcode to INT3
                and  word [bp+6], 0xFEFF; clear TF in flags on stack

                pop  es
                pop  ds
                pop  di
                pop  ax
                pop  bp
                iret

; breakpoint interrupt service routine
debugisr:       push bp
                mov  bp, sp             ; to read cs, ip and flags
                push ax
                push bx
                push cx
                push dx
                push si
                push di
                push ds
                push es

                sti                     ; waiting for keyboard interrupt
                push cs
                pop  ds                 ; initialize ds to data segment

                mov  ax, [bp+4]
                mov  es, ax             ; load interrupted segment in es
                dec  word [bp+2]        ; decrement the return address
                mov  di, [bp+2]         ; read the return address in di
                mov  word [opcodepos], di ; remember the return position
                mov  al, [opcode]       ; load the original opcode
                mov  [es:di], al        ; restore original opcode there

                mov  byte [flag], 0     ; set flag to wait for key
                call clrscr             ; clear the screen

                mov  si, 6              ; first register is at bp+6
                mov  cx, 12             ; total 12 registers to print
                mov  ax, 0              ; start from row 0
                mov  bx, 5              ; print at column 5

l3:             push ax                 ; row number
                push bx                 ; column number
                mov  dx, [bp+si]
                push dx                 ; number to be printed
                call printnum           ; print the number
                sub  si, 2              ; point to next register
                inc  ax                 ; next row number
                loop l3                 ; repeat for the 12 registers

                mov  ax, 0              ; start from row 0
                mov  bx, 0              ; start from column 0
                mov  cx, 12             ; total 12 register names
                mov  si, 4              ; each name length is 4 chars
                mov  dx, names          ; offset of first name in dx

l1:             push ax                 ; row number
                push bx                 ; column number
                push dx                 ; offset of string
                push si                 ; length of string
                call printstr           ; print the string
                add  dx, 4              ; point to start of next string
                inc  ax                 ; new row number
                loop l1                 ; repeat for 12 register names

                or word [bp+6], 0x0100  ; set TF in flags image on stack

keywait:        cmp  byte [flag], 0     ; has a key been pressed
                je   keywait            ; no, check again

                pop es
```

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

start:        xor  ax, ax
              mov  es, ax             ; point es to IVT base
              mov  word [es:1*4], trapisr ; store offset at n*4
              mov  [es:1*4+2], cs     ; store segment at n*4+2
              mov  word [es:3*4], debugisr ; store offset at n*4
              mov  [es:3*4+2], cs     ; store segment at n*4+2
              cli                     ; disable interrupts
              mov  word [es:9*4], kbisr ; store offset at n*4
              mov  [es:9*4+2], cs     ; store segment at n*4+2
              sti                     ; enable interrupts

              mov si, l2              ; load breakpoint position in si
              mov al, [cs:si]         ; read opcode at that position
              mov [opcode], al        ; save opcode for later use
              mov byte [cs:si], 0xCC  ; change opcode to INT3

; breakpoint is set now, INT3 will come at l2 on every iteration
; sample code to check the working of our elementary debugger
              mov ax, 0
              mov bx, 0x10
              mov cx, 0x20
              mov dx, 0x40

l2:           inc ax
              add bx, 2
              dec cx
              sub dx, 2
              jmp l2
```

4
Multitasking

4.1. CONCEPTS OF MULTITASKING

To experience the power of assembly language we introduce how to implement multitasking. We observed in the debugger that our thread of instructions was broken by the debugger; it got the control, used all registers, displayed an elaborate interface, waited for the key, and then restored processor state to what was immediately before interruption. Our program resumed as if nothing happened. The program execution was in the same logical flow.

If we have two different programs A and B. Program A is broken, its state saved, and returned to B instead of A. By looking at the instruction set, we can immediately say that nothing can stop us from doing that. IRET will return to whatever CS and IP it finds on the stack. Now B is interrupted somehow, its state saved, and we return back to A. A will have no way of knowing that it was interrupted as its entire environment has been restored. It never knew the debugger took control when it was debugged. It sill has no way of gaining this knowledge. If this work of breaking and restoring programs is done at high speed the user will feel that all the programs are running at the same time where actually they are being switched to and forth at high speed.

In essence multitasking is simple, even though we have to be extremely careful when implementing it. The environment of a program in the very simple case is all its registers and stack. We will deal with stack later. Now to get control from the program without the program knowing about it, we can use the IRQ 0 highest priority interrupt that is periodically coming to the processor.

Now we present a very basic example of multitasking. We have two subroutines written in assembly language. All the techiniques discussed here are applicable to code written in higher level languages as well. However the code to control this multitasking cannot be easily written in a higher level language so we write it in assembly language. The two subroutines rotate bars by changing characters at the two corners of the screen and have infinite loops. By hooking the timer interrupt and saving and restoring the registers of the tasks one by one, it appears that both tasks are running simultaneously.

Example 11.1

```
; elementary multitasking of two threads
[org 0x0100]
              jmp start

              ;   ax,bx,ip,cs,flags storage area
taskstates:   dw   0, 0, 0, 0, 0      ; task0 regs
              dw   0, 0, 0, 0, 0      ; task1 regs
              dw   0, 0, 0, 0, 0      ; task2 regs

current:      db   0                  ; index of current task
chars:        db   '\|/-'             ; shapes to form a bar

; one task to be multitasked
taskone:      mov  al, [chars+bx]     ; read the next shape
              mov  [es:0], al         ; write at top left of screen
```

```
                inc  bx                 ; increment to next shape
                and  bx, 3              ; taking modulus by 4
                jmp  taskone            ; infinite task

; second task to be multitasked
tasktwo:        mov  al, [chars+bx]     ; read the next shape
                mov  [es:158], al       ; write at top right of screen
                inc  bx                 ; increment to next shape
                and  bx, 3              ; taking modulus by 4
                jmp  tasktwo            ; infinite task

; timer interrupt service routine
timer:          push ax
                push bx

                mov  bl, [cs:current]   ; read index of current task
                mov  ax, 10             ; space used by one task
                mul  bl                 ; multiply to get start of task
                mov  bx, ax             ; load start of task in bx

                pop  ax                 ; read original value of bx
                mov  [cs:taskstates+bx+2], ax ; space for current task
                pop  ax                 ; read original value of ax
                mov  [cs:taskstates+bx+0], ax ; space for current task
                pop  ax                 ; read original value of ip
                mov  [cs:taskstates+bx+4], ax ; space for current task
                pop  ax                 ; read original value of cs
                mov  [cs:taskstates+bx+6], ax ; space for current task
                pop  ax                 ; read original value of flags
                mov  [cs:taskstates+bx+8], ax ; space for current task

                inc  byte [cs:current]  ; update current task index
                cmp  byte [cs:current], 3 ; is task index out of range
                jne  skipreset          ; no, proceed
                mov  byte [cs:current], 0 ; yes, reset to task 0

skipreset:      mov  bl, [cs:current]   ; read index of current task
                mov  ax, 10             ; space used by one task
                mul  bl                 ; multiply to get start of task
                mov  bx, ax             ; load start of task in bx

                mov  al, 0x20
                out  0x20, al           ; send EOI to PIC

                push word [cs:taskstates+bx+8] ; flags of new task
                push word [cs:taskstates+bx+6] ; cs of new task
                push word [cs:taskstates+bx+4] ; ip of new task
                mov  ax, [cs:taskstates+bx+0]  ; ax of new task
                mov  bx, [cs:taskstates+bx+2]  ; bx of new task
                iret                    ; return to new task

start:          mov  word [taskstates+10+4], taskone ; initialize ip
                mov  [taskstates+10+6], cs     ; initializae cs
                mov  word [taskstates+10+8], 0x0200 ; initialize flags
                mov  word [taskstates+20+4], tasktwo ; initialize ip
                mov  [taskstates+20+6], cs     ; initializae cs
                mov  word [taskstates+20+8], 0x0200 ; initialize flags
                mov  word [current], 0         ; set current task index

                xor  ax, ax
                mov  es, ax                    ; point es to IVT base
                cli
                mov  word [es:8*4], timer
                mov  [es:8*4+2], cs            ; hook timer interrupt
                mov  ax, 0xb800
                mov  es, ax                    ; point es to video base
                xor  bx, bx                    ; initialize bx for tasks
                sti

                jmp $                          ; infinite loop
```

The space where all registers of a task are stsored is called the process control block or PCB. Actual PCB contains a few more things that are not relevant to us now. INT 08 that is saving and restoring the registers is called the scheduler and the whole event is called a context switch.

4.2. ELABORATE MULTITASKING

In our next example we will save all 14 registers and the stack as well. 28 bytes are needed by these registers in the PCB. We add some more space to make the size 32, a power of 2 for easy calculations. One of these words is used to form a linked list of the PCBs so that strict ordering of active PCBs is not necessary. Also in this example we have given every thread its own stack. Now threads can have function calls, parameters and local variables etc. Another important change in this example is that the creation of threads is now dynamic. The thread registration code initializes the PCB, and adds it to the linked list so that the scheduler will give it a turn.

Example 11.2

```
; multitasking and dynamic thread registration
[org 0x0100]
              jmp start

; PCB layout:
; ax,bx,cx,dx,si,di,bp,sp,ip,cs,ds,ss,es,flags,next,dummy
;  0, 2, 4, 6, 8,10,12,14,16,18,20,22,24,  26 , 28 ,  30

pcb:          times 32*16 dw 0        ; space for 32 PCBs
stack:        times 32*256 dw 0       ; space for 32 512 byte stacks
nextpcb:      dw   1                  ; index of next free pcb
current:      dw   0                  ; index of current pcb
lineno:       dw   0                  ; line number for next thread

;;;;; COPY LINES 028-071 FROM EXAMPLE 10.1 (printnum) ;;;;;

; mytask subroutine to be run as a thread
; takes line number as parameter
mytask:       push bp
              mov  bp, sp
              sub  sp, 2              ; thread local variable
              push ax
              push bx

              mov  ax, [bp+4]         ; load line number parameter
              mov  bx, 70             ; use column number 70
              mov  word [bp-2], 0     ; initialize local variable

printagain:   push ax                 ; line number
              push bx                 ; column number
              push word [bp-2]        ; number to be printed
              call printnum           ; print the number
              inc  word [bp-2]        ; increment the local variable
              jmp  printagain         ; infinitely print

              pop  bx
              pop  ax
              mov  sp, bp
              pop  bp
              ret

; subroutine to register a new thread
; takes the segment, offset, of the thread routine and a parameter
; for the target thread subroutine
initpcb:      push bp
              mov  bp, sp
              push ax
              push bx
              push cx
              push si

              mov  bx, [nextpcb]      ; read next available pcb index
              cmp  bx, 32             ; are all PCBs used
              je   exit               ; yes, exit

```

```
                mov  cl, 5
                shl  bx, cl             ; multiply by 32 for pcb start

                mov  ax, [bp+8]         ; read segment parameter
                mov  [pcb+bx+18], ax    ; save in pcb space for cs
                mov  ax, [bp+6]         ; read offset parameter
                mov  [pcb+bx+16], ax    ; save in pcb space for ip

                mov  [pcb+bx+22], ds    ; set stack to our segment
                mov  si, [nextpcb]      ; read this pcb index
                mov  cl, 9
                shl  si, cl             ; multiply by 512
                add  si, 256*2+stack    ; end of stack for this thread
                mov  ax, [bp+4]         ; read parameter for subroutine
                sub  si, 2              ; decrement thread stack pointer
                mov  [si], ax           ; pushing param on thread stack
                sub  si, 2              ; space for return address
                mov  [pcb+bx+14], si    ; save si in pcb space for sp

                mov  word [pcb+bx+26], 0x0200 ; initialize thread flags
                mov  ax, [pcb+28]       ; read next of 0th thread in ax
                mov  [pcb+bx+28], ax    ; set as next of new thread
                mov  ax, [nextpcb]      ; read new thread index
                mov  [pcb+28], ax       ; set as next of 0th thread
                inc  word [nextpcb]     ; this pcb is now used

exit:           pop  si
                pop  cx
                pop  bx
                pop  ax
                pop  bp
                ret  6

; timer interrupt service routine
timer:          push ds
                push bx

                push cs
                pop  ds                 ; initialize ds to data segment

                mov  bx, [current]      ; read index of current in bx
                shl  bx, 1
                shl  bx, 1
                shl  bx, 1
                shl  bx, 1
                shl  bx, 1              ; multiply by 32 for pcb start
                mov  [pcb+bx+0], ax     ; save ax in current pcb
                mov  [pcb+bx+4], cx     ; save cx in current pcb
                mov  [pcb+bx+6], dx     ; save dx in current pcb
                mov  [pcb+bx+8], si     ; save si in current pcb
                mov  [pcb+bx+10], di    ; save di in current pcb
                mov  [pcb+bx+12], bp    ; save bp in current pcb
                mov  [pcb+bx+24], es    ; save es in current pcb

                pop  ax                 ; read original bx from stack
                mov  [pcb+bx+2], ax     ; save bx in current pcb
                pop  ax                 ; read original ds from stack
                mov  [pcb+bx+20], ax    ; save ds in current pcb
                pop  ax                 ; read original ip from stack
                mov  [pcb+bx+16], ax    ; save ip in current pcb
                pop  ax                 ; read original cs from stack
                mov  [pcb+bx+18], ax    ; save cs in current pcb
                pop  ax                 ; read original flags from stack
                mov  [pcb+bx+26], ax    ; save cs in current pcb
                mov  [pcb+bx+22], ss    ; save ss in current pcb
                mov  [pcb+bx+14], sp    ; save sp in current pcb

                mov  bx, [pcb+bx+28]    ; read next pcb of this pcb
                mov  [current], bx      ; update current to new pcb
                mov  cl, 5
                shl  bx, cl             ; multiply by 32 for pcb start

                mov  cx, [pcb+bx+4]     ; read cx of new process
                mov  dx, [pcb+bx+6]     ; read dx of new process
                mov  si, [pcb+bx+8]     ; read si of new process
                mov  di, [pcb+bx+10]    ; read diof new process
```

```
                mov  bp, [pcb+bx+12]    ; read bp of new process
                mov  es, [pcb+bx+24]    ; read es of new process
                mov  ss, [pcb+bx+22]    ; read ss of new process
                mov  sp, [pcb+bx+14]    ; read sp of new process

                push word [pcb+bx+26]   ; push flags of new process
                push word [pcb+bx+18]   ; push cs of new process
                push word [pcb+bx+16]   ; push ip of new process
                push word [pcb+bx+20]   ; push ds of new process

                mov  al, 0x20
                out  0x20, al           ; send EOI to PIC

                mov  ax, [pcb+bx+0]     ; read ax of new process
                mov  bx, [pcb+bx+2]     ; read bx of new process
                pop  ds                 ; read ds of new process
                iret                    ; return to new process

start:          xor  ax, ax
                mov  es, ax             ; point es to IVT base

                cli
                mov  word [es:8*4], timer
                mov  [es:8*4+2], cs     ; hook timer interrupt
                sti

nextkey:        xor ah, ah              ; service 0 – get keystroke
                int 0x16                ; bios keyboard services

                push cs                 ; use current code segment
                mov  ax, mytask
                push ax                 ; use mytask as offset
                push word [lineno]      ; thread parameter
                call initpcb            ; register the thread

                inc  word [lineno]      ; update line number
                jmp  nextkey            ; wait for next keypress
```

When the program is executed the threads display the numbers independently. However as keys are pressed and new threads are registered, there is an obvious slowdown in the speed of multitasking. To improve that, we can change the timer interrupt frequency. The following can be used to set to an approximately 1ms interval.

```
mov  ax, 1100
out  0x40, al
mov  al, ah
out  0x40, al
```

This makes the threads look faster. However the only real change is that the timer interrupt is now coming more frequently.

4.3. MULTITASKING KERNEL AS TSR

The above examples had the multitasking code and the multitasked code in one program. Now we separate the multitasking kernel into a TSR so that it becomes an operation system extension. We hook a software interrupt for the purpose of registering a new thread.

Example 11.3

```
; multitasking kernel as a TSR
[org 0x0100]
                jmp start

; PCB layout:
; ax,bx,cx,dx,si,di,bp,sp,ip,cs,ds,ss,es,flags,next,dummy
;  0, 2, 4, 6, 8,10,12,14,16,18,20,22,24,  26 , 28 ,  30

pcb:            times 32*16 dw 0        ; space for 32 PCBs
```

```
stack:        times 32*256 dw 0       ; space for 32 512 byte stacks
nextpcb:      dw   1                  ; index of next free pcb
current:      dw   0                  ; index of current pcb

;;;;; COPY LINES 133-192 FROM EXAMPLE 11.2 (timer) ;;;;;

; software interrupt to register a new thread
; takes parameter block in ds:si
; parameter block has cs, ip, ds, es, and param in this order
initpcb:      push ax
              push bx
              push cx
              push di

              mov  bx, [cs:nextpcb]   ; read next available pcb index
              cmp  bx, 32             ; are all PCBs used
              je   exit               ; yes, exit

              mov  cl, 5
              shl  bx, cl             ; multiply by 32 for pcb start

              mov  ax, [si+0]         ; read code segment parameter
              mov  [cs:pcb+bx+18], ax ; save in pcb space for cs
              mov  ax, [si+2]         ; read offset parameter
              mov  [cs:pcb+bx+16], ax ; save in pcb space for ip
              mov  ax, [si+4]         ; read data segment parameter
              mov  [cs:pcb+bx+20], ax ; save in pcb space for ds
              mov  ax, [si+6]         ; read extra segment parameter
              mov  [cs:pcb+bx+24], ax ; save in pcb space for es

              mov  [cs:pcb+bx+22], cs ; set stack to our segment
              mov  di, [cs:nextpcb]   ; read this pcb index
              mov  cl, 9
              shl  di, cl             ; multiply by 512
              add  di, 256*2+stack    ; end of stack for this thread
              mov  ax, [si+8]         ; read parameter for subroutine
              sub  di, 2              ; decrement thread stack pointer
              mov  [cs:di], ax        ; pushing param on thread stack
              sub  di, 4              ; space for far return address
              mov  [cs:pcb+bx+14], di ; save di in pcb space for sp

              mov  word [cs:pcb+bx+26], 0x0200 ; initialize flags
              mov  ax, [cs:pcb+28]    ; read next of 0th thread in ax
              mov  [cs:pcb+bx+28], ax ; set as next of new thread
              mov  ax, [cs:nextpcb]   ; read new thread index
              mov  [cs:pcb+28], ax    ; set as next of 0th thread
              inc  word [cs:nextpcb]  ; this pcb is now used

exit:         pop  di
              pop  cx
              pop  bx
              pop  ax
              iret

start:        xor ax, ax
              mov es, ax              ; point es to IVT base

              mov word [es:0x80*4], initpcb
              mov [es:0x80*4+2], cs   ; hook software int 80
              cli
              mov word [es:0x08*4], timer
              mov [es:0x08*4+2], cs   ; hook timer interrupt
              sti

              mov dx, start
              add dx, 15
              mov cl, 4
              shr dx, cl

              mov ax, 0x3100          ; terminate and stay resident
              int 0x21
```

The second part of our example is a simple program that has the threads to be registered with the multitasking kernel using its exported services.

Example 11.4

```
; multitasking TSR caller
[org 0x0100]
              jmp start

; parameter block layout:
; cs,ip,ds,es,param
;  0, 2, 4, 6,  8

paramblock:   times 5 dw 0            ; space for parameters
lineno:       dw   0                  ; line number for next thread

;;;;; COPY LINES 028-071 FROM EXAMPLE 10.1 (printnum) ;;;;;

; subroutine to be run as a thread
; takes line number as parameter
mytask:       push bp
              mov  bp, sp
              sub  sp, 2              ; thread local variable
              push ax
              push bx

              mov  ax, [bp+4]         ; load line number parameter
              mov  bx, 70             ; use column number 70
              mov  word [bp-2], 0     ; initialize local variable

printagain:   push ax                 ; line number
              push bx                 ; column number
              push word [bp-2]        ; number to be printed
              call printnum           ; print the number
              inc  word [bp-2]        ; increment the local variable
              jmp  printagain         ; infinitely print

              pop  bx
              pop  ax
              mov  sp, bp
              pop  bp
              retf

start:        mov  ah, 0              ; service 0 – get keystroke
              int  0x16               ; bios keyboard services

              mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
              mov  [paramblock+6], es ; extra segment parameter
              mov  ax, [lineno]
              mov  [paramblock+8], ax ; parameter for thread
              mov  si, paramblock     ; address of param block in si
              int  0x80               ; multitasking kernel interrupt

              inc  word [lineno]      ; update line number
              jmp  start              ; wait for next key
```

We introduce yet another use of the multitasking kernel with this new example. In this example three different sort of routines are multitasked by the same kernel instead of repeatedly registering the same routine.

Example 11.5

```
; another multitasking TSR caller
[org 0x0100]
              jmp start

; parameter block layout:
; cs,ip,ds,es,param
;  0, 2, 4, 6,  8

paramblock:   times 5 dw 0            ; space for parameters
lineno:       dw   0                  ; line number for next thread
chars:        db '\|/-'               ; chracters for rotating bar
message:      db 'moving hello'       ; moving string
message2:     db '            '       ; to erase previous string
```

```
messagelen:   dw 12                    ; length of above strings

;;;;; COPY LINES 028-071 FROM EXAMPLE 10.1 (printnum) ;;;;;
;;;;; COPY LINES 073-114 FROM EXAMPLE 10.1 (printstr) ;;;;;

; subroutine to run as first thread
mytask:       push bp
              mov  bp, sp
              sub  sp, 2               ; thread local variable
              push ax
              push bx

              xor  ax, ax              ; use line number 0
              mov  bx, 70              ; use column number 70
              mov  word [bp-2], 0      ; initialize local variable

printagain:   push ax                  ; line number
              push bx                  ; column number
              push word [bp-2]         ; number to be printed
              call printnum            ; print the number
              inc  word [bp-2]         ; increment the local variable
              jmp  printagain          ; infinitely print

              pop  bx
              pop  ax
              mov  sp, bp
              pop  bp
              retf

; subroutine to run as second thread
mytask2:      push ax
              push bx
              push es

              mov  ax, 0xb800
              mov  es, ax              ; point es to video base
              xor  bx, bx              ; initialize to use first shape

rotateagain:  mov  al, [chars+bx]      ; read current shape
              mov  [es:40], al         ; print at specified place
              inc  bx                  ; update to next shape
              and  bx, 3               ; take modulus with 4
              jmp  rotateagain         ; repeat infinitely

              pop  es
              pop  bx
              pop  ax
              retf

; subroutine to run as third thread
mytask3:      push bp
              mov  bp, sp
              sub  sp, 2               ; thread local variable
              push ax
              push bx
              push cx

              mov  word [bp-2], 0      ; initialize line number to 0

nextline:     push word [bp-2]         ; line number
              mov  bx, 50
              push bx                  ; column number 50
              mov  ax, message
              push ax                  ; offset of string
              push word [messagelen]   ; length of string
              call printstr            ; print the string

              mov  cx, 0x100
waithere:     push cx                  ; save outer loop counter
              mov  cx, 0xffff
              loop $                   ; repeat ffff times
              pop  cx                  ; restore outer loop counter
              loop waithere            ; repeat 0x100 times

              push word [bp-2]         ; line number
              mov  bx, 50              ; column number 50
```

```
              push bx
              mov  ax, message2
              push ax                ; offset of blank string
              push word [messagelen]  ; length of string
              call printstr          ; print the string

              inc  word [bp-2]       ; update line number
              cmp  word [bp-2], 25   ; is this the last line
              jne  skipreset         ; no, proceed to draw
              mov  word [bp-2], 0    ; yes, reset line number to 0

skipreset:    jmp  nextline          ; proceed with next drawing

              pop  cx
              pop  bx
              pop  ax
              mov  sp, bp
              pop  bp
              retf

start:        mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
              mov  [paramblock+6], es ; extra segment parameter
              mov  word [paramblock+8], 0 ; parameter for thread
              mov  si, paramblock    ; address of param block in si
              int  0x80              ; multitasking kernel interrupt

              mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask2 ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
              mov  [paramblock+6], es ; extra segment parameter
              mov  word [paramblock+8], 0 ; parameter for thread
              mov  si, paramblock    ; address of param block in si
              int  0x80              ; multitasking kernel interrupt

              mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask3 ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
              mov  [paramblock+6], es ; extra segment parameter
              mov  word [paramblock+8], 0 ; parameter for thread
              mov  si, paramblock    ; address of param block in si
              int  0x80              ; multitasking kernel interrupt

              jmp $
```

EXERCISES

1. Change the multitasking kernel such that a new two byte variable is introduced in the PCB. This variable contains the number of turns this process should be given. For example if the first PCB contains 20 in this variable, the switch to second process should occur after 20 timer interrupts (approx one second at default speed) and similarly the switch from second to third process should occur after the number given in the second process's PCB.
2. Change the scheduler of the multitasking kernel to enque the current process index a ready queue, and dequeue the next process index from it, and assign it to current. Therefore the next field of the PCB is no longer used. Use queue functions from Exercise 5.XX.
3. Add a function in the multitasking kernel to fork the current process through a software interrupt. Fork should allocate a new PCB and copy values of all registers of the caller's PCB to the new PCB. It should allocate a stack and change SS, SP appropriately in the new PCB. It has to copy the caller's stack on the newly allocated stack. It will set AX in the new PCB to 0 and in the old PB to 1 so that both threads can identify which is the creator and which is the created process and can act accordingly.

4. Add a function in the multitasking kernel accessible via a software interrupt that allows the current process to terminate itself.
5. Create a queue in the multitasking kernel called kbQ. This queue initially empty will contain characters typed by the user. Hook the keyboard interrupt for getting user keys. Convert the scan code to ASCII if the key is from a-z or 0-9 and enque it in kbQ. Ignore all other scan codes. Write a function checkkey accessible via a software interrupt that returns the process in AX a value removed from the queue. It waits if there is no key in the queue. Be aware of enabling interrupts if you wait here.
6. Modify the multitasking kernel such that the initial process displays at the last line of the screen whatever is typed by the user and clears that line on enter. If the user types quit followed by enter restore everything to normal as it was before the multitasking kernel was there. If the user types start followed by enter, start one more rotating bar on the screen. The first rotating bar should appear in the upper left, the next in the second column, then third and so on. The bar color should be white. The user can type the commands 'white', 'red', and 'green' to change the color of new bars.

5
Video Services

5.1. BIOS VIDEO SERVICES

The Basic Input Output System (BIOS) provides services for video, keyboard, serial port, parallel port, time etc. The video services are exported via INT 10. We will discuss some very simple services. Video services are classified into two broad categories; graphics mode services and text mode services. In graphics mode a location in video memory corresponds to a dot on the screen. In text mode this relation is not straightforward. The video memory holds the ASCII of the character to be shown and the actual shape is read from a font definition stored elsewhere in memory. We first present a list of common video services used in text mode.

```
INT 10 - VIDEO - SET VIDEO MODE
AH = 00h
AL = desired video mode
```

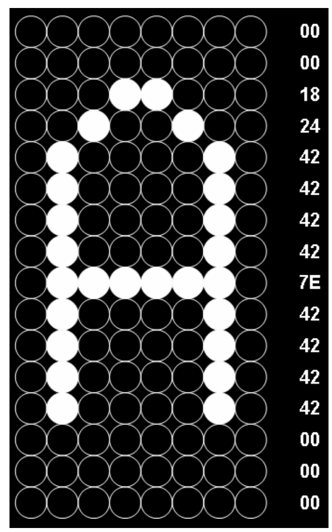
Some common video modes include 40x25 text mode (mode 0), 80x25 text mode (mode 2), 80x50 text mode (mode 3), and 320x200 graphics mode (mode D).

```
INT 10 - VIDEO - SET TEXT-MODE CURSOR SHAPE
AH = 01h
CH = cursor start and options
CL = bottom scan line containing cursor (bits 0-4)
INT 10 - VIDEO - SET CURSOR POSITION
AH = 02h
BH = page number
    0-3 in modes 2&3
    0-7 in modes 0&1
    0 in graphics modes
DH = row (00h is top)
DL = column (00h is left)
INT 10 - VIDEO - SCROLL UP WINDOW
AH = 06h
AL = number of lines by which to scroll up (00h = clear entire window)
BH = attribute used to write blank lines at bottom of window
CH, CL = row, column of window's upper left corner
DH, DL = row, column of window's lower right corner
INT 10 - VIDEO - SCROLL DOWN WINDOW
AH = 07h
AL = number of lines by which to scroll down (00h=clear entire window)
BH = attribute used to write blank lines at top of window
CH, CL = row, column of window's upper left corner
DH, DL = row, column of window's lower right corner
INT 10 - VIDEO - WRITE CHARACTER AND ATTRIBUTE AT CURSOR POSITION
AH = 09h
AL = character to display
BH = page number
```

```
BL = attribute (text mode) or color (graphics mode)
CX = number of times to write character
```
INT 10 - VIDEO - WRITE CHARACTER ONLY AT CURSOR POSITION
```
AH = 0Ah
AL = character to display
BH = page number
BL = attribute (text mode) or color (graphics mode)
CX = number of times to write character
```
INT 10 - VIDEO - WRITE STRING
```
AH = 13h
AL = write mode
   bit 0: update cursor after writing
   bit 1: string contains alternating characters and attributes
   bits 2-7: reserved (0)
BH = page number
BL = attribute if string contains only characters
CX = number of characters in string
DH, DL = row, column at which to start writing
ES:BP -> string to write
```

Chargen Services

In our first example we will read the font definition in memory and change it to include a set of all on pixels in the last line showing an effect of underline on all character including space. An 8x16 font is stored in 16 bytes. A sample character and the corresponding 16 values stored in the font information are shown for the character ‘A’. We start with two services from the chargen subset of video services that we are going to use.



INT 10 - VIDEO - GET FONT INFORMATION
```
AX = 1130h
BH = pointer specifier
Return:
ES:BP = specified pointer
CX = bytes/character of on-screen font
DL = highest character row on screen
```
INT 10 - TEXT-MODE CHARGEN
```
AX = 1110h
ES:BP -> user table
CX = count of patterns to store
DX = character offset into map 2 block
BL = block to load in map 2
BH = number of bytes per character pattern
```

We will use 6 as the pointer specifier which means the 8x16 font stored in ROM.

Example 12.1

```
; put underlines on screen font
[org 0x0100]
              jmp  start

font:         times 256*16 db 0       ; space for font

start:        mov  ax, 0x1130         ; service 11/30 – get font info
              mov  bx, 0x0600         ; ROM 8x16 font
              int  0x10               ; bios video services

              mov  si, bp             ; point si to rom font data
              mov  di, font           ; point di to space for font
```

```
                mov  cx, 256*16         ; font size
                push ds
                push es
                pop  ds                 ; ds:si to rom font data
                pop  es                 ; es:di to space for font
                cld                     ; auto increment mode
                rep  movsb              ; copy font

                push cs
                pop  ds                 ; restore ds to data segment

                mov  si, font-1         ; point si before first char
                mov  cx, 0x100          ; total 256 characters
change:         add  si, 16             ; one character has 16 bytes
                mov  byte [si], 0xFF    ; change last line to all ones
                loop change             ; repeat for each character

                mov  bp, font           ; es:bp points to new font
                mov  bx, 0x1000         ; bytes per char & block number
                mov  cx, 0x100          ; number of characters to change
                xor  dx, dx             ; first character to change
                mov  ax, 0x1110         ; service 11/10 – load user font
                int  0x10               ; bios video services

                mov ax, 0x4c00          ; terminate program
                int 0x21
```

Our second example is similar to the last example however in this case we are doing something funny on the screen. We are reversing the shapes of all the characters on the screen.

Example 12.2

```
; reverse each character of screen font
[org 0x0100]
                jmp  start

font:           times 256*16 db 0       ; space for font

start:          mov  ax, 0x1130         ; service 11/30 – get font info
                mov  bx, 0x0600         ; ROM 8x16 font
                int  0x10               ; bios video services

                mov  si, bp             ; point si to rom font data
                mov  di, font           ; point di to space for font
                mov  cx, 256*16         ; font size
                push ds
                push es
                pop  ds                 ; ds:si to rom font data
                pop  es                 ; es:di to space for font
                cld                     ; auto increment mode
                rep  movsb              ; copy font

                push cs
                pop  ds                 ; restore ds to data segment

                mov si, font            ; point si to start of font
change:         mov al, [si]            ; read one byte
                mov cx, 8
inner:          shl al, 1               ; shift left with MSB in carry
                rcr bl, 1               ; rotate right using carry
                loop inner              ; repeat eight times
                mov [si], bl            ; write back reversed byte
                inc si                  ; next byte of font
                cmp si, font+256*16     ; is whole font reversed
                jne change              ; no, reverse next byte

                mov  bp, font           ; es:bp points to new font
                mov  bx, 0x1000         ; bytes per char & block number
                mov  cx, 0x100          ; number of characters to change
                xor  dx, dx             ; first character to change
                mov  ax, 0x1110         ; service 11/10 – load user font
                int  0x10               ; bios video services

```

```
                mov  ax, 0x4c00         ; terminate program
                int  0x21
```

Graphics Mode Services

We will take an example of using graphics mode video services as well. We will draw a line across the screen using the following service.

```
INT 10 - VIDEO - WRITE GRAPHICS PIXEL
AH = 0Ch
BH = page number
AL = pixel color
CX = column
DX = row
```

Example 12.3

```
; draw line in graphics mode
[org 0x0100]
                mov  ax, 0x000D         ; set 320x200 graphics mode
                int  0x10               ; bios video services

                mov  ax, 0x0C07         ; put pixel in white color
                xor  bx, bx             ; page number 0
                mov  cx, 200            ; x position 200
                mov  dx, 200            ; y position 200

l1:             int  0x10               ; bios video services
                dec  dx                 ; decrease y position
                loop l1                 ; decrease x position and repeat

                mov  ah, 0              ; service 0 – get keystroke
                int  0x16               ; bios keyboard services

                mov  ax, 0x0003         ; 80x25 text mode
                int  0x10               ; bios video services

                mov  ax, 0x4c00         ; terminate program
                int  0x21
```

5.2. DOS VIDEO SERVICES

Services of DOS are more cooked and at a higher level than BIOS. They provide less control but make routine tasks much easier. Some important DOS services are listed below.

```
INT 21 - READ CHARACTER FROM STANDARD INPUT, WITH ECHO
AH = 01h
Return: AL = character read
INT 21 - WRITE STRING TO STANDARD OUTPUT
AH = 09h
DS:DX -> $ terminated string
INT 21 - BUFFERED INPUT
AH = 0Ah
DS:DX -> dos input buffer
```

The DOS input buffer has a special format where the first btye stores the maximum characters buffer can hold, the second byte holds the number of characters actually read on return, and the following space is used for the actual characters read. We start will an example of reading a string with service 1 and displaying it with service 9.

Example 12.4

```
; character input using dos services
[org 0x0100]
              jmp start

maxlength:    dw   80                 ; maximum length of input
message:      db   10, 13, 'hello $'  ; greetings message
buffer:       times 81 db 0           ; space for input string

start:        mov  cx, [maxlength]    ; load maximum length in cx
              mov  si, buffer         ; point si to start of buffer

nextchar:     mov  ah, 1              ; service 1 – read character
              int  0x21               ; dos services

              cmp  al, 13             ; is enter pressed
              je   exit               ; yes, leave input
              mov  [si], al           ; no, save this character
              inc  si                 ; increment buffer pointer
              loop nextchar           ; repeat for next input char

exit:         mov  byte [si], '$'     ; append $ to user input

              mov  dx, message        ; greetings message
              mov  ah, 9              ; service 9 – write string
              int  0x21               ; dos services

              mov  dx, buffer         ; user input buffer
              mov  ah, 9              ; service 9 – write string
              int  0x21               ; dos services

              mov  ax, 0x4c00         ; terminate program
              int  0x21
```

Our next example uses the more cooked buffered input service of DOS and using the same service 9 to print the string.

Example 12.5

```
; buffer input using dos services
[org 0x0100]
              jmp  start

message:      db   10,13,'hello ', 10, 13, '$'
buffer:       db   80                 ; length of buffer
              db   0                  ; number of character on return
              times 80 db 0           ; actual buffer space

start:        mov  dx, buffer         ; input buffer
              mov  ah, 0x0A           ; service A – buffered input
              int  0x21               ; dos services

              mov  bh, 0
              mov  bl, [buffer+1]     ; read actual size in bx
              mov  byte [buffer+2+bx], '$' ; append $ to user input

              mov  dx, message        ; greetings message
              mov  ah, 9              ; service 9 – write string
              int  0x21               ; dos services

              mov  dx, buffer+2       ; user input buffer
              mov  ah, 9              ; service 9 – write string
              int  0x21               ; dos services

              mov  ax, 0x4c00         ; terminate program
              int  0x21
```

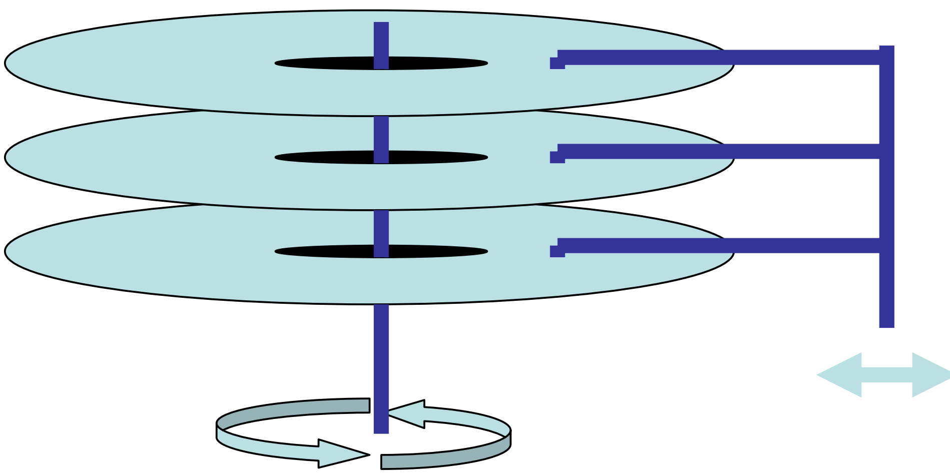
More detail of DOS and BIOS interrupts is available in the Raplh Brown Interrupt List.

6
Secondary Storage

6.1. PHYSICAL FORMATION

A floppy disk is a circular plate with a fine coating of magnetic material over it. The plate is enclosed in a plastic jacket which has a cover that can slide to expose the magnetic surface. The drive motor attaches itself to the central piece and rotatets the plate. Two heads on both sides can read the magnetically encoded data on the disk.

If the head is fixed and the motor rotates the disk the readable area on the disk surface forms a circle called a track. Head moved to the next step forms another track and so on. In hard disks the same structure is extended to a larger number of tracks and plates. The tracks are further cut vertically into sectors. This is a logical division of the area on the tracks. Each sector holds 512 bytes of data. A standard floppy disk has 80 tracks and 18 sectors per track with two heads, one on each side totallying to 2880 sectors or 1440 KB of data. Hard disks have varying number of heads and tracks pertaining to their different capacities.



BIOS sees the disks as a combination of sectors, tracks, and heads, as a raw storage device without concern to whether it is reading a file or directory. BIOS provides the simplest and most powerful interface to the storage medium. However this raw storage is meaningless to the user who needs to store his files and organize them into directories. DOS builds a logical structure on this raw storage space to provide these abstractions. This logical formation is read and interpreted by DOS. If another file system is build on the same storage medium the interpretations change. Main units of the DOS structure are the boot sector in head 0, track 0, and sector 1, the first FAT starting from head 0, track 0, sector 2, the second copy of FAT starting from head 0, track 0, sector 11, and the root directory starting from head 1, track 0, sector 2. The area from head 0, track 1, sector 16 to head 1, track 79, sector 18 is used for storing the data of the files. Among this we will be exploring the directory structure further. The 32 sectors reserved for the root directory contain 512 directory entries. The format of a 32 byte directory entry is shown below.

```
+00 Filename (8 bytes)
+08 Extension (3 bytes)
+0B Flag Byte (1 byte)
+0C Reserved (1 byte)
+0D Creation Date/Time (5 bytes)
+12 Last Accessed Data (2 bytes)
+14 Starting Cluster High Word (2 bytes) for FAT32
+16 Time (2 bytes)
+18 Date (2 bytes)
+1A Starting Cluster Low Word (2 bytes)
+1C File Size (4 bytes)
```

6.2. STORAGE ACCESS USING BIOS

We will be using BIOS disk services to directly see the data stored in the directory entries by DOS. For this purpose we will be using the BIOS disk services.

```
INT 13 - DISK - RESET DISK SYSTEM
AH = 00h
DL = drive
Return:
CF = error flag
AH = error code
INT 13 - DISK - READ SECTOR(S) INTO MEMORY
AH = 02h
AL = number of sectors to read (must be nonzero)
CH = low eight bits of cylinder number
CL = sector number 1-63 (bits 0-5)
     high two bits of cylinder (bits 6-7, hard disk only)
DH = head number
DL = drive number (bit 7 set for hard disk)
ES:BX -> data buffer
Return:
CF = error flag
AH = error code
AL = number of sectors transferred
INT 13 - DISK - WRITE DISK SECTOR(S)
AH = 03h
AL = number of sectors to write (must be nonzero)
CH = low eight bits of cylinder number
CL = sector number 1-63 (bits 0-5)
     high two bits of cylinder (bits 6-7, hard disk only)
DH = head number
DL = drive number (bit 7 set for hard disk)
ES:BX -> data buffer
Return:
CF = error flag
AH = error code
AL = number of sectors transferred
INT 13 - DISK - GET DRIVE PARAMETERS
AH = 08h
DL = drive (bit 7 set for hard disk)
Return:
CF = error flag
AH = error code
CH = low eight bits of maximum cylinder number
CL = maximum sector number (bits 5-0)
```

```
     high two bits of maximum cylinder number (bits 7-6)
DH = maximum head number
DL = number of drives
ES:DI -> drive parameter table (floppies only)
```

Example 13.1

```
; floppy directory using bios services
[org 0x0100]
              jmp  start

sector:       times 512 db 0          ; space for directory sector
entryname:    times 11 db 0           ; space for a file name
              db   10, 13, '$'        ; new line and terminating $

start:        mov  ah, 0              ; service 0 – reset disk system
              mov  dl, 0              ; drive A:
              int  0x13               ; bios disk services
              jc   error              ; if error, terminate program

              mov  ah, 2              ; service 2 – read sectors
              mov  al, 1              ; count of sectors
              mov  ch, 0              ; cyliner
              mov  cl, 2              ; sector
              mov  dh, 1              ; head
              mov  dl, 0              ; drive A:
              mov  bx, sector         ; buffer to read sector
              int  0x13               ; bios disk services
              jc   error              ; if error, terminate program

              mov  bx, 0              ; start from first entry
nextentry:    mov  di, entryname      ; point di to space for filename
              mov  si, sector         ; point si to sector
              add  si, bx             ; move ahead to desired entry
              mov  cx, 11             ; one filename is 11 bytes long
              cld                     ; auto increment mode
              rep  movsb              ; copy filename

              mov  ah, 9              ; service 9 – output string
              mov  dx, entryname      ; filename to be printed
              int  0x21               ; dos services

              add  bx, 32             ; point to next dir entry
              cmp  bx, 512            ; is last entry in this sector
              jne  nextentry          ; no, print next entry

error:        mov  ax, 0x4c00         ; terminate program
              int  0x21
```

With the given services and the bits allocated for heads, tracks, and sectors only 8GB disks can be accessed. This limitation can be overcome by using INT 13 extensions that take a linear 64bit sector number and handle all the head, track, sector conversion themselves. The important services in this category are listed below.

INT 13 - INT 13 Extensions - EXTENDED READ

```
AH = 42h
DL = drive number
DS:SI -> disk address packet
Return:
CF = error flag
AH = error code
     disk address packet's block count field set to number of blocks
     successfully transferred
```

INT 13 - INT 13 Extensions - EXTENDED WRITE

```
AH = 43h
AL = write flags
```

```
DL = drive number
DS:SI -> disk address packet
Return:
CF = error flag
AH = error code
     disk address packet's block count field set to number of blocks
     successfully transferred
```

The format of the disk address packet used above is as follows.

```
Offset  Size   Description
 00h    BYTE   size of packet = 10h
 01h    BYTE   reserved (0)
 02h    WORD   number of blocks to transfer
 04h    DWORD  -> transfer buffer
 08h    QWORD  starting absolute block number
```

Hard disks have a different formation from floppy disks in that there is a partition table at the start that allows several logical disks to be maintained within a single physical disk. The physical sector 0 holds the master boot record and a partition table towards the end. The first 446 bytes contain MBR, then there are 4 16 byte partition entries and then there is a 2 byte signature. A partition table entry has the following format.

```
Byte 0 – 0x80 for active 0x00 for inactive
Byte 1-3 – Starting CHS
Byte 4 – Partition Type
Byte 5-7 – Ending CHS
Byte 8-B – Starting LBA
Byte C-F – Size of Partition
```

Some important partition types are listed below.

```
00 Unused Entry
01 FAT12
05 Extended Partition
06 FAT16
0b FAT32
0c FAT32 LBA
0e FAT16 LBA
0f Extended LBA
07 NTFS
```

Extended partition type signals that the specified area is treated as a complete hard disk with its own partition table and partitions. Therefore extended partitions allow a recursion in partitioning and consequently an infinite number of partitions are possible. The following program reads the partition tables (primary and extended) using recursion and displays in an indented form all partitions present on the first hard disk in the system.

Example 13.2

```
; a program to display the partition table
[org 0x0100]
              jmp  start

dap:          db   0x10, 0        ; disk address packet
              dw   1
              dd   0, 0, 0
```

```

msg:          times 17 db ' '
              db   10, 13, '$'
fat12:        db   'FAT12...$'
fat16:        db   'FAT16...$'
fat32:        db   'FAT32...$'
ntfs:         db   'NTFS....$'
extended:     db   'EXTEND..$'
unknown:      db   'UNKNOWN.$'

partypes:     dw   0x1, fat12     ; table of known partition types
              dw   0x5, extended
              dw   0x6, fat16
              dw   0xe, fat16
              dw   0xb, fat32
              dw   0xc, fat32
              dw   0x7, ntfs
              dw   0xf, extended
              dw   0x0, unknown

; subroutine to print a number in a string as hex
; takes address of string and a 16bit number as parameter
printnum:     push bp
              mov  bp, sp
              push ax
              push bx
              push cx
              push dx
              push di

              mov  di, [bp+6]          ; string to store the number
              add  di, 3

              mov  ax, [bp+4]          ; load number in ax
              mov  bx, 16              ; use base 16 for division
              mov  cx, 4

nextdigit:    mov  dx, 0
              div  bx                  ; divide by 16
              add  dl, 0x30            ; convert into ascii value
              cmp  dl, 0x39
              jbe  skipalpha

              add  dl, 7

skipalpha:    mov  [di], dl            ; update char in string
              dec  di
              loop nextdigit

              pop  di
              pop  dx
              pop  cx
              pop  bx
              pop  ax
              pop  bp
              ret  4

; subroutine to print the start and end of a partition
; takes the segment and offset of the partition table entry
printpart:    push bp
              mov  bp, sp
              push es
              push ax
              push di

              les  di, [bp+4]          ; point es:di to dap

              mov  ax, msg
              push ax
              push word [es:di+0xA]
              call printnum            ; print first half of start

              add  ax, 4
              push ax
              push word [es:di+0x8]
              call printnum            ; print second half of start
```

```
                add  ax, 5
                push ax
                push word [es:di+0xE]
                call printnum            ; print first half of end

                add  ax, 4
                push ax
                push word [es:di+0xC]
                call printnum            ; print second half of end

                mov  dx, msg
                mov  ah, 9
                int  0x21                ; print the whole on the screen

                pop  di
                pop  ax
                pop  es
                pop  bp
                ret  4

; recursive subroutine to read the partition table
; take indentation level and 32bit absolute block number as parameters
readpart:       push bp
                mov  bp, sp
                sub  sp, 512             ; local space to read sector
                push ax
                push bx
                push cx
                push dx
                push si

                mov  ax, bp
                sub  ax, 512
                mov  word [dap+4], ax    ; init dest offset in dap
                mov  [dap+6], ds         ; init dest segment in dap
                mov  ax, [bp+4]
                mov  [dap+0x8], ax       ; init sector no in dap
                mov  ax, [bp+6]
                mov  [dap+0xA], ax       ; init second half of sector no

                mov  ah, 0x42            ; read sector in LBA mode
                mov  dl, 0x80            ; first hard disk
                mov  si, dap             ; address of dap
                int  0x13                ; int 13

                jc   failed              ; if failed, leave

                mov  si, -66             ; start of partition info
nextpart:       mov  ax, [bp+4]          ; read relative sector number
                add  [bp+si+0x8], ax     ; make it absolute
                mov  ax, [bp+6]          ; read second half
                adc  [bp+si+0xA], ax     ; make seconf half absolute

                cmp  byte [bp+si+4], 0   ; is partition unused
                je   exit

                mov  bx, partypes        ; point to partition types
                mov  di, 0
nextmatch:      mov  ax, [bx+di]
                cmp  [bp+si+4], al       ; is this partition known
                je   found               ; yes, so print its name
                add  di, 4               ; no, try next entry in table
                cmp  di, 32              ; are all entries compared
                jne  nextmatch           ; no, try another

found:          mov  cx, [bp+8]          ; load indentation level
                jcxz noindent            ; skip if no indentation needed
indent:         mov  dl, ' '
                mov  ah, 2               ; display char service
                int  0x21                ; dos services
                loop indent              ; print required no of spaces

noindent:       add  di, 2
                mov  dx, [bx+di]         ; point to partition type name
                mov  ah, 9               ; print string service
```

```
                int  0x21               ; dos services

                push ss
                mov  ax, bp
                add  ax, si
                push ax                 ; pass partition entry address
                call printpart          ; print start and end from it

                cmp  byte [bp+si+4], 5   ; is it an extended partition
                je   recurse            ; yes, make a recursive call

                cmp  byte [bp+si+4], 0xf ; is it an extended partition
                jne  exit               ; yes, make a recursive call

recurse:        mov  ax, [bp+8]
                add  ax, 2              ; increase indentation level
                push ax
                push word [bp+si+0xA]    ; push partition type address
                push word [bp+si+0x8]
                call readpart           ; recursive call

exit:           add  si, 16             ; point to next partition entry
                cmp  si, -2             ; gone past last entry
                jne  nextpart           ; no, read this entry

failed:         pop  si
                pop  dx
                pop  bx
                pop  cx
                pop  ax
                mov  sp, bp
                pop  bp
                ret  6

start:          xor  ax, ax
                push ax                 ; start from zero indentation
                push ax                 ; main partition table at 0
                push ax
                call readpart           ; read and print it

                mov  ax, 0x4c00         ; terminate program
                int  0x21
```

6.3. STORAGE ACCESS USING DOS

BIOS provides raw access to the storage medium while DOS gives a more logical view and more cooked services. Everything is a file. A directory is a specially organized file that is interpreted by the operating system itself. A list of important DOS services for file manipulation is given below.

INT 21 - CREATE OR TRUNCATE FILE
```
AH = 3Ch
CX = file attributes
DS:DX -> ASCIZ filename
Return:
CF = error flag
AX = file handle or error code
```
INT 21 - OPEN EXISTING FILE
```
AH = 3Dh
AL = access and sharing modes
DS:DX -> ASCIZ filename
CL = attribute mask of files to look for (server call only)
Return:
CF = error flag
AX = file handle or error code
```
INT 21 - CLOSE FILE
```
AH = 3Eh
```

```
BX = file handle
Return:
CF = error flag
AX = error code
```

INT 21 - READ FROM FILE

```
AH = 3Fh
BX = file handle
CX = number of bytes to read
DS:DX -> buffer for data
Return:
CF = error flag
AX = number of bytes actually read or error code
```

INT 21 - WRITE TO FILE

```
AH = 40h
BX = file handle
CX = number of bytes to write
DS:DX -> data to write
Return:
CF = error flag
AX = number of bytes actually written or error code
```

INT 21 - DELETE FILE

```
AH = 41h
DS:DX -> ASCIZ filename (no wildcards, but see notes)
Return:
CF = error flag
AX = error code
```

INT 21 - SET CURRENT FILE POSITION

```
AH = 42h
AL = origin of move
BX = file handle
CX:DX = offset from origin of new file position
Return:
CF = error flag
DX:AX = new file position in bytes from start of file
AX = error code in case of error
```

INT 21 - GET FILE ATTRIBUTES

```
AX = 4300h
DS:DX -> ASCIZ filename
Return:
CF = error flag
CX = file attributes
AX = error code
```

INT 21 - SET FILE ATTRIBUTES

```
AX = 4301h
CX = new file attributes
DS:DX -> ASCIZ filename
Return:
CF = error flag
AX = error code
```

We will use some of these services to find that two files are same in contents or different. We will read the file names from the command prompt. The command string is passed to the program in the program segment prefix located at offset 0 in the current segment. The area from 0-7F contains information for DOS, while the command tail length is stored at 80. From 81 to FF, the actual command tail is stored terminated by a CR (Carriage Retrun).

Example 13.3

```
; file comparison using dos services
[org 0x0100]
              jmp  start

filename1:    times 128 db 0          ; space for first filename
filename2:    times 128 db 0          ; space for second filename
handle1:      dw   0                  ; handle for first file
handle2:      dw   0                  ; handle for second file
buffer1:      times 4096 db 0         ; buffer for first file
buffer2:      times 4096 db 0         ; buffer for second file

format:       db   'Usage error: diff <filename1> <filename2>$'
openfailed:   db   'First file could not be opened$'
openfailed2:  db   'Second file could not be opened$'
readfailed:   db   'First file could not be read$'
readfailed2:  db   'Second file could not be read$'
different:    db   'Files are different$'
same:         db   'Files are same$'

start:        mov  ch, 0
              mov  cl, [0x80]         ; command tail length in cx
              dec  cx                 ; leave the first space
              mov  di, 0x82           ; start of command tail in di
              mov  al, 0x20           ; space for parameter separation
              cld                     ; auto increment mode
              repne scasb             ; search space
              je   param2             ; if found, proceed
              mov  dx, format         ; else, select error message
              jmp  error              ; proceed to error printing

param2:       push cx                 ; save original cx
              mov  si, 0x82           ; set si to start of param
              mov  cx, di             ; set di to end of param
              sub  cx, 0x82           ; find param size in cx
              dec  cx                 ; excluding the space
              mov  di, filename1      ; set di to space for filename 1
              rep  movsb              ; copy filename there
              mov  byte [di], 0       ; terminate filename with 0
              pop  cx                 ; restore original cx
              inc  si                 ; go to start of next filename
              mov  di, filename2      ; set di to space for filename 2
              rep  movsb              ; copy filename there
              mov  byte [di], 0       ; terminate filename with 0

              mov  ah, 0x3d           ; service 3d – open file
              mov  al, 0              ; readonly mode
              mov  dx, filename1      ; address of filename
              int  0x21               ; dos services
              jnc  open2              ; if no error, proceed
              mov  dx, openfailed     ; else, select error message
              jmp  error              ; proceed to error printing

open2:        mov [handle1], ax       ; save handle for first file
              mov  ah, 0x3d           ; service 3d – open file
              mov  al, 0              ; readonly mode
              mov  dx, filename2      ; address of filename
              int  0x21               ; dos services
              jnc  store2             ; if no error, proceed
              mov  dx, openfailed2    ; else, select error message
              jmp  error              ; proceed to error printing

store2:       mov  [handle2], ax      ; save handle for second file

readloop:     mov  ah, 0x3f           ; service 3f – read file
              mov  bx, [handle1]      ; handle for file to read
              mov  cx, 4096           ; number of bytes to read
              mov  dx, buffer1        ; buffer to read in
              int  0x21               ; dos services
              jnc  read2              ; if no error, proceed
              mov  dx, readfailed     ; else, select error message
              jmp  error              ; proceed to error printing

read2:        push ax                 ; save number of bytes read
              mov  ah, 0x3f           ; service 3f – read file
```

```
                mov  bx, [handle2]      ; handle for file to read
                mov  cx, 4096           ; number of bytes to read
                mov  dx, buffer2        ; buffer to read in
                int  0x21               ; dos services
                jnc  check              ; if no error, proceed
                mov  dx, readfailed2    ; else, select error message
                jmp  error              ; proceed to error printing

check:          pop  cx                 ; number of bytes read of file 1
                cmp  ax, cx             ; are number of byte same
                je   check2             ; yes, proceed to compare them
                mov  dx, different      ; no, files are different
                jmp  error              ; proceed to message printing

check2:         test ax, ax             ; are zero bytes read
                jnz  compare            ; no, compare them
                mov  dx, same           ; yes, files are same
                jmp  error              ; proceed to message printing

compare:        mov  si, buffer1        ; point si to file 1 buffer
                mov  di, buffer2        ; point di to file 2 buffer
                repe cmpsb              ; compare the two buffers
                je   check3             ; if equal, proceed
                mov  dx, different      ; else, files are different
                jmp  error              ; proceed to message printing

check3:         cmp  ax, 4096           ; were 4096 bytes read
                je   readloop           ; yes, try to read more
                mov  dx, same           ; no, files are same

error:          mov  ah, 9              ; service 9 – output message
                int  0x21               ; dos services

                mov  ah, 0x3e           ; service 3e – close file
                mov  bx, [handle1]      ; handle of file to close
                int  0x21               ; dos services

                mov  ah, 0x3e           ; service 3e – close file
                mov  bx, [handle2]      ; handle of file to close
                int  0x21               ; dos services

                mov  ax, 0x4c00         ; terminate program
                int  0x21
```

Another interesting service that DOS provides regarding files is executing them. An important point to understand here is that whenever a program is executed in DOS all available memory is allocated to it. No memory is available to execute any new programs. Therefore memory must be freed using explicit calls to DOS for this purpose before a program is executed. Important services in this regard are listed below.

INT 21 - ALLOCATE MEMORY
AH = 48h
BX = number of paragraphs to allocate
Return:
CF = error flag
AX = segment of allocated block or error code in case of error
BX = size of largest available block in case of error
INT 21 - FREE MEMORY
AH = 49h
ES = segment of block to free
Return:
CF = error flag
AX = error code
INT 21 - RESIZE MEMORY BLOCK
AH = 4Ah
BX = new size in paragraphs
ES = segment of block to resize

```
Return:
CF = error flag
AX = error code
BX = maximum paragraphs available for specified memory block
```

```
INT 21 - LOAD AND/OR EXECUTE PROGRAM
AH = 4Bh
AL = type of load (0 = load and execute)
DS:DX -> ASCIZ program name (must include extension)
ES:BX -> parameter block
Return:
CF = error flag
AX = error code
```

The format of parameter block is as follows.

```
Offset  Size   Description
 00h    WORD   segment of environment to copy for child process
               (copy caller's environment if 0000h)
 02h    DWORD  pointer to command tail to be copied into child's PSP
 06h    DWORD  pointer to first FCB to be copied into child's PSP
 0Ah    DWORD  pointer to second FCB to be copied into child's PSP
 0Eh    DWORD  (AL=01h) will hold subprogram's initial SS:SP on return
 12h    DWORD  (AL=01h) will hold entry point (CS:IP) on return
```

As an example we will use the multitasking kernel client from the multitasking chapter and modify it such that after running all three threads it executes a new instance of the command prompt instead of indefinitely hanging around.

Example 13.4

```
; another multitasking TSR caller
[org 0x0100]
              jmp  start

; parameter block layout:
; cs,ip,ds,es,param
;  0, 2, 4, 6,  8

paramblock:   times 5 dw 0            ; space for parameters
lineno:       dw   0                  ; line number for next thread
chars:        db '\|/-'               ; chracters for rotating bar
message:      db 'moving hello'       ; moving string
message2:     db '            '       ; to erase previous string
messagelen:   dw 12                   ; length of above strings
tail:         db ' ',13
command:      db 'COMMAND.COM', 0
execblock:    times 11 dw 0

;;;;; COPY LINES 028-071 FROM EXAMPLE 10.1 (printnum) ;;;;;
;;;;; COPY LINES 073-114 FROM EXAMPLE 10.1 (printstr) ;;;;;
;;;;; COPY LINES 103-126 FROM EXAMPLE 11.5 (mytask) ;;;;;
;;;;; COPY LINES 128-146 FROM EXAMPLE 11.5 (mytask2) ;;;;;
;;;;; COPY LINES 148-193 FROM EXAMPLE 11.5 (mytask3) ;;;;;

start:        mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
              mov  [paramblock+6], es ; extra segment parameter
              mov  word [paramblock+8], 0 ; parameter for thread
              mov  si, paramblock     ; address of param block in si
              int  0x80               ; multitasking kernel interrupt

              mov  [paramblock+0], cs ; code segment parameter
              mov  word [paramblock+2], mytask2 ; offset parameter
              mov  [paramblock+4], ds ; data segment parameter
```

```
                mov  [paramblock+6], es ; extra segment parameter
                mov  word [paramblock+8], 0 ; parameter for thread
                mov  si, paramblock     ; address of param block in si
                int  0x80               ; multitasking kernel interrupt

                mov  [paramblock+0], cs ; code segment parameter
                mov  word [paramblock+2], mytask3 ; offset parameter
                mov  [paramblock+4], ds ; data segment parameter
                mov  [paramblock+6], es ; extra segment parameter
                mov  word [paramblock+8], 0 ; parameter for thread
                mov  si, paramblock     ; address of param block in si
                int  0x80               ; multitasking kernel interrupt

                mov  ah, 0x4a           ; service 4a – resize memory
                mov  bx, end            ; end of memory retained
                add  bx, 15             ; rounding up
                mov  cl, 4
                shr  bx, cl             ; converting into paras
                int  0x21               ; dos services

                mov  ah, 0x4b           ; service 4b - exec
                mov  al, 0              ; load and execute
                mov  dx, command        ; command to be executed
                mov  bx, execblock      ; address of execblock
                mov  word [bx+2], tail  ; offset of command tail
                mov  [bx+4], ds         ; segment of command tail
                int  0x21               ; dos services

                jmp $                   ; loop infinitely if returned
end:
```

6.4. DEVICE DRIVERS

Device drivers are operating system extensions that become part of the operating system and extend its services to new devices. Device drivers in DOS are very simple. They just have their services exposed through the file system interface.

Device driver file starts with a header containing a link to the next driver in the first four bytes followed by a device attribute word. The most important bit in the device attribute word is bit 15 which dictates if it is a character device or a block device. If the bit is zero the device is a character device and otherwise a block device. Next word in the header is the offset of a strategy routine, and then is the offset of the interrupt routine and then in one byte, the number of units supported is stored. This information is padded with seven zeroes.

Strategy routine is called whenever the device is needed and it is passed a request header. Request header stores the unit requested, the command code, space for return value and buffer pointers etc. Important command codes include 0 to initialize, 1 to check media, 2 to build a BIOS parameter block, 4 and 8 for read and write respectively. For every command the first 13 bytes of request header are same.

RH+0	BYTE	Length of request header
RH+1	BYTE	Unit requested
RH+2	BYTE	Command code
RH+3	BYTE	Driver’s return code
RH+5	9 BYTES	Reserved

The request header details for different commands is listed below.

0 – Driver Initialization

Passed to driver

RH+18 DWORD Pointer to character after equal sign on CONFIG.SYS line that loaded driver (read-only)

```
RH+22   BYTE    Drive number for first unit of this block driver
(0=A...)
Return from driver
RH+13   BYTE    Number of units (block devices only)
RH+14   DWORD   Address of first free memory above driver (break
address)
RH+18   DWORD   BPB pointer array (block devices only)
```

1 – Media Check

```
RH+13   BYTE    Media descriptor byte
Return
RH+14   BYTE    Media change code
                        -1 if disk changed
                        0 if dont know whether disk changed
                        1 if disk not changed
RH+15   DWORD   pointer to previous volume label if device attrib bit
11=1 (open/close/removable media supported)
```

2 – Build BPB

```
RH+13   BYTE    Media descriptor byte
RH+14   DWORD   buffer address (one sector)
Return
RH+18   DWORD   pointer to new BPB
if bit 13 (ibm format) is set buffer is first sector of fat, otherwise
scrach space
```

4 – Read / 8 – Write / 9 – Write with verify

```
RH+13   BYTE    Media descriptor byte
RH+14   DWORD   transfer address
RH+18   WORD    byte or sector count
RH+20   WORD    starting sector number (for block devices)
Return
RH+18   WORD    actual byte or sectors transferred
RH+22   DWORD   pointer to volume label if error 0Fh is returned
```

The BIOS parameter block discussed above is a structure that provides parameters about the storage medium. It is stored in the first sector or the boot sector of the device. Its contents are listed below.

```
00-01   bytes per sector
02      sectors per allocation unit
03-04   Number of reserved sectors ( 0 based)
05      number of file allocation tables
06-07   max number of root directory entries
08-09   total number of sectors in medium
0A      media descriptor byte
0B-0C   number of sectors occupied by a single FAT
0D-0E   sectors per track (3.0 or later)
0F-10   number of heads (3.0 or later)
11-12   number of hidden sectors (3.0 or later)
13-14   high-order word of number of hidden sectors (4.0)
15-18   IF bytes 8-9 are zero, total number of sectors in medium
19-1E   Reserved should be zero
```

We will be building an example device driver that takes some RAM and expresses it as a secondary storage device to the operating system. Therefore a new drive is added and that can be browsed to, filed copied to and from just like ordinary drives expect that this drive is very fast as it is located in the RAM. This program cannot be directly executed since it is not a user program. This must be loaded by adding the line “device=filename.sys” in the “config.sys” file in the root directory.

Example 13.5

```
; ram disk dos block device driver
header:     dd   -1                   ; no next driver
            dw   0x2000               ; driver attributes: block device
            dw   strategy             ; offset of strategy routine
            dw   interrupt            ; offset of interrupt routine
            db   1                    ; no of units supported
            times 7 db 0              ; reserved

request:    dd   0                    ; space for request header

ramdisk:    times 11 db 0             ; initial part of boot sector
bpb:        dw   512                  ; bytes per sector
            db   1                    ; sectors per cluster
            dw   1                    ; reserved sectors
            db   1                    ; fat copies
            dw   48                   ; root dir entries
            dw   105                  ; total sectors
            db   0xf8                 ; media desc byte: fixed disk
            dw   1                    ; sectors per fat
            times 482 db 0            ; remaining part of boot sector
            db   0xfe, 0xff, 0xff     ; special bytes at start of FAT
            times 509 db 0            ; remaining FAT entries unused
            times 103*512 db 0        ; 103 sectors for data
bpbptr:     dw   bpb                  ; array of bpb pointers

dispatch:   dw   init                 ; command 0: init
            dw   mediacheck           ; command 1: media check
            dw   getbpb               ; command 2: get bpb
            dw   unknown              ; command 3: not handled
            dw   input                ; command 4: input
            dw   unknown              ; command 5: not handled
            dw   unknown              ; command 6: not handled
            dw   unknown              ; command 7: not handled
            dw   output               ; command 8: output
            dw   output               ; command 9: output with verify

; device driver strategy routine
strategy:   mov  [cs:request], bx     ; save request header offset
            mov  [cs:request+2], es   ; save request header segment
            retf

; device driver interrupt routine
interrupt:  push ax
            push bx
            push cx
            push dx
            push si
            push di
            push ds
            push es

            push cs
            pop  ds

            les  di, [request]
            mov  word [es:di+3], 0x0100
            mov  bl, [es:di+2]
            mov  bh, 0
            cmp  bx, 9
            ja   skip
            shl  bx, 1

            call [dispatch+bx]

skip:       pop  es
            pop  ds
            pop  di
            pop  si
            pop  dx
            pop  cx
            pop  bx
            pop  ax
            retf

```

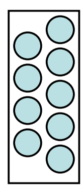
```
mediacheck: mov  byte [es:di+14], 1
            ret

getbpb:     mov  word [es:di+18], bpb
            mov  [es:di+20], ds
            ret

input:      mov  ax, 512
            mul  word [es:di+18]
            mov  cx, ax

            mov  ax, 512
            mul  word [es:di+20]
            mov  si, ax
            add  si, ramdisk

            les  di, [es:di+14]
            cld
            rep  movsb
            ret

output:     mov  ax, 512
            mul  word [es:di+18]
            mov  cx, ax

            lds  si, [es:di+14]
            mov  ax, 512
            mul  word [es:di+20]
            mov  di, ax
            add  di, ramdisk

            push cs
            pop  es
            cld
            rep  movsb
unknown:    ret

init:       mov  ah, 9
            mov  dx, message
            int  0x21

            mov  byte [es:di+13], 1
            mov  word [es:di+14], init
            mov  [es:di+16], ds
            mov  word [es:di+18], bpbptr
            mov  [es:di+20], ds
            ret

message:    db   13, 10, 'RAM Disk Driver loaded',13,10,'$'
```

7
Serial Port Programming

7.1. INTRODUCTION

Serial port is a way of communication among two devices just like the parallel port. The basic difference is that whole bytes are sent from one place to another in case of parallel port while the bits are sent one by one on the serial port in a specially formatted fashion. The serial port connection is a 9pin DB-9 connector with pins assigned as shown below.

6 – Data Set Ready (DSR)
7 – Request to Send (RTS)
8 – Clear to Send (CTS)
9 – Ring Indicator (RI)



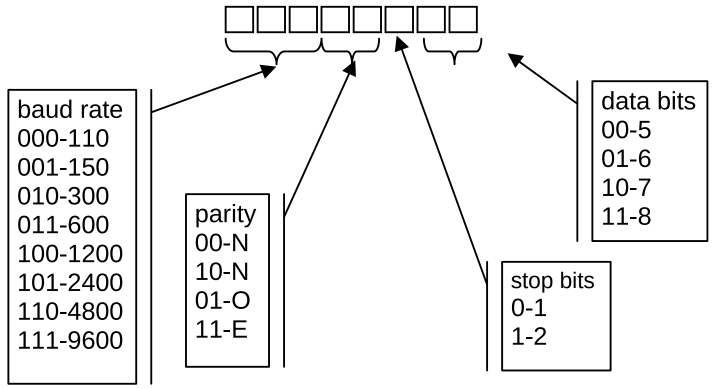
1 – Carrier Detect (CD)
2 – Received Data (RD)
3 – Transmitted Data (TD)
4 – Data Terminal Ready (DTR)
5 – Signal Ground

We have made a wire that connects signal ground of the two connectors, the TD of one to the RD of the other and the RD of one to the TD of the other. This three wire connection is sufficient for full duplex serial communication. The data on the serial port is sent in a standard format called RS232 communication. The data starts with a 1 bit called the start bit, then five to eight data bits, an optional parity bit, and one to two 0 bits called stop bits. The number of data bits, parity bits, and the number of stop bits have to be configured at both ends. Also the duration of a bit must be precisely known at both ends called the baud rate of the communication.

The BIOS INT 14 provides serial port services. We will use a mix of BIOS services and direct port access for our example. A major limitation in using BIOS is that it does not allows interrupt driven data transfer, i.e. we are interrupted whenever a byte is ready to be read or a byte can be transferred since the previous transmission has completed. To achieve this we have to resort to direct port access. Important BIOS services regarding the serial port are discussed below.

```
INT 14 - SERIAL - INITIALIZE PORT
AH = 00h
AL = port parameters
DX = port number (00h-03h)
Return:
AH = line status
AL = modem status
```

Every bit of line status conveys different information. From most significant to least significant, the meanings are timeout, transmitter shift register empty, transmitter holding register empty, break detect, receiver ready, overrun, parity error, and framing error. Modem status is not used in direct serial communication. The port parameters in AL consist of the baud rate, parity scheme, number of stop bits, and number of data bits. The description of various bits is as under.




```
INT 14 - SERIAL - WRITE CHARACTER TO PORT
AH = 01h
AL = character to write
DX = port number (00h-03h)
Return:
AH bit 7 = error flag
AH bits 6-0 = port status
INT 14 - SERIAL - READ CHARACTER FROM PORT
AH = 02h
DX = port number (00h-03h)
Return:
AH = line status
AL = received character if AH bit 7 clear
INT 14 - SERIAL - GET PORT STATUS
AH = 03h
DX = port number (00h-03h)
Return:
AH = line status
AL = modem status
```

Serial port is also accessible via I/O ports. COM1 is accessible via ports 3F8-3FF while COM2 is accessible via 2F8-2FF. The first register at 3F8 (or 2F8 for the other port) is the transmitter holding register if written to and the receiver buffer register if read from. Other registers of our interest include 3F9 whose bit 0 must be set to enable received data available interrupt and bit 1 must be set to enable transmitter holding register empty interrupt. Bit 0 of 3FA is set if an interrupt is pending and its bits 1-3 identify the cause of the interrupt. The three bit causes are as follows.

```
110    (16550, 82510) timeout interrupt pending
101    (82510) timer interrupt
100    (82510) transmit machine
011    receiver line status interrupt. priority=highest
010    received data available register interrupt. priority=second
001    transmitter holding register empty interrupt. priority=third
000    modem status interrupt. priority=fourth
```

The register at 3FB is line control register while the one at 3FD is line status register. The line status register has the same bits as returned in line status by the get port status BIOS interrupt however the most significant bit is reserved in this case instead of signaling a timeout. The register at 3FC is the modem control register. Bit 3 of this register must be set to enable interrupt generation by the serial port.

7.2. SERIAL COMMUNICATION

We give an example where two computers are connected using a serial cable made just as described above. The program is to be run on both computers. After that whatever is typed on one computer appears on the screen of the other.

Example 14.1

```
; a program using serial port to transfer data back and forth
[org 0x0100]
              jmp  start

screenpos:    dw   0              ; where to display next character

; subroutine to clear the screen
clrscr:       push es
              push ax
              push cx
              push di

              mov  ax, 0xb800
              mov  es, ax         ; point es to video base
              xor  di, di         ; point di to top left column
              mov  ax, 0x0720     ; space char in normal attribute
              mov  cx, 2000       ; number of screen locations

              cld                 ; auto increment mode
              rep  stosw          ; clear the whole screen

              pop  di
              pop  cx
              pop  ax
              pop  es
              ret

serial:       push ax
              push bx
              push dx
              push es

              mov  dx, 0x3FA      ; interrupt identification register
              in   al, dx         ; read register
              and  al, 0x0F       ; leave lowerniblle only
              cmp  al, 4          ; is receiver data available
              jne  skipall        ; no, leave interrupt handler

              mov  dx, 0x3F8      ; data register
              in   al, dx         ; read character

              mov  dx, 0xB800
              mov  es, dx         ; point es to video memory
              mov  bx, [cs:screenpos] ; get current screen position
              mov  [es:bx], al    ; write character on screen
              add  word [cs:screenpos], 2 ; update screen position
              cmp  word [cs:screenpos], 4000 ; is the screen full
              jne  skipall        ; no, leave interrupt handler

              call clrscr         ; clear the screen
              mov  word [cs:screenpos], 0 ; reset screen position

skipall:      mov  al, 0x20
              out  0x20, al       ; end of interrupt

              pop  es
              pop  dx
              pop  bx
              pop  ax
              iret
```

```

start:        call clrscr         ; clear the screen

              mov  ah, 0          ; initialize port service
              mov  al, 0xE3       ; line settings = 9600, 8, N, 1
              xor  dx, dx         ; port = COM1
              int  0x14           ; BIOS serial port services

              xor  ax, ax
              mov  es, ax         ; point es to IVT base
              mov  word [es:0x0C*4], serial
              mov  [es:0x0C*4+2], cs ; hook serial port interrupt

              mov  dx, 0x3FC      ; modem control register
              in   al, dx         ; read register
              or   al, 8          ; enable bit 3 (OUT2)
              out  dx, al         ; write back to register

              mov  dx, 0x3F9      ; interrupt enable register
              in   al, dx         ; read register
              or   al, 1          ; receiver data interrupt enable
              out  dx, al         ; write back to register

              in   al, 0x21       ; read interrupt mask register
              and  al, 0xEF       ; enable IRQ 4
              out  0x21, al       ; write back to register

main:         mov  ah, 0          ; read key service
              int  0x16           ; BIOS keybaord services
              push ax             ; save key for later use

retest:       mov  ah, 3          ; get line status
              xor  dx, dx         ; port = COM1
              int  0x14           ; BIOS keyboard services
              and  ah, 32         ; trasmitter holding register empty
              jz   retest         ; no, test again

              pop  ax             ; load saved key
              mov  dx, 0x3F8      ; data port
              out  dx, al         ; send on serial port

              jmp  main
```

8
Protected Mode Programming

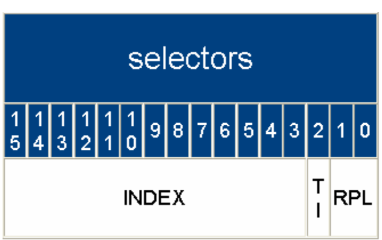
8.1. INTRODUCTION

Till now we have been discussing the 8088 architecture which was a 16bit processor. Newer processors of the Intel series provide 32bit architecture. Till now we were in real mode of a newer processor which is basically a compatibility mode making the newer processor just a faster version of the original 8088. Switching processor in the newer 32bit mode is a very easy task. Just turn on the least significant bit of a new register called CR0 (Control Register 0) and the processor switches into 32bit mode called protected mode. However manipulations in the protected mode are very different from those in the read mode.

All registers in 386 have been extended to 32bits. The new names are EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, and EFLAGS. The original names refer to the lower 16bits of these registers. A 32bit address register can access upto 4GB of memory so memory access has increased a lot.

As regards segment registers the scheme is not so simple. First of all we call them segment selectors instead of segment registers and they are still 16bits wide. We are also given two other segment selectors FS and GS for no specific purpose just like ES.

The working of segment registers as being multiplied by 10 and added into the offset for obtaining the physical address is totally changed. Now the selector is just an index into an array of segment descriptors where each descriptor describes the base, limit, and attributes of a segment. Role of selector is to select on descriptor from the table of descriptors and the role of descriptor is to define the actual base address. This decouples the selection and actual definition which is needed in certain protection mechanisms introduced into this processor. For example an operating system can define the possible descriptors for a program and the program is bound to select one of them and nothing else. This sentence also hints that the processor has some sense of programs that can or cannot do certain things like change this table of descriptors. This is called the privilege level of the program and varies for 0 (highest privilege) to 3 (lowest privilege). The format of a selector is shown below.



The table index (TI) is set to 0 to access the global table of descriptors called the GDT (Global Descriptor Table). It is set to 1 to access another table, the local descriptor table (LDT) that we will not be using. RPL is the requested privilege level that ranges from 0-3 and informs what privilege level the program wants when using this descriptor. The 13bit index is the actual index into the GDT to select the appropriate descriptor. 13 bits mean that a maximum of 8192 descriptors are possible in the GDT.

The GDT itself is an array of descriptors where each descriptor is an 8byte entry. The base and limit of GDT is stored in a 48bit register called the GDTR. This register is loaded with a special instruction LGDT and is given a memory address from where the 48bits are fetched. The first entry of the GDT must always be zero. It is called the null descriptor. After that any number of entries upto a maximum of 8191 can follow. The format of a code and data descriptor is shown below.

<table>
<tr><th colspan="33">code (application) segment descriptor</th></tr>
<tr><td>offset</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>+4</td><td colspan="8">BASE (bit31..24)</td><td>G</td><td>D</td><td>r.</td><td>AVL</td><td colspan="4">LIMIT (bit19..16)</td><td>P</td><td colspan="2">DPL</td><td>S = 1</td><td>X = 1</td><td>C</td><td>R</td><td>A</td><td colspan="8">BASE (bit23..16)</td></tr>
<tr><td>+0</td><td colspan="16">BASE (bit15..0)</td><td colspan="16">LIMIT (bit15..0)</td></tr>
</table>

<table>
<tr><th colspan="33">data (application) segment descriptor</th></tr>
<tr><td>offset</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>+4</td><td colspan="8">BASE (bit31..24)</td><td>G</td><td>B</td><td>r.</td><td>AVL</td><td colspan="4">LIMIT (bit19..16)</td><td>P</td><td colspan="2">DPL</td><td>S = 1</td><td>X = 0</td><td>E</td><td>W</td><td>A</td><td colspan="8">BASE (bit23..16)</td></tr>
<tr><td>+0</td><td colspan="16">BASE (bit15..0)</td><td colspan="16">LIMIT (bit15..0)</td></tr>
</table>

The 32bit base in both descriptors is scattered into different places because of compatibility reasons. The limit is stored in 20 bits but the G bit defines that the limit is in terms of bytes of 4K pages therefore a maximum of 4GB size is possible. The P bit must be set to signal that this segment is present in memory. DPL is the descriptor privilege level again related to the protection levels in 386. D bit defines that this segment is to execute code is 16bit mode or 32bit mode. C is conforming bit that we will not be using. R signals that the segment is readable. A bit is automatically set whenever the segment is accessed. The combination of S (system) and X (executable) tell that the descriptors is a code or a data descriptor. B (big) bit tells that if this data segment is used as stack SP is used or ESP is used.

Our first example is a very rudimentary one that just goes into protected mode and prints an A on the screen by directly accessing 000B8000.

Example 15.1

```
[org 0x0100]
              jmp  start

gdt:          dd   0x00000000, 0x00000000   ; null descriptor
              dd   0x0000FFFF, 0x00CF9A00   ; 32bit code
              ;      \--/\--/    \/||||\/
              ;       |   |      | ||||+--- Base (16..23)=0 fill later
              ;       |   |      | |||+--- X=1 C=0 R=1 A=0
              ;       |   |      | ||+--- P=1 DPL=00 S=1
              ;       |   |      | |+--- Limit (16..19) = F
              ;       |   |      | +--- G=1 D=1 r=0 AVL=0
```

```
              ;       |   |      +--- Base (24..31) = 0
              ;       |   +--- Limit (0..15) = FFFF
              ;       +--- Base (0..15)=0 fill later
              dd   0x0000FFFF, 0x00CF9200   ; data
              ;      \--/\--/    \/||||\/
              ;       |   |      | ||||+--- Base (16..23) = 0
              ;       |   |      | |||+--- X=0 E=0 W=1 A=0
              ;       |   |      | ||+--- P=1 DPL=00 S=1
              ;       |   |      | |+--- Limit (16..19) = F
              ;       |   |      | +--- G=1 B=1 r=0 AVL=0
              ;       |   |      +--- Base (24..31) = 0
              ;       |   +--- Limit (0..15) = FFFF
              ;       +--- Base (0..15) = 0

gdtreg:       dw   0x17      ; 16bit limit
              dd   0         ; 32bit base (filled later)

stack:        times 256 dd 0 ; for use in p-mode
stacktop:

start:        mov  ax, 0x2401
              int  0x15           ; enable A20

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              mov  [gdt+0x08+2], ax
              shr  eax, 16
              mov  [gdt+0x08+4], al         ; fill base of code desc

              xor  edx, edx
              mov  dx, cs
              shl  edx, 4
              add  edx, stacktop             ; edx = stack top for p-
mode

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              add  eax, gdt
              mov  [gdtreg+2], eax          ; fill phy base of gdt
              lgdt [gdtreg]                 ; load gdtr

              mov  eax, cr0
              or   eax, 1

              cli                           ; MUST disable interrupts
              mov  cr0, eax                 ; P-MODE ON
              jmp  0x08:pstart              ; load cs

;;;;; 32bit protected mode ;;;;;

[bits 32] ; ask assembler to generate 32bit code
pstart:       mov  eax, 0x10
              mov  ds, ax
              mov  es, ax                   ; load other seg regs
              mov  fs, ax                   ; flat memory model
              mov  gs, ax
              mov  ss, ax
              mov  esp, edx

              mov  byte [0x000b8000], 'A'   ; direct poke at video
              jmp  $                        ; hang around
```

Gate A20 is a workaround for a bug that is not detailed here. The BIOS call will simply enable it to open the whole memory for us. Another important thing is that the far jump we used loaded 8 into CS but CS is now a selector so it means Index=1, TI=0, and RPL=0 and therefore the actual descriptor loaded is the one at index 1 in the GDT.

8.2. 32BIT PROGRAMMING

Our next example is to give a falvour of 32bit programming. We have written the printstr function for read and for protected mode. The availability of larger registers and flexible addressing rules allows writing a much comprehensive version of the code. Also offsets to parameters and default widths change.

Example 15.2

```
[org 0x0100]
              jmp  start

gdt:          dd   0x00000000, 0x00000000   ; null descriptor
              dd   0x0000FFFF, 0x00CF9A00   ; 32bit code
              dd   0x0000FFFF, 0x00CF9200   ; data

gdtreg:       dw   0x17                     ; 16bit limit
              dd   0                        ; 32bit base

rstring:      db   'In Real Mode...', 0
pstring:      db   'In Protected Mode...', 0

stack:        times 256 dd 0                ; 1K stack
stacktop:

printstr:     push bp                       ; real mode print string
              mov  bp, sp
              push ax
              push cx
              push si
              push di
              push es

              mov  di,[bp+4]  ;load string address
              mov  cx,0xffff  ;load maximum possible size in cx
              xor  al,al      ;clear al reg
              repne scasb     ;repeat scan
              mov ax,0xffff   ;
              sub ax,cx       ;calculate length
              dec ax          ;off by one, as it includes zero
              mov cx,ax       ;move length to counter

              mov ax, 0xb800
              mov es, ax               ; point es to video base
              mov ax,80                ;its a word move, clears ah
              mul byte [bp+8]         ;its a byte mul to calc y offset
              add ax,[bp+10]           ;add x offset
              shl ax,1                 ;mul by 2 to get word offset
              mov di,ax                ;load pointer

              mov si, [bp+4]           ; string to be printed
              mov ah, [bp+6]           ; load attribute

              cld                      ; set auto increment mode
nextchar:     lodsb                    ;load next char and inc si by 1
              stosw                    ;store ax and inc di by 2
              loop nextchar

              pop es
              pop di
              pop si
              pop cx
              pop ax
              pop bp
              ret 8

start:          push byte 0                 ; 386 can directly push
immediates
              push byte 10
              push byte 7
              push word rstring
              call printstr

```

```
                mov  ax, 0x2401
                int  0x15                 ; enable a20

                xor  eax, eax
                mov  ax, cs
                shl  eax, 4
                mov  [gdt+0x08+2], ax
                shr  eax, 16
                mov  [gdt+0x08+4], al          ; set base of code desc

                xor  edx, edx
                mov  dx, cs
                shl  edx, 4
                add  edx, stacktop        ; stacktop to be used in p-mode

                xor  ebx, ebx
                mov  bx, cs
                shl  ebx, 4
                add  ebx, pstring         ; pstring to be used in p-mode

                xor  eax, eax
                mov  ax, cs
                shl  eax, 4
                add  eax, gdt
                mov  [gdtreg+2], eax           ; set base of gdt
                lgdt [gdtreg]                  ; load gdtr

                mov  eax, cr0
                or   eax, 1

                cli                            ; disable interrupts
                mov  cr0, eax                  ; enable protected mode
                jmp  0x08:pstart               ; load cs

;;;;; 32bit protected mode ;;;;;

[bits 32]
pprintstr:      push ebp                  ; p-mode print string routine
                mov  ebp, esp
                push eax
                push ecx
                push esi
                push edi

                mov  edi, [ebp+8]  ;load string address
                mov  ecx, 0xffffffff  ;load maximum possible size in cx
                xor  al, al      ;clear al reg
                repne scasb     ;repeat scan
                mov  eax, 0xffffffff   ;
                sub  eax, ecx       ;calculate length
                dec  eax          ;off by one, as it includes zero
                mov  ecx, eax       ;move length to counter

                mov  eax, 80               ;its a word move, clears ah
                mul   byte [ebp+16]          ;its a byte mul to calc y
offset
                add  eax, [ebp+20]          ;add x offset
                shl  eax, 1                ;mul by 2 to get word offset
                add  eax, 0xb8000
                mov  edi, eax               ;load pointer

                mov esi, [ebp+8]          ; string to be printed
                mov ah, [ebp+12]          ; load attribute

                cld                       ; set auto increment mode
pnextchar:      lodsb                     ;load next char and inc si by 1
                stosw                     ;store ax and inc di by 2
                loop pnextchar

                pop  edi
                pop  esi
                pop  ecx
                pop  eax
                pop  ebp
                ret  16                   ; 4 args now mean 16 bytes

```

```
pstart:       mov  ax, 0x10           ; load all seg regs to 0x10
              mov  ds, ax             ; flat memory model
              mov  es, ax
              mov  fs, ax
              mov  gs, ax
              mov  ss, ax
              mov  esp, edx           ; load saved esp on stack

              push byte 0
              push byte 11
              push byte 7
              push ebx
              call pprintstr              ; call p-mode print string
routine

              mov  eax, 0x000b8000
              mov  ebx, '/-\|'

nextsymbol:   mov  [eax], bl
              mov  ecx, 0x00FFFFFF
              loop $
              ror  ebx, 8
              jmp  nextsymbol
```

8.3. VESA LINEAR FRAME BUFFER

As an example of accessing a really large area of memory for which protected mode is a necessity, we will be accessing the video memory in high resolution and high color graphics mode where the necessary video memory is alone above a megabyte. We will be using the VESA VBE 2.0 for a standard for these high resolution modes.

VESA is the Video Electronics Standards Association and VBE is the set of Video BIOS Extensions proposed by them. The VESA VBE 2.0 standard includes a linear frame buffer mode that we will be using. This mode allows direct access to the whole video memory. Some important VESA services are listed below.

INT 10 – VESA – Get SuperVGA Infromation
AX = 4F00h
ES:DI -> buffer for SuperVGA information
Return:
AL = 4Fh if function supported
AH = status
INT 10 – VESA – Get SuperVGA Mode Information
AX = 4F01h
CX = SuperVGA video mode
ES:DI -> 256-byte buffer for mode information
Return:
AL = 4Fh if function supported
AH = status
ES:DI filled if no error
INT 10 – VESA – Set VESA Video Mode
AX = 4F02h
BX = new video mode
Return:
AL = 4Fh if function supported
AH = status

One of the VESA defined modes is 8117 which is a 1024x768 mode with 16bit color and a linear frame buffer. The 16 color bits for every pixel are organized in 5:6:5 format with 5 bits for red, 6 for green, and 5 for blue. This makes 32 shades of red and blue and 64 shades of green and 64K total possible colors. The 32bit linear frame buffer base address is available at offset 28 in the mode information buffer. Our example will produces shades of green on the screen and clear them and again print them in an infinite loop with delays inbetween.

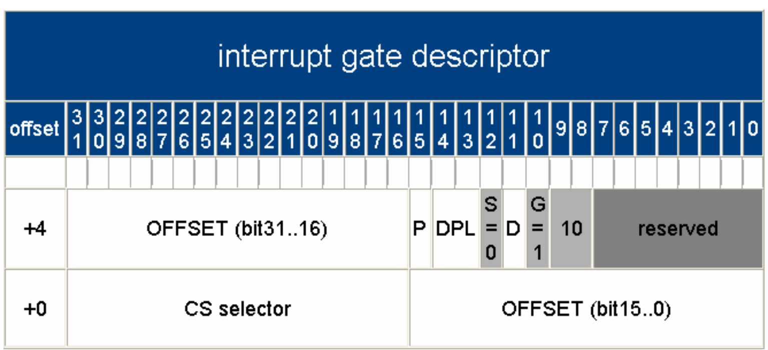
Example 15.3

```
[org 0x0100]
              jmp  start

modeblock:    times 256 db 0

gdt:          dd   0x00000000, 0x00000000   ; null descriptor
              dd   0x0000FFFF, 0x00CF9A00   ; 32bit code
              dd   0x0000FFFF, 0x00CF9200   ; data

gdtreg:       dw   0x17                     ; 16bit limit
              dd   0                        ; 32bit base

stack:        times 256 dd 0                ; 1K stack
stacktop:

start:        mov  ax, 0x4f01          ; get vesa mode information
              mov   cx,  0x8117            ;  1024*768*64K  linear  frame
buffer
              mov  di, modeblock
              int  0x10
              mov  esi, [modeblock+0x28]    ; save frame buffer base

              mov  ax, 0x4f02          ; set vesa mode
              mov  bx, 0x8117
              int  0x10

              mov  ax, 0x2401
              int  0x15                ; enable a20

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              mov  [gdt+0x08+2], ax
              shr  eax, 16
              mov  [gdt+0x08+4], al         ; set base of code desc

              xor  edx, edx
              mov  dx, cs
              shl  edx, 4
              add  edx, stacktop       ; stacktop to be used in p-mode

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              add  eax, gdt
              mov  [gdtreg+2], eax          ; set base of gdt
              lgdt [gdtreg]                 ; load gdtr

              mov  eax, cr0
              or   eax, 1

              cli                           ; disable interrupts
              mov  cr0, eax                 ; enable protected mode
              jmp  0x08:pstart              ; load cs

;;;;; 32bit protected mode ;;;;;

[bits 32]
pstart:       mov  ax, 0x10            ; load all seg regs to 0x10
              mov  ds, ax              ; flat memory model
              mov  es, ax
              mov  fs, ax
              mov  gs, ax
              mov  ss, ax
              mov  esp, edx            ; load saved esp on stack

l1:           xor  eax, eax
```

```
                mov  edi, esi
                mov  ecx, 1024*768*2/4        ; divide by 4 as dwords
                cld
                rep  stosd

                mov  eax, 0x07FF07FF
                mov  ecx, 32                  ; no of bands
                mov  edi, esi

l2:             push ecx
                mov  ecx, 768*16              ; band width = 32 lines
                cld
                rep stosd

                mov  ecx, 0x000FFFFF          ; small wait
                loop $
                pop  ecx

                sub  eax, 0x00410041
                loop l2

                mov  ecx, 0x0FFFFFFF          ; long wait
                loop $
                jmp  l1
```

8.4. INTERRUPT HANDLING

Handling interrupts in protected mode is also different. Instead of the IVT at physical address 0 there is the IDT (interrupt descriptor table) located at physical address stored in IDTR, a special purpose register. The IDTR is also a 48bit register similar in structure to the GDTR and loaded with another special instruction LGDT. The format of the interrupt descriptor is as shown below.



The P and DPL have the same meaning as in data and code descriptors. The S bit tells that this is a system descriptor while the 1110 following it tells that it is a 386 interrupt gate. Our example hooks the keyboard and timer interrupts and displays certain things on the screen to show that they are working.

Example 15.4

```
[org 0x0100]
                jmp  start

gdt:            dd   0x00000000, 0x00000000   ; null descriptor
                dd   0x0000FFFF, 0x00CF9A00   ; 32bit code
                dd   0x0000FFFF, 0x00CF9200   ; data

gdtreg:         dw   0x17                     ; 16bit limit
                dd   0                        ; 32bit base
```

```

idt:          times 8 dw unhandled, 0x0008, 0x8e00, 0x0000
              dw   timer, 0x0008, 0x8e00, 0x0000
                   \---/  \----/    ||\/  \----/
                     |      |       || |    +----- offset bits 16..32
                     |      |       || +----- reserved
                     |      |       |+------ Type=E 386 Interrupt Gate
                     |      |       +--- P=1 DPL=00 S=0
                     |      +------- selector
                     +-------- offset bits 0..15
              dw   keyboard, 0x0008, 0x8e00, 0x0000
              times 246 dw unhandled, 0x0008, 0x8e00, 0x0000

idtreg:       dw   0x07FF
              dd   0

stack:        times 256 dd 0                ; 1K stack
stacktop:

start:        mov  ax, 0x2401
              int  0x15                ; enable a20

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              mov  [gdt+0x08+2], ax
              shr  eax, 16
              mov  [gdt+0x08+4], al         ; set base of code desc

              xor  edx, edx
              mov  dx, cs
              shl  edx, 4
              add  edx, stacktop       ; stacktop to be used in p-mode

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              add  eax, gdt
              mov  [gdtreg+2], eax          ; set base of gdt
              lgdt [gdtreg]                 ; load gdtr

              xor  eax, eax
              mov  ax, cs
              shl  eax, 4
              add  eax, idt
              mov  [idtreg+2], eax          ; set base of idt

              cli                           ; disable interrupts
              lidt [idtreg]                 ; load idtr

              mov  eax, cr0
              or   eax, 1
              mov  cr0, eax                 ; enable protected mode

              jmp  0x08:pstart              ; load cs

;;;;; 32bit protected mode ;;;;;

[bits 32]
unhandled:    iret

timer:        push eax

              inc  byte [0x000b8000]

              mov  al, 0x20
              out  0x20, al
              pop  eax
              iret

keyboard:     push eax

              in   al, 0x60
              mov  ah, al
              and  al, 0x0F
              shr  ah, 4
```

```
                add  ax, 0x3030
                cmp  al, 0x39
                jbe  skip1
                add  al, 7
skip1:          cmp  ah, 0x39
                jbe  skip2
                add  ah, 7
skip2:          mov  [0x000b809C], ah
                mov  [0x000b809E], al

skipkb:         mov  al, 0x20
                out  0x20, al
                pop  eax
                iret

pstart:         mov  ax, 0x10            ; load all seg regs to 0x10
                mov  ds, ax              ; flat memory model
                mov  es, ax
                mov  fs, ax
                mov  gs, ax
                mov  ss, ax
                mov  esp, edx            ; load saved esp on stack

                mov  al, 0xFC
                out  0x21, al            ; no unexpected int comes

                sti                      ; interrupts are okay now

                jmp  $
```

EXERCISES

1. Write very brief and to-the-point answers.
    a. Why loading idtr with a value appropriate for real mode is necessary while gdtr is not?
    b. What should we do in protected mode so that when we turn protection off, we are in unreal mode?
    c. If the line jmp code:next is replaced with call code:somefun, the prefetch queue is still emptied. What problem will occur when somefun will return?
    d. How much is ESP decremented when an interrupt arrives. This depends on weather we are in 16-bit mode or 32-bit. Does it depend on any other thing as well? If yes, what?
    e. Give two instructions that change the TR register.
2. Name the following descriptors like code descriptor, data descriptor, interrupt gate etc.

```
gdt:    dd 0x00000000, 0x00000000
        dd 0x00000000, 0x00000000
        dd 0x80000fA0, 0x0000820b
        dd 0x0000ffff, 0x00409a00
        dd 0x80000000, 0x0001d20b
```

3. Using the above GDT, which of the following values, when moved into DS will cause an exception and why.

```
0x00
0x08
0x10
0x18
0x28
0x23
```

4. Using the above GDT, if DS contains 0x20, which of the following offsets will cause an exception on read access?

```
0x0ffff
0x10000
0x10001
```

5. The following function is written in 32-bit code for a 16-bit stack. Against every instruction, write the prefixes generated before that instruction. Prefixes can be address size, operand size, repeat, or segment override. Then rewrite the code such that no prefixes are generated considering that this is assembled and executed in 32-bit mode. Don't care for retaining register values. The function copies specified number of DWORDs between two segments.

```
[bits 32]
memcpy:      mov  bp, sp
             lds  esi, [bp+4]        ; source address
             les  edi, [bp+10]       ; destination address
             mov  cx, [bp+16]        ; count of DWORDs to move
             shl  cx, 1              ; make into count of WORDs
L1:          mov  dx, [si]
             mov  [es:di], dx
             dec  cx
             jnz  L1
             ret
```

6. Rewrite the following scheduler so that it schedules processes stored in readyQ, where enque and deque functions are redefined and readyQ contains TSS selectors of processes to be multitasked. Remember you can't use a register as a segment in a jump (eg jmp ax:0) but you can jump to an indirect address (eg jmp far [eax]) where eax points to a six-byte address. Declare any variables you need.

```
             mov al, 0x20
scheduler:   jmp USERONESEL:0
             out  0x20, al
             mov  byte [USERONEDESC+5], 0x89
             jmp  USERTWOSEL:0
             out  0x20, al
             mov  byte [USERTWODESC+5], 0x89
             jmp  scheduler
```

7. Protected mode has specialized mechanism for multitasking using task state segments but the method used in real mode i.e. saving all registers in a PCB, selecting the next PCB and loading all registers from there is still applicable. Multitask two tasks in protected mode multitasking without TSS. Assume that all processes are at level zero so no protection issues arise. Be careful to save the complete state of the process.
8. Write the following descriptors.
    a. 32 bit, conforming, execute-only code segment at level 2, with base at 6MB and a size of 4MB.
    b. 16 bit, non-conforming, readable code segment at level 0, with base at 1MB and a size of 10 bytes.
    c. Read only data segment at level 3, with base at 0 and size of 1MB.
    d. Interrupt Gate with selector 180h and offset 11223344h.
9. Write physical addresses for the following accesses where CS points to the first descriptor above, DS to the second, ES to the third, EBX contains 00010000h, and ESI contains 00020000h
    a. [bx+si]
    b. [ebx+esi-2ffffh]
    c. [es:ebx-10h]
10. Which of the following will cause exceptions and why. The registers have the same values as the last question.
    a. mov eax, [cs:10000h]
    b. mov [es:esi:100h], ebx
    c. mov ax, [es:ebx]
11. Give short answers.
    a. How can a GPF (General protection fault) occur while running the following code

```
push es
pop  es
```

   b. How can a GPF occur during the following instruction? Give any two reasons.

```
jmp  10h:100h
```

   c. What will happen if we call interrupt 80h after loading out IDT and before switching to protected mode?
   d. What will happen if we call interrupt 80h after switching into protected mode but before making a far jump?
12. Write the following descriptors. Assume values for attributes not specifically mentioned.
   a. Write able 32-bit data segment with 1 GB base and 1 GB limit and a privilege level of 2.
   b. Readable 16-bit code descriptor with 1 MB base and 1 MB limit and a privilege level of 1.
   c. Interrupt gate given that the handler is at 48h:12345678h and a privilege level of 0.
13. Describe the following descriptors. Give their type and the value of all their fields.

```
dd 01234567h, 789abcdeh
dd 30405060h, 70809010h
dd 00aabb00h, 00ffee00h
```

14. Make an EXE file, switch into protected mode, rotate an asterisk on the border of the screen, and return to real mode when the border is traversed.

9
Interfacing with High Level Languages

9.1. CALLING CONVENTIONS

To interface an assembly routine with a high level language program means to be able to call functions back and forth. And to be able to do so requires knowledge of certain behavior of the HLL when calling functions. This behavior of calling functions is called the calling conventions of the language. Two prevalent calling conventions are the C calling convention and the Pascal calling convention.

What is the naming convention

C prepends an underscore to every function or variable name while Pascal translates the name to all uppercase. C++ has a weird name mangling scheme that is compiler dependent. To avoid it C++ can be forced to use C style naming with extern “C” directive.

How are parameters passed to the routine

In C parameters are pushed in reverse order with the rightmost being pushed first. While in Pascal they are pushed in proper order with the leftmost being pushed first.

Which registers must be preserved

Both standards preserve EBX, ESI, EDI, EBP, ESP, DS, ES, and SS.

Which registers are used as scratch

Both standards do not preserve or gurantee the value of EAX, ECX, EDX, FS, GS, EFLAGS, and any other registers.

Which register holds the return value

Both C and Pascal return upto 32bit large values in EAX and upto 64bit large values in EDX:EAX.

Who is responsible for removing the parameters

In C the caller removes the parameter while in Pascal the callee removes them. The C scheme has reasons pertaining to its provision for variable number of arguments.

9.2. CALLING C FROM ASSEMBLY

For example we take a function divide declared in C as follows.

```
int divide( int dividend, int divisor );
```

To call this function from assembly we have to write.

```
push dword [mydivisor]
```

```
push dword [mydividend]
call _divide
add esp, 8
; EAX holds the answer
```

Observe the order of parameters according to the C calling conventions and observe that the caller cleared the stack. Now take another example of a function written in C as follows.

```
void swap( int* p1, int* p2 )
{
        int temp = *p1;
        *p1 = *p2;
        *p2 = temp;
}
```

To call it from assembly we have to write this.

```
[section .text]
extern _swap
x:              dd 4
y:              dd 7

push dword y
push dword x
call _swap      ; will only retain the specified registers
add esp, 8
```

Observe how pointers were initialized appropriately. The above function swap was converted into assembly by the gcc compiler as follows.

```
; swap generated by gcc with no optimizations (converted to Intel
syntax)
; 15 instructions AND 13 memory accesses
_swap:
        push ebp
        mov     ebp, esp
        sub     esp, 4                          ; space created for temp

        mov eax, [ebp+8]
        mov eax, [eax]
        mov     [ebp-4], eax                    ; temp = *p1

        mov edx, [ebp+8]
        mov eax, [ebp+12]
        mov eax, [eax]
        mov     [edx], eax                      ; *p1 = *p2

        mov edx, [ebp+12]
        mov eax, [ebp-4]
        mov [edx], eax                  ; *p2 = temp

        leave   ;;;;; EQUIVALENT TO mov esp, ebp AND pop ebp ;;;;;
        ret
```

If we turn on optimizations the same function is compiled into the following code.

```
; generated with full optimization by gcc compiler
; 12 instructions AND 11 memory accesses
_swap:
        push    ebp
        mov     ebp, esp
        push    ebx

        mov     edx, [ebp+8]
        mov     ecx, [ebp+12]
        mov     ebx, [edx]
        mov     eax, [ecx]
        mov     [edx], eax
        mov     [ecx], ebx

        pop     ebx
        pop     ebp
        ret
```

9.3. CALLING ASSEMBLY FROM C

We now write a hand optimized version in assembly. Our version is only 6 instructions and 6 memory accesses.

Example 16.1

```
[section .text]
global        _swap
_swap:        mov  ecx,[esp+4]       ; copy parameter p1 to ecx
              mov  edx,[esp+8]       ; copy parameter p2 to edx
              mov  eax,[ecx]  ; copy *p1 into eax
              xchg eax,[edx]  ; exchange eax with *p2
              mov  [ecx],eax  ; copy eax into *p1
              ret                    ; return from this function
```

We assemble the above program with the following command.

- nasm –f win32 swap.asm

This produces a swap.obj file. The format directive told the assembler that it is to be linked with a 32bit Windows executable. The linking process involves resolving imported symbols of one object files with export symbols of another. In NASM an imported symbol is declared with the extern directive while and exported symbol is declared with the global directive.

We write the following program in C to call this assembly routine. We should have provided the swap.obj file to the C linker otherwise an unresolved external symbol error will come.

Example 16.1

```
#include <stdio.h>

void swap( int* p1, int* p2 );

int main()
{
  int a = 10, b = 20;
  printf( "a=%d b=%d\n", a, b );
  swap(&a, &b );
  printf( "a=%d b=%d\n", a, b );
  system( "PAUSE" );
  return 0;
}
```

EXERCISES

1. Write a traverse function in assembly, which takes an array, the number of elements in the array and the address of another function to be called for each member of the array. Call the function from a C program.
2. Make the linked list functions make in Exercise 5.XX available to C programs using the following declarations.

```
struct node {
        int data;
        struct node* next;
};
void init( void );
struct node* createlist( void );
void insertafter( struct node*, int );
void deleteafter( struct node* );
void deletelist( struct node* );
```

3. Add two functions to the above program implemented in C. The function “printnode” should print the data in the passed node using printf, while “countfree” should count the number of free nodes by traversing the free list starting from the node address stored in firstfree.

```
void printnode( struct node* );
void countfree( void );
```

4. Add the function “printlist” to the above program and implement in assembly. This function should traverse the list whose head is passed as parameter and for each node containing data (head is dummy and doesn’t contain data) calls the C function printnode to actually print the contained data.

```
void printlist( struct node* );
```

5. Modify the createlist and deletelist functions in the above program to increment and decrement an integer variable “listcount” declared in C to maintain a count of linked lists present.

10
Comparison with Other Processors

We emphasized that assembly language has to be learned once and every processor can be programmed by that person. To give a flavour of two different widely popular processors we introduce the Motorolla 68K series and the Sun SPARC processors. The Motorolla 68K processors are very popular in high performance embedded applications while the Sun SPARC processors are popular in very high end enterprise servers. We will compare them with the Intel x86 series which is known for its success in the desktop market.

10.1. MOTOROLLA 68K PROCESSORS

Motorolla 68K processors are very similar to Intel x86 series in their architecture and instruction set. The both are of the same era and added various features at the same time. The instructions are very similar however the difference in architecture evident from a programmer's point of view must be understood.

68K processors have 16 23bit general purpose registers named from A0-A7 and D0-D7. A0-A7 can hold addresses in indirect memory accesses. These can also be used as software stack pointers. Stack in 68K is not as rigit a structure as it is in x86. There is a 32bit program counter (PC) that holds the address of currently executing instruction. The 8bit condition code register (CCR) holds the X (Extend) N (Negative) Z (Zero) V (Overflow) C (Carry) flags. X is set to C for extended operations (addition, subtraction, or shifting).

Motrolla processors allow bit addressing, that is a specific bit in a byte or a bit field, i.e. a number of bits can be directly accessed. This is a very useful feature especially in control applications. Other data types include byte, word, long word, and quad word. A special MOVE16 instruction also accepts a 16byte block.

68K allows indirect memory access using any A register. A special memory access allows postincrement or predecrement as part of memory access. These forms are written as (An), (An)+, and –(An). Other forms allow addressing with another regiser as index and with constant displacement. Using one of the A registers as the stack pointer and using the post increment and pre decrement forms of addressing, stack is implemented. Immediates can also be given as arguments and are preceded with a hash sign (#). Addressing is indicated with parenthesis instead of brackets.

68K has no segmentation; it however has a paged memory model. It used the big endian format in contrast to the little endian used by the Intel processors. It has varying instruction lengths from 1-11 words. It has a decrementing stack just like the Intel one. The format of instructions is "operation source, destination" which is different from the Intel order of operatnds. Some instructions from various instruction groups are given below.

Data Movement

```
EXG D0, D2
MOVE.B (A1), (A2)
```

```
MOVEA (2222).L, A4
MOVEQ #12, D7
```
Arithmetic
```
ADD D7, (A4)
CLR (A3)              (set to zero)
CMP (A2), D1
ASL, ASR, LSL, LSR, ROR, ROL, ROXL, ROXR (shift operations)
```
Program Control
```
BRA label
JMP (A3)
BSR label             (CALL)
JSR (A2)              (indirect call)
RTD #4                (RET N)
```
Conditional Branch
```
BCC                   (branch if carry clear)
BLS                   (branch if Lower or Same)
BLT                   (branch if Less Than)
BEQ                   (branch if Equal)
BVC                   (branch if Overflow clear)
```

10.2. SUN SPARC PROCESSOR

The Sun SPARC is a very popular processing belonging to the RISC (reduced instruction set computer) family of processors. RISC processors originally named because of the very few rudimentary instructions they provided, are now providing almost as many instruction as CISC (complex instruction set computer). However some properties like a fixed instruction size and single clock execution for most instructions are there.

SPARC stands for Scalable Processor ARChitecture. SPARC is a 64bit processor. It byte order is user settable and even on a per program basis. So one program may be using little endian byte order and another may be using big endian at the same time. Data types include byte, Halfword, Word (32bit), and Double Word (64bits) and Quadword. It has a fixed 32bit instruction size. It has a concept of ASI (Address Space Identifier); an 8bit number that works similar to a segment.

There are 8 global registers and 8 alternate global registers. One of them is active at a time and accessible as g0-g7. Apart from that it has 8 in registers (i0-i7), 8 local registers (l0-l7), and 8 out registers (o0-o7). All registers are 64bit in size. The global registers can also be called r0-r7, in registers as r8-r15, local registers as r16-r23, and out registers as r24-r31.

SPARC introduces a concept of register window. One window is 24 registers and the active window is pointed to by a special register called Current Window Pointer (CWP). The actual number of registers in the processor is in hundreds not restricted by the architecture definition. Two instruction SAVE and RESTORE move this register window forward and backward by 16 registers. Therefore one SAVE instruction makes the out register the in registers and brings in new local and out registers. A RESTORE instruction makes the in registers out registers and restores the old local and in registers. This way parameters passing and returning can be totally done in registers and there is no need to save and restore registers inside subroutines.

The register o6 is conventionally used as the stack pointer. Return address is stored in o7 by the CALL instruction. The register g0 (r0) is always 0 so loading 0 in a register is made easy. SPARC is a totally register based architecture, or it is called a load-store architecture where memory access is only allowed in data movement instruction. Rest of the operations must be done on registers.

SPARC instructions have two sources and a distinct destination. This allows more flexibility in writing programs. Some examples of instructions of this processor follow.

Data Movement

LDSB [rn], rn	(load signed byte)
LDUW [rn], rn	(load unsigned word)
STH [rn], rn	(store half word)

Arithmetic

source1 = rn	
source2 = rn or simm13	
dest = rn	
ADD r2, r3, r4	
SUB r2, 4000, r5	
SLL, SRA, SRL	(shifting)
AND, OR, XOR	(logical)

Program Control

CALL	(direct call)
JMPL	(register indirect)
RET	
SAVE	
RESTORE	
BA label	(Branch Always)
BE label	(branch if equal)
BCC label	(branch if carry clear)
BLE label	(branch if less or equal)
BVS label	(branch if overflow set)