System Programming CS609
<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>68</td>
</tr>
<tr>
<td>10</td>
<td>76</td>
</tr>
<tr>
<td>11</td>
<td>83</td>
</tr>
<tr>
<td>12</td>
<td>95</td>
</tr>
<tr>
<td>13</td>
<td>103</td>
</tr>
<tr>
<td>14</td>
<td>110</td>
</tr>
<tr>
<td>15</td>
<td>117</td>
</tr>
<tr>
<td>16</td>
<td>125</td>
</tr>
<tr>
<td>17</td>
<td>133</td>
</tr>
<tr>
<td>18</td>
<td>146</td>
</tr>
<tr>
<td>19</td>
<td>155</td>
</tr>
<tr>
<td>20</td>
<td>163</td>
</tr>
<tr>
<td>21</td>
<td>172</td>
</tr>
<tr>
<td>22</td>
<td>180</td>
</tr>
<tr>
<td>23</td>
<td>186</td>
</tr>
<tr>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>25</td>
<td>199</td>
</tr>
<tr>
<td>26</td>
<td>207</td>
</tr>
<tr>
<td>27</td>
<td>216</td>
</tr>
<tr>
<td>28</td>
<td>223</td>
</tr>
<tr>
<td>29</td>
<td>229</td>
</tr>
<tr>
<td>30</td>
<td>236</td>
</tr>
<tr>
<td>31</td>
<td>244</td>
</tr>
<tr>
<td>32</td>
<td>249</td>
</tr>
<tr>
<td>33</td>
<td>256</td>
</tr>
<tr>
<td>34</td>
<td>262</td>
</tr>
<tr>
<td>35</td>
<td>267</td>
</tr>
<tr>
<td>36</td>
<td>274</td>
</tr>
<tr>
<td>37</td>
<td>283</td>
</tr>
<tr>
<td>38</td>
<td>291</td>
</tr>
<tr>
<td>39</td>
<td>301</td>
</tr>
<tr>
<td>40</td>
<td>306</td>
</tr>
<tr>
<td>41</td>
<td>312</td>
</tr>
<tr>
<td>42</td>
<td>317</td>
</tr>
<tr>
<td>43</td>
<td>324</td>
</tr>
<tr>
<td>44</td>
<td>329</td>
</tr>
<tr>
<td>45</td>
<td>332</td>
</tr>
</tbody>
</table>
Lecture # 1

What is Systems Programming?

Computer programming can be categorized into two categories i.e.

While designing software the programmer may determine the required inputs for that program, the wanted outputs and the processing the software would perform in order to give those wanted outputs. The implementation of the processing part is associated with application programming. Application programming facilitates the implementation of the required processing that software is supposed to perform; everything that is left now is facilitated by system programming.

Systems programming is the study of techniques that facilitate the acquisition of data from input devices, these techniques also facilitates the output of data which may be the result of processing performed by an application.

Three Layered Approach

A system programmer may use a three layered approach for systems programming. As you can see in the figure the user may directly access the programmable hardware in order to perform I/O operations. The user may use the trivial BIOS (Basic Input Output System) routines in order to perform I/O in which case the programmer need not know the internal working of the hardware and need only the knowledge BIOS routines and their parameters.

In this case the BIOS programs the hardware for required I/O operation which is hidden to the user. In the third case the programmer may invoke operating systems (DOS or whatever) routines in order to perform I/O operations. The operating system in turn will use BIOS routines or may program the hardware directly in order to perform the operation.
Methods of I/O

In the three layered approach if we are following the first approach we need to program the hardware. The hardware can be programmed to perform I/O in three ways i.e.

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access

In case of programmed I/O the CPU continuously checks the I/O device if the I/O operation can be performed or not. If the I/O operations can be performed the CPU performs the computations required to complete the I/O operation and then again starts waiting for the I/O device to be able to perform next I/O operation. In this way the CPU remains tied up and is not doing anything else besides waiting for the I/O device to be idle and performing computations only for the slower I/O device.

In case of interrupt driven the flaws of programmed driven I/O are rectified. The processor does not check the I/O device for the capability of performing I/O operation rather the I/O device informs the CPU that it's idle and it can perform I/O operation, as a result the execution of CPU is interrupted and an Interrupt Service Routine (ISR) is invoked which performs the computations required for I/O operation. After the execution of ISR the CPU continues with whatever it was doing before the interruption for I/O operation. In this way the CPU does not remain tied up and can perform computations for other processes while the I/O devices are busy performing I/O and hence is more optimal.

Usually it takes two bus cycles to transfer data from some I/O port to memory or vice versa if this is done via some processor register. This transfer time can be reduced bypassing the CPU as ports and memory device are also interconnected by system bus. This is done with the support of DMA controller. The DMA (direct memory access) controller can controller the buses and hence the CPU can be bypassed data item can be transferred from memory to ports or vice versa in a single bus cycle.
I/O controllers

I/O device

I/O controller

CPU

No I/O device is directly connected to the CPU. To provide control signals to the I/O device a I/O controller is required. I/O controller is located between the CPU and the I/O device. For example the monitor is not directly collected to the CPU rather the monitor is connected to a VGA card and this VGA card is in turn connected to the CPU through busses. The keyboard is not directly connected to CPU rather its connected to a keyboard controller and the keyboard controller is connected to the CPU. The function of this I/O controller is to provide

- I/O control signals
- Buffering
- Error Correction and Detection

We shall discuss various such I/O controllers interfaced with CPU and also the techniques and rules by which they can be programmed to perform the required I/O operation.

Some of such controllers are

- DMA controller
- Interrupt controller
- Programmable Peripheral Interface (PPI)
- Interval Timer
- Universal Asynchronous Receiver Transmitter

We shall discuss all of them in detail and how they can be used to perform I/O operations.

Operating systems

Systems programming is not just the study of programmable hardware devices. To develop effective system software one needs to the internals of the operating system as well. Operating systems make use of some data structures or tables for management of computer resources. We will take up different functions of the operating systems and discuss how they are performed and how can the data structures used for these operations be accessed.
**File Management**

File management is an important function of the operating systems. DOS/Windows uses various data structures for this purpose. We will see how it performs I/O management and how the data structures used for this purpose can be directly accessed. The various data structures are popularly known as FAT which can be of 12, 16 and 32 bit wide, Other data structures include BPB(BIOS parameter block), DPB( drive parameter block) and the FCBs(file control block) which collectively forms the directory structure. To understand the file structure the basic requirement is the understanding of the disk architecture, the disk formatting process and how this process divides the disk into sectors and clusters.

**Memory management**

Memory management is another important aspect of operating systems. Standard PC operate in two mode in terms of memory which are
- **Real Mode**
- **Protected Mode**

In real mode the processor can access only first one MB of memory to control the memory within this range the DOS operating system makes use of some data structures called
- **FCB (File control block)**
- **PSP (Program segment prefix)**

We shall discuss how these data structures can be directly accessed, what is the significance of data in these data structures. This information can be used to traverse through the memory occupied by the processes and also calculate the total amount of free memory available.

Certain operating systems operate in protected mode. In protected mode all of the memory interfaced with the processor can be accessed. Operating systems in this mode make use of various data structures for memory management which are
- **Local Descriptor Table**
- **Global Descriptor Table**
- **Interrupt Descriptor Table**

We will discuss the significance these data structures and the information stored in them. Also we will see how the logical addresses can be translated into physical addresses using the information these tables

**Viruses and Vaccines**

Once an understanding of the file system and the memory Management is developed it is possible to understand the working of viruses. Virus is a simple program which can embed itself within the computer resources and propagate itself. Mostly viruses when activated would perform something hazardous.
We will see where do they embed themselves and how can they be detected. Moreover we will discuss techniques of how they can be removed and mostly importantly prevented to perform any infections. There are various types of viruses but we will discuss those which embed themselves within the program or executable code which are Executable file viruses Partition Table or boot sector viruses

**Device Drivers**

Just connecting a device to the PC will not make it work unless its device drivers are not installed. This is so important because a device driver contains the routines which perform I/O operations on the device. Unless these routines are provided no I/O operation on the I/O device can be performed by any application. We will discuss the integrated environment for the development of device drivers for DOS and Windows.

We shall begin our discussion from means of I/O. On a well designed device it is possible to perform I/O operations from three different methods

- Programmed I/O
- Interrupt driven I/O
- DMA driven I/O

In case of programmed I/O the CPU is in a constant loop checking for an I/O opportunity and when its available it performs the computations operations required for the I/O operations. As the I/O devices are generally slower than the CPU, CPU has to wait for I/O operation to complete so that next data item can be sent to the device. The CPU sends data on the data lines. The device need to be signaled that the data has been sent this is done with the help of STROBE signal. An electrical pulse is sent to the device by turning this signal to 0 and then 1. The device on getting the strobe signal receives the data and starts its output. While the device is performing the output it’s busy and cannot accept any further data on the other and CPU is a lot faster device and can process lot more bytes during the output of previously sent data so it should be synchronized with the slower I/O device. This is usually done by another feed back signal of BUSY which is kept active as long as the device is busy. So the CPU is only waiting for the
device to get idle by checking the BUSY signal as long as the device is busy and when the device gets idle the CPU will compute the next data item and send it to the device for I/O operation. Similar is the case of input, the CPU has to check the DR (data Ready) signal to see if data is available for input and when its not CPU is busy waiting for it.

**Interrupt Driven I/O**

The main disadvantage of programmed I/O as can be noticed is that the CPU is busy waiting for an I/O opportunity and as a result remain tied up for that I/O operation. This disadvantage can be overcome by means of interrupt driven I/O. In Programmed I/O CPU itself checks for an I/O opportunity but in case of interrupt driven I/O the I/O controller interrupts the execution of CPU when ever and I/O operation is required for the computation of the required I/O operation. This way the CPU can perform other computation and interrupted to perform and interrupt service routine only when an I/O operation is required, which is quite an optimal technique.
**DMA driven I/O**

In case data is needed to transferred from main memory to I/O port this can be done using CPU which will consume 2 bus cycles for a single word, one bus cycle from memory to CPU and other from CPU to I/O port in case of output and the vice versa in case of input. In case no computation on data is required CPU can be bypassed and another device DMA (direct memory access) controller can be used. Its possible to transfer a data word directly from memory to CPU and vice versa in a single bus cycle using the DMA, this technique is definitely faster.

We shall start our discussion with the study of interrupt and the techniques used to program them. We will discuss other methods of I/O as required.

**What are interrupts?**

Literal to interrupt means to break the continuity of some on going task. When we talk of computer interrupt we mean exactly the same in terms of the processor. When an interrupt occurs the continuity of the processor is broken and the execution branches to an interrupt service routine. This interrupt service routine is a set of instruction carried out by the CPU to perform or initiate an I/O operation generally. When the routine is over the execution of the CPU returns to the point of interruption and continues with the on going process.
Interrupts can be of two types
- Hardware interrupts
- Software interrupts

Only difference between them is the method by which they are invoked. Software interrupts are invoked by means of some software instruction or statement and hardware interrupt is invoked by means of some hardware controller generally.

**Interrupt Mechanism**

Interrupts are quite similar to procedures or function because it is also another form temporary execution transfer, but there some differences as well. Note that when procedures are invoked by there names which represents their addresses is specified whereas in case of interrupts their number is specified. This number can be any 8 bit value which certainly is not its address. So the first question is what is the significance of this number? Another thing should also be noticed that procedures are part of the program but the interrupts invoked in the program are no where declared in the program. So the next question is where do these interrupts reside in memory and if they reside in memory then what would be the address of the interrupt?

Firstly lets see where do interrupts reside. Interrupts certainly reside somewhere in memory, the interrupts supported by the operating system resides in kernel which you already know is the core part of the operating system. In case of DOS the kernel is io.sys which loads in memory at boot time and in case of windows the kernel is kernel32.dll or kernel.dll. these files contain most of the I/O routines and are loaded as required. The interrupts supported by the ROM BIOS are loaded in ROM part of the main memory which usually starts at the address F000:0000H. Moreover it is possible that some device drivers have been installed these device drivers may provide some I/O routines so when the system boots these I/O routines get memory resident at interrupt service routines. So these are the three possibilities.

Secondly a program at compile time does not know the exact address where the interrupt service routine will be residing in memory so the loader cannot assign addresses for interrupt invocations. When a device driver loads in memory it places the address of the services provided by itself in the interrupt vector table. Interrupt Vector Table (IVT) in short is a 1024 bytes sized table which can hold 256 far addresses as each far address occupies 4 bytes. So its possible to store the addresses of 256 interrupts hence there are a maximum of 256 interrupt in a standard PC. The interrupt number is used as an index into the table to get the address of the interrupt service routine.
Lecture # 2

Interrupt Mechanism
Interrupt follow a follow a certain mechanism for their invocation just like near or far procedures. To understand this mechanism we need to understand its differences with procedure calls.

Difference between interrupt and procedure calls
Procedures or functions of sub-routines in various different languages are called by different methods as can be seen in the examples.

- Call MyProc
- A= Addition(4,5);
- Printf(“hello world”);

The general concept for procedure call in most of the programming languages is that on invocation of the procedure the parameter list and the return address (which is the value if IP register in case of near or the value of CS and IP registers in case of far procedure) is pushed. Moreover in various programming languages whenever a procedure is called its address need to be specified by some notation i.e. in C language the name of the procedure is specified to call a procedure which effectively can be used as its address.

However in case of interrupts the a number is used to specify the interrupt number in the call

- Int 21h
- Int 10h
- Int3

Moreover when an interrupt is invoked three registers are pushed as the return address i.e. the values of IP, CS and Flags in the described order which are restored on return. Also
no parameters are pushed onto the stack on invocation parameters can only be passed through registers.

The interrupt vector table
The interrupt number specified in the interrupt call is used as an index into the interrupt vector table. Interrupt vector table is a global table situated at the address 0000:0000H. The size of interrupt vector table is 1024 bytes or 1 KB. Each entry in the IVT is sized 4 bytes hence 256 interrupt vectors are possible numbered (0-FFH). Each entry in the table contains a far address of an interrupt handlers hence there is a maximum of 256 handlers however each handlers can have a number of services within itself. So the number operations that can be performed by calling an interrupt service routine (ISR) is indefinite depending upon the nature of the operating system. Each vector contains a far address of an interrupt handler. The address of the vector and not the address of interrupt handler can be easily calculated if the interrupt number is known. The segment address of the whole IVT is 0000H the offset address for a particular interrupt handler can be determined by multiplying its number with 4 eg. The offset address of the vector of INT 21H will be 21H * 4 = 84H and the segment for all vectors is 0 hence its far address is 0000:0084H,( this is the far address of the interrupt vector and not the interrupt service routine or interrupt handler). The vector in turn contains the address of the interrupt service routine which is an arbitrary value depending upon the location of the ISR residing in memory.

Fig 2 (Interrupt Vector Table)

<table>
<thead>
<tr>
<th>Interrupt Vector Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT0</td>
</tr>
<tr>
<td>INT1</td>
</tr>
<tr>
<td>INTFF</td>
</tr>
</tbody>
</table>

Moreover it is important to understand the meaning of the four bytes within the interrupt vector. Each entry within the IVT contain a far address the first two bytes (lower word) of which is the offset and the next two bytes (higher word) is the segment address.
Location of ISRs (Interrupt service routines)
Generally there are three kind of ISR within a system depending upon the entity which implements it
- BIOS (Basic I/O services) ISRs
- DOS ISRs
- ISRs provided by third party device drivers
When the system has booted up and the applications can be run all these kind of ISRs maybe provided by the system. Those provided by the ROM-BIOS would be typically resident at any location after the address F000:0000H because this the address within memory from where the ROM-BIOS starts, the ISRs provided by DOS would be resident in the DOS kernel (mainly IO.SYS and MSDOS.SYS loaded in memory) and the ISR provided by third party device drivers will be resident in the memory occupied by the device drivers.
where IO.SYS and other device drivers have been loaded but the location of ROM BIOS is not shown by this command.

```
C:\>mem /d
Address     Name          Size       Type
-------     --------     ------     ------
000000                   000400     Interrupt Vector
000400                   000100     ROM Communication Area
000500                   000200     DOS Communication Area

000700      IO           000370     System Data
  CON       System Device Driver
  AUX       System Device Driver
  PRN       System Device Driver
  CLOCK$                System Device Driver
  COM1      System Device Driver
  LPT1      System Device Driver
  LPT2      System Device Driver
  LPT3      System Device Driver
  COM2      System Device Driver
  COM3      System Device Driver
  COM4      System Device Driver

000A70      MSDOS        001610     System Data

002080      IO           002030     System Data
  KBD       000C00     System Program
  HIMEM     00040E     DEVICE=
  XMSXXXXX0              Installed Device Driver
                     000490     FILES=
                     000090     FCRS=
                     000120     LASTDRIVE=
                     0007D0     STACKS=

0040C0      COMMAND      000A20     Program
004AF0      MSDOS        000070     -- Free --
004B70      COMMAND      000600     Environment
005250      DOSX        0087A0     Program
00DA00      MEM         000610     Environment
025510      MSDOS        07AAD0     -- Free --
09F000      SYSTEM       02F000     System Program

655360 bytes total conventional memory
655360 bytes available to MS-DOS
597952 largest executable program size

1048576 bytes total contiguous extended memory
0 bytes available contiguous extended memory
941056 bytes available XMS memory
MS-DOS resident in High Memory Area
```

**Interrupt Invocation**

Although hardware and software interrupts are invoked differently i.e hardware interrupts are invoked by means of some hardware whereas software interrupts are invoked by means of software instruction or statement but no matter how an interrupt has been invoked processor follows a certain set steps after invocation of interrupts in exactly same way in both the cases. These steps are listed as below:

- Push Flags, CS, IP Registers, Clear Interrupt Flag
- Use (INT#)*4 as Offset and Zero as Segment
• This is the address of interrupt Vector and not the ISR
• Use lower two bytes of interrupt Vector as offset and move into IP
• Use the higher two bytes of Vector as Segment Address and move it into CS=0:[offset+2]
• Branch to ISR and Perform I/O Operation
• Return to Point of Interruption by Popping the 6 bytes i.e. Flags CS, IP.

This can be analyzed practically by the use of debug program, used to debug assembly language code, by assembling and debugging INT instructions

C:\>debug
-d 0:84
0000:0080  7C 10 A7 00-4F 03 55 05 8A 03 55 05 |....O.U....U.
0000:0090  17 03 55 05 86 10 A7 00-90 10 A7 00 9A 10 A7 00 .U............
0000:00A0  B8 10 A7 00 54 02 70 00-F2 04 74 CC B8 10 A7 00 ....T.p......
0000:00B0  B8 10 A7 00 B8 10 A7 00-40 01 21 04 50 09 AB D4 ........@.!.P...
0000:00C0  EA AE 10 A7 00 B8 00 FO-B8 10 A7 00 C4 23 02 C9 00 ...........
0000:00D0  B8 10 A7 00 B8 10 A7 00 10 A7 00 B8 10 A7 00 00A7H...........
0000:00E0  B8 10 A7 00 B8 10 A7 00-10 A7 00 B8 10 A7 00 00A7H...........
0000:00F0  B8 10 A7 00 B8 10 A7 00-10 A7 00 B8 10 A7 00 00A7H...........
0000:0100  8A 04 10 02 ....
-a
0AF1:0100 int 21
0AF1:0102 -t
AX=0000 BX=0000 CX=0000 DX=0000 SP=FEEE BP=0000 SI=0000 DI=0000
DS=0AF1 ES=0AF1 SS=0AF1 CS=0AF1 IP=0100 NV UP EI PL NZ NA PO NC
0AF1:0100 CD21 INT 21
-t
AX=0000 BX=0000 CX=0000 DX=0000 SP=FEE8 BP=0000 SI=0000 DI=0000
DS=0AF1 ES=0AF1 SS=0AF1 CS=00A7 IP=107C NV UP DI PL NZ NA PO NC
00A7:107C 90 NOP
-d ss:ffe8
0AF1:FEE0 02 01 F1 0A 02 F2 00 00
0AF1:FF00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

The dump at the address 0000:0084 H shows the value of the vector of the interrupt #21H i.e. 21H * 4 = 84H. This address holds the value 107CH in lower word and 00A7H in the higher word which indicates that the segment address of interrupt #21 is 00A7H and the offset address of this ISR is 107CH.

Moreover the instruction INT 21H can be assembled and executed in the debug program, on doing exactly so the instruction is traced through and the result is monitored. It can be seen that on execution of this instruction the value of IP is changed to 107CH and the value of CS is changed to 00A7H which cause the execution to branch to the Interrupt #21H in memory and the previous values of flags, CS and IP registers are temporarily saved onto the stack as the value of SP is reduced by 6 and the dump at the location SS:SP will show these saved values as well.

Parameter passing into Software interrupts

In case of procedures or function in various programming languages parameters are passed through stack. Interrupts are also kind of function provided by the operating system but they do not accept parameters by stack rather they need to passed parameters through registers.

Software interrupts invocation

Now let’s see how various interrupts can be invoked by means of software statements. First there should be way to pass parameters into a software interrupt before invoking the
interrupt; there are several methods for doing this. One of the methods is the use of pseudo variables. A variable can be defined a space within the memory whose value can be changed during the execution of a program but a pseudo variable acts very much like a variable as its value can be changed anywhere in the program but is not a true variable as it is not stored in memory. C programming language provides the use of pseudo variables to access various registers within the processor.

The are various registers like AX, BX, CX and DX within the processor they can be directly accessed in a program by using their respective pseudo variable by just attaching a "_" (underscore) before the register’s name eg. \_AX = 5; \_A = \_BX.

After passing the appropriate parameters the interrupt can be directly invoked by calling the \_geninterrupt() function. The interrupt number needs to be passed as parameter into the \_geninterrupt() function.

**Interrupt # 21H, Service # 09 description**

Now lets learn by means of an example how this can be accomplished. Before invoking the interrupt the programmer needs to know how the interrupt behaves and what parameters it requires. Lets take the example of interrupt # 21H and service # 09 written as 21H/09H in short. It is used to print a string ending by a ‘$’ character and other parameters describing the string are as below

**Inputs**

\ AH = 0x09  
\ DS = Segment Address of string  
\ DX = Offset Address of string  

**Output**

The ‘$’ terminated string at the address DS:DX is displayed

One thing is note worthy that the service # is placed in AH which is common with almost all the interrupts and its service. Also this service is not returning any significant data, if some service needs to return some data it too is received in registers depending upon the particular interrupt.

**Example:**

```c
#include<stdio.h>  
#include<BIOS.H>  
#include<DOS.H>  
#include<conio.h>

char st[80] ={"Hello World$"};

void main()
{
    clrscr(); //to clear the screen contents
    _DX = (unsigned int) st;
    _AH = 0x09;
    \_geninterrupt(0x21);
    getch(); //waits for the user to press any key
}
```

this is a simple example in which the parameters of int 21H/09H are loaded and then int 21H is invoked. DX and AH registers are accessed through pseudo variables and then
geninterrupt() is called to invoke the ISR. Also note that _DS is not loaded. This is the case as the string to be loaded is of global scope and the C language compiler automatically loads the segment address of the global data into the DS register.

Another Method for invoking software interrupts
This method makes use of a Union. This union is formed by two structure which correspond to general purpose registers AX, BX, CX and DX. And also the half register AH, AL, BH, BL, CH, CL, DH, DL. These structures are combined such that through this structure the field ax can be accessed to load a value and also its half components al and ah can be accessed individually. The declaration of this structure goes as below. If this union is to be used a programmer need not declare the following declaration rather declaration already available through its header file “dos.h”

```
struct full
{
    unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
};
struct half
{
    unsigned char al;
    unsigned char ah;
    unsigned char bl;
    unsigned char bh;
    unsigned char cl;
    unsigned char ch;
    unsigned char dl;
    unsigned char dh;
};
typedef union tagREGS
{
    struct full x;
    struct half h;
}REGS;
```

This union can be used to signify any of the full or half general purpose register shows if the field ax in x struct is to be accessed then accessing the fields al and ah in h will also have the same effect as show in the example below.

**Example:**

```
#include<DOS.H>
union REGS regs;
void main (void )
{
    regs.h.al = 0x55;
    regs.h.ah = 0x99;
    printf (“%x”,regs.x.ax);
}
The `int86()` function
The significance of this REGS union can only be understood after understanding the `int86()` function. The `int86()` has three parameters. The first parameter is the interrupt number to be invoked, the second parameter is the reference to a REGS type union which contains the value of parameters that should be passed as inputs, and third parameter is a reference to a REGS union which will contain the value of registers returned by this function. All the required parameters for an ISR are placed in REGS type of union and its reference is passed to an `int86()` function. This function will put the value in this union into the respective register and then invoke the interrupt. As the ISR returns it might leave some meaningful value in the register (ISR will return values), these values can be retrieved from the REGS union whose reference was passed into the function as the third parameter.

Example using interrupt # 21H service # 42H
To make it more meaningful we can again elaborate it by means of an example. Here we make use of ISR 21H/42H which is used to move the file pointer. Its detail is as follows

<table>
<thead>
<tr>
<th>Int # 21</th>
<th>Service # 42H</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>AL = Move Technique</td>
<td></td>
</tr>
<tr>
<td>BX = File Handle</td>
<td></td>
</tr>
<tr>
<td>CX-DX = No of Bytes File to be moved</td>
<td></td>
</tr>
<tr>
<td>AH = Service # = 42H</td>
<td></td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>DX-AX = No of Bytes File pointer actually moved.</td>
<td></td>
</tr>
</tbody>
</table>

This service is used to move the file pointer to a certain position relative to a certain point. The value in AL specify the point relative to which the pointer is moved. If the value of AL = 0 then file pointer is moved relative to the BOF (begin of File) if AL=1 then its moved relative to current position and if AL = 2 then its moved relative to the EOF (end of file).

CX-DX specify the number of bytes to move a double word is needed to specify this value as the size of file in DOS can be up to 2 GB.

On return of the service DX-AX will contain the number of bytes the file pointer is actually moved eg. If the file pointer is moved relative to the EOF zero bytes the DX-AX on return will contain the size of file if the file pointer was at BOF before calling the service.
Lecture # 3

The above described service can be used to get the size of a file in the described manner. The following C language program tries to accomplish just that. This program has been saved as .C file and not as .CPP file and then compiled.

Example 21H/42H:
#include<stdio.h>
#include<fcntl.h>
#include<io.h>
#include<BIOS.H>
#include<DOS.H>

unsigned int handle;
void main()
{
    union REGS regs;
    unsigned long int size;
    handle = open("c:\abc.txt",O_RDONLY);
    regs.x.bx = handle;
    regs.h.ah = 0x42;
    regs.h.al = 0x02; //correction
    regs.x.cx = 0;
    regs.x.dx = 0;
    int86(0x21,&regs,&regs);
    *((int*)(&size)) = regs.x.ax;
    *((int*)(&size))+1 =regs.x.dx;
    printf ("Size is %d" ,size);
}

This program opens a file and saves its handle in the handle variable. This handle is passed to the ISR 21H/42H along with the move technique whose value is 2 signifying movement relative to the EOF and the number of bytes to move are specified to be zero indicating that the pointer should move to the EOF. As the file was just opened the previous location of the file pointer will be BOF. On return of this service DX-AX will contain the size of the file. The low word of this size in ax is placed in the low word of size variable and the high word in dx is placed in the high word of size variable.

Another Example:
Lets now illustrate how ISR can be invoked by means of another example of BIOS service. Here we are choosing the ISR 10h/01h. This interrupt is used to perform I/O on the monitor. Moreover this service is used to change the size of cursor in text mode. The description of this service is given as under.
Int # 10H            Service # 01H

Entry
AH = 01
CH = Beginning Scan Line
CL = Ending Scan Line

On Exit
Unchanged

The size of the cursor depends upon the number of net scan lines used to display the
cursor if the beginning scan line is greater than the ending scan line the cursor will
disappear. The following tries to accomplish just that

```c
void main()
{
    char st[80];
    union REGS regs;
    regs.h.ah = 0x01;
    regs.h.ch = 0x01;
    regs.h.cl = 0x00;
    int86(0x10,&regs,&regs); //corrected
    gets(st);
}
```

The program is quite self explanatory as it puts the starting scan line to be 1 and the
ending scan line to be 0. Henceforth when the service execute the cursor will disappear.

Use of ISRs for C Library functions

There are various library function that a programmer would typically use in a program to
perform input output operations. These library functions perform trivial I/O operations
like character input (`putch()`) and character output (`getch()`, `getc()` etc). All
these function call various ISRs to perform this I/O. In BIOS and DOS documentation
number of services can be found that lie in correspondence with some C library function
in terms of its functionality.

Writing S/W ISRs

Lets now see how can a programmer write an ISR routine and what needs to be done in
order make the service work properly. To exhibit this we will make use of an interrupt
which is not used by DOS or BIOS so that our experiment does not put any interference
to the normal functions of DOS and BIOS. One such interrupt is interrupt # 65H. The
vector of int 65H is typically filled with zeros another indication that it is not being used.

Getting interrupt vector

As we have discussed earlier IVT is a table containing 4 byte entries each of which is a
far address of an interrupt service routine. All the vectors are arranged serially such that
the interrupt number can be used as an index into the IVT.

Getting interrupt vector refers to the operation which used to reading the far address
stored within the vector. The vector is double word, the lower word of it being the offset
address and the higher word being the segment address. Moreover the address read from a
vector can be used as a function pointer. The C library function used to do the exactly
same is `getvect(int#)` which requires the interrupt number a parameter and returns the value of its vector.

**Fig 1 (Vector being read from IVT )**

<table>
<thead>
<tr>
<th>Segment</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Function pointers**

Another thing required to be understood are the function pointers. C language is a very flexible language just like there are pointers for integers, characters and other data types there are pointers for functions as well as illustrated by the following example

```c
void myfunc()
{

}

void (*funcptr) ( )

funcptr = myfunc;
(*funcptr) ( );
myfunc();
```

There are three fragments of code in this example. The first fragment shows the declaration of a function `myfunc()`.

The second fragment show declaration of a pointer to function named `funcptr` which is a pointer to a function that returns void.

In the third fragment `funcptr` is assigned the address of `myfunc` as the name of the function can be used as its address just like in the cases of arrays in C. Then the function pointed by `funcptr` by the statement `(*funcptr)();` is called and then the original `myfunc()` is called. The user will observe in both the cases same function `myproc()` will be invoked.
Interrupt pointers and functions
Interrupt functions are special functions that as compared to simple functions for reasons discussed earlier. It can be declared using the keyword interrupt as shown in the following examples.

```c
void interrupt newint ( )
{
    ...
    ...
}
```

Similarly a pointer to such interrupt type function can also be declared as following

```c
void interrupt (*intptr) ( );
```

where intptr is the interrupt pointer and it can be assigned an address using the getvect() function

```c
intptr = getvect(0x08);
```

Now interrupt number 8 can be invoked using the interrupt vector as following

```c
(*intptr) ( );
```

Setting Interrupt Vector
Setting interrupt vector is just the reverse process of getting interrupt vector. To set the interrupt vector means is to change the double word sized interrupt vector within the IVT. This task can be accomplished using the function setvect(int #, newint) which requires the number of interrupt whose vector is to be changed and the new value of the vector.

```
INT #
   Offset
   Offset
   Segment
   Segment
```
In the following example a certain interrupt type function has been declared. The address of this function can be placed on to the vector of any interrupt using `setvect()` function as following. The following code places the address of `newint` function at the vector of int 8

```c
void interrupt newint ( )
{
    ...
    ...
}
setvect(0x08, newint);
```

C program making use of Int 65H

Here is a listing of a program that makes use of int 65H to exhibit how software interrupts needs to be programmed.

```c
void interrupt (*oldint65)( );
char st[80] = {"Hello World$"};
void interrupt newint65(void);
void main()
{
    oldint65 = getvect(0x65);
    setvect(0x65, newint65);
    geninterrupt (0x65);
    geninterrupt (0x65);
    geninterrupt (0x65);
    setvect(0x65, oldint65);
}
void interrupt newint65( )
{
    _AH = 0x09;
    _DX=(unsigned int)st;
    geninterrupt (0x21);
}
```

The above listing saves the address of original int 65H in the pointer `oldint65`. It then places the address of its own function `newint65` at the vector of interrupt number 65H. From this point onwards whenever int 65H is invokes the function `newint65` will be invoked. Int 65 is invoked thrice which will force the `newint65` function to be invoked thrice accordingly. After this the original value of the vector stored in `oldint65` is restored. The `newint65` function only displays the string `st`. As the interrupt 65 is invoked thrice this string will be printed thrice.

The Keep function

One deficiency in the above listing is that it is not good enough for other application i.e. after the termination of this program the `newint65` function is de-allocated from the memory and the interrupt vector needs to be restored otherwise it will act as a dangling
pointer (pointing to a place where there is garbage or where there is no meaningful function). To make the effect of this program permanent the newint65 function need to be memory resident. This can be achieved by the function keep() which is quite similar to exit() function. The exit() function returns the execution to the parent shell program and de-allocates the memory allocated to the program whereas the keep() function also returns the execution to the parent program but the memory allocated to the process may still remain allocated.

keep (return code, no. of paras);

the keep() function requires the return code which is usually zero for normal termination and the number of paragraphs required to be allocated. Each paragraph is 16 bytes in size.

TSR Programs
Following is a listing of a TSR (Terminate and Stay Resident) program which programs the interrupt number 65H but in this case the new interrupt 65H function remains in memory even after the termination of the program and hence the vector of int 65h does not become a dangling pointer.

```
#include<BIOS.H>
#include<DOS.H>

char st[80] ="Hello World$";
void interrupt (*oldint65)( );
void interrupt newint65( );
void main()
{
    oldint65 = getvect(0x65);
    setvect(0x65, newint65);
    keep(0, 1000);
}
void interrupt newint65( )
{
    _AH = 0x09;
    _DX=(unsigned int)st;
    geninterrupt (0x21);
}
```

The main() function gets and sets the vector of int 65H such that the address of newint65 is placed at its vector. In this case the program is made memory resident using the keep function and 1000 paragraphs of memory is reserved for the program (the amount of paragraphs is just a calculated guess work based upon the size of application). Now if any application as in the following case invokes int 65H the string st which is also now memory resident will be displayed.
#include<BIOS.H>
#include<DOS.H>

void main()
{
    geninterrupt (0x65);
    geninterrupt (0x65);
}

This program invokes the interrupt 65H twice which has been made resident.
Lecture # 4

Another Example:

```c
#include<BIOS.H>
#include<DOS.H>
char st[80] ={"Hello World$"};
char st1[80] ={"Hello Students!$"};
void interrupt (*oldint65)( );
void interrupt newint65( );
void main()
{
    oldint65 = getvect(0x65);
    setvect(0x65, newint65);
    keep(0, 1000);
}
void interrupt newint65( )
{
    if (( _AH ) == 0) //corrected
    {
        _AH = 0x09;
        _DX = (unsigned int) st;
        geninterrupt (0x21);
    }
    else
    {
        if (( _AH ) == 1) //corrected
        {
            _AH = 0x09;
            _DX = (unsigned int) st1;
            geninterrupt (0x21);
        }
    }
}
```

Various interrupts provide a number of services. The service number is usually placed in the AH register before invoking the interrupt. The ISR should in turn check the value in AH register and then perform the function accordingly. The above example exemplifies just that. In this example int 65 is assigned two services 0 and 1. Service 0 prints the string st and service 1 prints the string st1. These services can be invoked in the following manner.

```c
#include<BIOS.H>
#include<DOS.H>
void main()
{
    _AH = 1;
    geninterrupt (0x65);
}
```
AH = 0;
geninterrupt (0x65);
}

Interrupt stealing or interrupt hooks
Previously we have discussed how a new interrupt can be written and implemented. Interrupt stealing is a technique by which already implemented services can be altered by the programmer.

This technique makes use of the fact that the vector is stored in the IVT and it can be read and written. The interrupt which is to be hooked its (original routine ) vector is first read from the IVT and then stored in a interrupt pointer type variable, after this the vector is changed to point to one of the interrupt function (new routine) within the program. If the interrupt is invoked now it will force the new routine to be executed provided that its memory resident. Now two things can be done, the original routine might be performing an important task so it also needs to invoked, it can either be invoked in the start of the new routine or at the end of the new routine using its pointer as shown in the following execution charts below

Fig 1 (Normal Execution of an ISR)
   Execution Interrupted

Fig 2 (The original ISR being called at he end of new routine)
   New Routine
Other form of Interrupt Interception

Care must be taken while invoking the original interrupt. Generally in case hardware interrupts are intercepted invoking the original interrupt at the start of new routine might cause some problems whereas in case of software interrupts the original interrupt can be invoked anywhere.

Sample Program for interrupt Interception

```c
void interrupt newint();
void interrupt (*old)();
void main()
{
    old=getvect(0x08);
    setvect(0x08,newint);
    keep(0,1000);
}
void interrupt newint()
{
...
...
(*old)();
}
```

The above program gets the address stored at the vector of interrupt 8 and stores it in the pointer oldint. The address of the interrupt function newint is then placed at the vector of int 8 and the program is made memory resident. From this point onwards whenever interrupt 8 occurs the interrupt function newint is invoked. This function after performing its operation calls the original interrupt 8 whose address has been stored in oldint pointer.

Timer Interrupt

In the coming few examples we will intercept interrupt 8. This is the timer interrupt. The timer interrupt has following properties.

- It is an Hardware Interrupts
- It is Invoked by Means of Hardware
- It approximately occurs 18.2 times every second by means of hardware.
BIOS Data Area

BIOS contains trivial I/O routines which have been programmed into a ROM type device and is interfaced with the processor as a part of main memory. However the BIOS routines would require a few variables, these variables are stored in the BIOS data area at the location 0040:0000H in the main memory.

One such byte stored in the BIOS data area is the keyboard status byte at the location 40:17H. This contains the status of various keys like alt, shift, caps lock etc. This byte can be described by the diagram below.

Fig 4 (Keyboard status byte)

Another Example

```c
#include <dos.h>
void interrupt (*old)();
void interrupt new();
char far *scr=(char far* ) 0x00400017;
void main()
{
old=getvect(0x08);
setvect(0x08,new);
keep(0,1000);
}
void interrupt new (){*
scr=64;
(*old)();
}
```

This fairly simple example intercepts the timer interrupt such that whenever the timer interrupt occurs the function new() is invoked. Remember this is .C program and not a .CPP program. Save the code file with .C extension after writing this code. On occurrence of interrupt 8 the function new sets the caps lock bit in keyboard status by placing 64 at this position through its far pointer. So even if the user turns off the caps lock on the next occurrence of int 8 (almost immediately) the caps lock will be turned on again (turning on the caps lock on like this will not effect its LED in the keyboard only letters will be typed in caps).
Memory Mapped I/O and Isolated I/O
A device may be interfaced with the processor to perform memory mapped or isolated I/O. Main memory and I/O ports both are physically a kind of memory device. In case of Isolated I/O, I/O ports are used to hold data temporary while sending/receiving the data to/from the I/O device. If the similar function is performed using a dedicated part of main memory then the I/O operation is memory mapped.

Fig 5 (Isolated I/O)

**Isolated I/O**

![Isolated I/O Diagram]

Fig 6 (Memory mapped I/O)

**Memory Mapped I/O**

![Memory Mapped I/O Diagram]

Memory Mapped I/O on Monitor
One of the devices in standard PCs that perform memory mapped I/O is the display device (Monitor). The output on the monitor is controller by a controller called video controller within the PC. One of the reason for adopting memory mapped I/O for the monitor is that a large amount of data is needed to be conveyed to the video controller in order to describe the text or that graphics that is to be displayed. Such large amount of data being output through isolated I/O does not form into a feasible idea as the number of port in PCs is limited to 65536.

The memory area starting from the address b800:0000H. Two bytes (a word) are reserved for a single character to be displayed in this area. The low byte contains the ASCII code of the character to be displayed and the high byte contains the attribute of the character to be displayed. The address b800:0000h corresponds to the character displayed at the top left corner of the screen, the next word b800:0002 corresponds to the next character on the same row of the text screen and so on as described in the diagram below.
Fig 7 (Memory mapped I/O on monitor)

**Memory Mapped I/O ON Monitor**

Low Byte = ASCII CODE  
High Byte = Attribute Byte

The attribute byte (higher byte) describes the forecolor and the backcolor in which the character will be displayed. The DOS screen carries black as the backcolor and white as the fore color by default. The lower 4 bits (lower nibble) represents the forecolor and the higher 4 bits (higher nibble) represents the back color as described by the diagram below.

Fig 8 (Attribute Byte)

**Memory Mapped I/O ON Monitor**

<table>
<thead>
<tr>
<th>Low Byte = Ascii Code</th>
<th>High Byte = Attribute Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Black</td>
</tr>
<tr>
<td>100</td>
<td>Red</td>
</tr>
<tr>
<td>010</td>
<td>Green</td>
</tr>
<tr>
<td>001</td>
<td>Blue</td>
</tr>
<tr>
<td>111</td>
<td>White</td>
</tr>
</tbody>
</table>

To understand all describe above lets take a look at this example.

```c
unsigned int far *scr=0xb8000000;

void main()
{
    (*scr)=0x0756;
    (*(scr+1))=0x7055;
}
```
This example will generate the output VU.
The far pointer scr is assigned the value 0xb800H in the high word which is the segment address and value 0x0000H in the low word which is the offset address. The word at this address is loaded with the value 0x0756H and the next word is loaded by the value 0x7055H, 0x07 is the attribute byte meaning black back color and white fore color and the byte 0x70h means white back color and black fore color. 0x56 and 0x55 are the ASCII value of “V” and “U” respectively.
Lecture # 5

This same task can be performed by the following program as well. In this case the video text memory is accessed byte by byte.

```c
unsigned char far *scr=(unsigned char far*)0xb8000000;
void main()
{
 *scr=0x56;
 *(scr+1)=0x07;
 *(scr+2)=0x55;
 *(scr+3)=0x70;
}
```

The next example fills whole of the screen with spaces. This will clear the contents of the screen.

```c
unsigned char far *scr=(unsigned char far*)0xb8000000; //corrected
t void main()
{
 int i; //instruction added
 for (i=0;i<2000;i++) //corrected
 {
   *scr=0x20; //corrected
   *(scr+1)=0x07; //corrected
   scr=scr+2;
 }
}
```

Usually the in text mode there are 80 columns and 25 rows making a total of 2000 characters that can be shown simultaneously on the screen. This program runs a loop 2000 times placing 0x20 ASCII code of space character in whole of the text memory in this text mode. Also the attribute is set to whiteforecolor and black backcolor.

Another Example
In the following example memory mapped I/O is used in combination with interrupt interception to perform an interesting task.

```c
#include <dos.h>
t void interrupt (*old)();
t void interrupt newfunc();
char far *scr=(char far*) 0xb8000000;
void main()
{
 old=getvect(0x08);
 setvect (0x08,newfunc);
 keep(0,1000);
}
```
void interrupt newfunc ()
{
    *scr=0x41;  //corrected
    *(scr+1)=0x07;  //corrected
    (*old)();
}

In the above example the timer interrupt is intercepted such that whenever the timer interrupt is invoked (by means of hardware) the memory resident newfunc() is invoked. This function simply displays the ASCII character 0x41 or ‘A’ in the top left corner of the text screen.

Here is another example.
#include <stdio.h>
void interrupt (*old)();
void interrupt newfunc();
char far *scr=(char far*) 0xb8000000;
int j;
void main( )
{
    old=getvect(0x08);
    setvect(0x08,newfunc);  //corrected
    keep(0,1000);  //corrected
}

void interrupt newfunc ( )
{
    for ( j=0;j<4000;j+=2){  //corrected
        if(*(scr+j)=='1'){
            *(scr+j)='9';
        }
    }
    (*old)();
}

This program scans through all the bytes of text display memory when int 8 occurs. It once resident will replace all the ‘1’ on the screen by ‘9’. If even somehow a ‘1’ is displayed on the screen it will be converted to ‘9’ on occurrence of interrupt 8 which occurs 18.2 times every second.

The keyboard Interrupt
Keyboard is a hardware device and it makes use of interrupt number 9 for its input operations. Whenever a key is pressed interrupt # 9 occurs. The operating system processes this interrupt in order to process the key pressed. This interrupt usually reads the scan code from the keyboard port and converts it into the appropriate ASCII code and places the ASCII code in the keyboard buffer in BIOS data area as described in the diagram below
Any Process

INT 9
Reads Scan Code converts to ASCII & place it in Keyboard Buffer & returns

Keyboard Controller

60H

Kbd

INT

Lets now experiment on the keyboard interrupt to understand its behavior

Example

```
#include <dos.h>
void interrupt (*old)( );
void interrupt newfunc( );

void main( )
{
    old = getvect(0x09);
    setvect(0x09,newfunc);
    keep(0,1000);
}
void interrupt newfunc ( )
{
    (*old)( );
    (*old)( );
    (*old)( );
}
```

This program simply intercepts the keyboard interrupt and places the address of newint in the IVT. The newint simply invokes the original interrupt 9 thrice. Therefore the same character input will be placed in the keyboard buffer thrice i.e three characters will be received for each character input.

Example

```
#include <dos.h>
void interrupt (*old)( );
void interrupt newfunc( );
char far *scr = (char far* ) 0x00400017;
```
void main( )
{
    old = getvect(0x09);
    setvect(0x09,newfunc);
    keep(0,1000);
}

void interrupt newfunc( )
{
    *scr = 64;
    (*old)( );
}

The above program is quite familiar it will just set the caps lock status whenever a key is pressed. In this case the keyboard interrupt is intercepted.

Example

void interrupt (*old)( );
void interrupt newfunc( );
char far *scr = (char far*) 0xB8000000;
int j;
void main( )
{
    old = getvect(0x09);
    setvect(0x09,newfunc);
    keep(0,1000);
}

void interrupt newfunc ( )
{
    for( j = 0; j < 4000; j += 2)
    {
        if (*(scr +j) == '1')
            *(scr + j) = '9';
    }
    (*old)( );
}

This too is a familiar example. Whenever a key is pressed from the keyboard the newfunc functions runs through whole of the test display memory and replaces the ASCII ‘1’ displayed by ASCII ‘9’.
Timer & Keyboard Interrupt Program

#include <dos.h>
void interrupt (*oldTimer)( ); //corrected
void interrupt (*oldKey)( ); //corrected
void interrupt newTimer ( );
void interrupt newKey ( );
char far *scr = (char far* ) 0xB8000000;
int i, t = 0, m = 0;
char charscr [4000];
void main( )
{  
    oldTimer = getvect(8);
    oldKey = getvect (9);
    setvect (8,newTimer);
    setvect (9,newKey);
    getch();
    getch();
    getch();
    getch();
}

void interrupt newTimer ( )
{  
    t++;
    if ((t >= 182) && (m == 0))
    {
        for (i =0; i < 4000; i++)
      
            charscr [i] = *(scr + i);
        for (i =0; i <=4000; i +=2)
        {
            *(scr + i) = 0x20;
            *(scr + i + 1) = 0x07;
        }
    
        t = 0; m = 1;
    }

    (*oldTimer) ( );
}

void interrupt newKey ( )
{  
    int w;
    if (m == 1)
    {
        for (w =0; w < 4000; w++)
      
            *(scr + w) = charscr [w];
            m = 0;
        }

    (*oldKey) ( );
}

This program works like a screen saver. The newTimer function increments \( t \) whenever it is invoked so the value of \( t \) reaches 182 after ten second. At this moment the function saves the value in display text memory in a character array and fills the screen with spaces and sets a flag \( m \). The newKey function is invoked when a key press occurs.
The flag is checked if the it’s set then the screen is restored from the values saved in that character array.

**Reentrant Procedures & Interrupt**

If on return of a function the values within the registers are unchanged as compared to the values which were stored in registers on entry into the procedures then the procedure is called reentrant procedure. Usually interrupt procedures are reentrant procedures especially those interrupt procedure compiled using C language compiler are reentrant. This can be understood by the following example

\[ AX = 1234H \]

\[ Proc1() \]

\[ AX = ? \]

\[ AX = FF55H \]

In the above example the function Proc1() is invoked. On invocation the register AX contained the value 1234H, the code within the function Proc1() changes the value in AX to FF55H. On return AX will contain the value 1234H if the function have been implemented as a reentrant procedure i.e a reentrant procedure would restore the values in registers their previous value (saved in the stacked) before returning.

C language reentrant procedures save the registers in stack following the order AX, BX, CX, DX, ES, DS, SI, DI, BP on invocation and restores in reverse order before return.

This fact about reentrant procedures can be analysed through following example.

```c
#include <stdio.h>
void interrupt *old();
void interrupt newint()
void main ()
{
  old = getvect(0x65);
  setvect(0x65,newint);
  _AX=0xf00f;
  geninterrupt(0x65);
  a = _AX
  printf("%x",a);
}
```
void interrupt newint()
{
    _AX=0x1234;
}

Firstly its important to compile this above and all the rest of the examples as .C files and not as .CPP file. It these codes are compiled using .CPP extension then there is no surety that this program could be compiled. Again int 65H is used for this experiment. The int 65H vector is made to point at the function newint(). Before calling the interrupt 65H value 0xF00F is placed in the AX register. After invocation of int 65H the value of AX register is changed to 0x1234. But after return if the value of AX is checked it will not be 0x1234 rather it will be 0xF00F indicating that the values in registers are saved on invocation and restore before return and also that the interrupt type procedures are reentrant.
The typical sequence in which registers will be pushed and popped into the stack on invocation and on return can be best described by the following diagrams:

```
Push AX, BX, CX, DX, ES, DS, SI, DI, BP

Push flags, CS, IP

Pop BP, DI, SI, DS, ES, DX, CX, BX, AX

Pop IP, CS, flags

---
---
---
---
Push AX, Push BX,
Push CX, Push DX,
Push ES, Push DS,
Push SI, Push DI, Push BP

---
---
---
---
---
---
---
---
---
---
Pop BP, Pop DI,
Pop SI, Pop DS,
Pop ES, Pop DX,
Pop CX, Pop BX, Pop AX,
IRET
```

The registers Flags, CS and IP are pushed on execution of INT instruction and executions branches to the interrupt procedure. The interrupt procedure pushes register AX, BX, CX, DX, ES, DS, SI, DI, BP in this order. The interrupt procedure then executes, before returning it pops all the registers in the reverse order as BP, DI, SI, DS, ES, DX, CX, BX and AX. IP, CS and flags are popped on execution of the IRET instruction.

Next diagram shows the status of the stack after invocation of the interrupt procedure.
The arguments in simple procedure or functions are saved in the stack for the scope of the function/procedure. When an argument is accessed in fact stack memory is accessed. Now we will take a look how stack memory can be accessed for instance in case of interrupt procedures to modify the value of register in stack.

**Accessing Stack Example**

```c
void interrupt newint ( unsigned int BP,unsigned int DI,
unsigned int SI,unsigned int DS, unsigned int ES,unsigned int DX, unsigned int CX,unsigned int BX, unsigned int AX,unsigned int IP,
unsigned int int CS,unsigned int flags)
{//corrected
    a = AX;
    b = BX;
    d = ES;
}
```

Although interrupt do not take parameters through stack but an interrupt procedure can still have parameters. This parameter list can be used to access the stack. The leftmost parameter accesses the item on top of the stack and the rest of the parameters accesses deeper into the stack according to its order toward left. In the above example value of AX in stack is moved in a, the value of BX is moved into b and the value of ES is moved into d.
Example:

```c
void interrupt newint ( unsigned int BP,unsigned int DI, unsigned int SI, unsigned int DS, unsigned int ES, unsigned int DX, unsigned int CX, unsigned int BX, unsigned int AX, unsigned int IP, unsigned int CS, unsigned int flags) //corrected
{
    AX = 0xF00F;
}
void main ( ) {
    setvect(0x65,newint);
    AX = 0x1234;
geninterrupt (0x65);
a = __AX;
printf ("%x", a);
}
```

In this example the value on invocation in AX is 0x1234, the interrupt procedure does not change the current value of the register through pseudo variables rather it changes the corresponding of AX in stack which will be restored in AX before return.

**Disk Interrupt**
The following example makes use of disk interrupt 13H and its service 3H. The details of this service are as under.

**On Entry**
- **AH** = Service # = 03
- **AL** = No of Blocks to write
- **BX** = Offset Address of Data
- **CH** = Track # , **CL** = Sector #
- **DH** = Head #
- **DL** = Drive #(Starts from 0x80 for fixed disk & 0 for removable disks)
- **ES** = Segment Address of data buffer.

**On Exit**
- **AH** = return Code
- Carry flag = 0 ( No Error AH = 0)
- Carry flag = 1 ( Error AH = Error Code)

Boot block is a special block on disk which contains information about the operating system to be loaded. If the data on boot block is somehow destroyed the disk would be rendered inaccessible. The address of partition block on hard disk is head # =1, track# = 0 and sector # = 1. Now let’s write an application that will protect the boot block to be written by any other application.
#pragma inline
#include <dos.h>
#include <bios.h>
void interrupt (*oldtsr) ( );
void interrupt newtsr {unsigned int BP, ..., flags};
//must provide all the arguments
void main ( )
{
    oldtsr = getvect (0x13);
    setvect(0x13, newtsr); //corrected
    keep (0, 1000);
}
void interrupt newtsr(unsigned int BP, unsigned int DI,
unsigned int SI, unsigned int DS, unsigned int ES, unsigned int
DX, unsigned int CX, unsigned int BX, unsigned int AX,
unsigned int IP, unsigned int CS,
unsigned int flags) //corrected
{
    if ( _AH == 0x03)
    if(( _DH == 1 && _CH == 0 && _CL == 1)& _DL >= 0x80)
    {
        asm clc;
        asm pushf;
        asm pop flags;
        return;
    }
    _ES = ES; _DX = DX;
    _CX = CX; _BX = BX;
    _AX = AX;
    *oldtsr;
    asm pushf;
    asm pop flags;
    AX = _AX; BX = _BX;
    CX = _CX; DX = _DX;
    ES = _ES;
}

The above program intercepts interrupt 13H. The new interrupt procedure first check AH
for service number and other parameters for the address of boot block. If the boot block is
to be written it simply returns and clears the carry flag before returning to fool the calling
program that the operation was successful. And if the boot block is not to be written then
it places the original parameters back into the registers and calls the original interrupt.
The values returned by the original routine are then restored to the corresponding register
values in the stack so that they maybe updated into the registers on return.
The keyboard Hook
The service 15H/4FH is called the keyboard hook service. This service does not perform any useful output, it is there to be intercepted by applications which need to alter the keyboard layout. It called by interrupt 9H after it has acquired the scan code of input character from the keyboard port while the scan code is in AL register. When this service returns interrupt 9H translates the scan code into ASCII code and places it in buffer. This service normally does nothing and returns as it is but a programmer can intercept it in order to change the scan code in AL and hence altering the input or keyboard layout.

Key Pressed
Move Scan Code from 60H port to AL
Convert to ASCII & place it in keyboard buffer

The following application show how this can be done.

```c
#include <dos.h>
#include <bios.h>
#include <stdio.h>
void interrupt (*oldint15) ( );
void interrupt newint15(unsigned int BP, ..., flags);
void main ( )
{
    oldint15 = getvect (0x15);
    setvect (0x15, newint15);
    keep (0, 1000);
}
void interrupt newint15(unsigned int BP, unsigned int DI,
unsigned int SI, unsigned int DS, unsigned int ES, unsigned int DX,
unsigned int CX, unsigned int BX, unsigned int AX,
unsigned int IP, unsigned int CS,
unsigned int flags)
{
    if (*((char*)&AX) == 0x4F )
    {
        if (*((char*)&AX) == 0x2C)
            *((char*)&AX) = 0x1E;
        else if (*((char*)&AX) == 0x1E)
            *((char*)&AX) = 0x2C;  //corrected
    }
    else
        (*oldint15)();
}
```
The application intercepts interrupt 15H. The newint15 function checks for the service #4F in the high byte of AX, if this value is 4F the definitely the value in AL with be the scan code. Here a simple substitution have been performed 0x1E is the scan code of ‘A’ and 0x2C is the scan code of ‘Z’. If the scan code is AL is that of ‘A’ it is substituted with the scan code of ‘Z’ and vice versa. If some other service of 15H is invoked the original interrupt function is invoked.
Lecture # 7

The microprocessor package has many signals for data, control and addresses. Some of these signals may be input signals and some might be output. Hardware interrupts make use of two of such input signals namely NMI (Non maskable Interrupt) & INTR(Interrupt Request).

**Microprocessor**

- **Reset**
- **Hold**
- **NMI**
- **INTR**

NMI is a higher priority signal than INTR, HOLD has even higher priority and RESET has the highest priority. If any of the NMI or INTR pins are activated the microprocessor is interrupted on the basis of priority, if no higher priority signals are present. This is how microprocessor can be interrupted without the use of any software instruction hence the name hardware interrupts.

**Hardware Interrupt and Arbitration**

Most of the devices use the INTR line. NMI signal is used by devices which perform operations of the utmost need like the division by zero interrupt which is generated by ALU circuitry which performs division. Definitely this operation is not possible and the circuitry generates an interrupt if it receives a 0 as divisor from the control unit.

INTR is used by other devices like COM ports LPT ports, keyboard, timer etc. Since only one signal is available for microprocessor interruption, this signal is arbitrated among various devices. This arbitration can be performed by a special hardware called the Programmable Interrupt Controller (PIC).
Interrupt Controller
A single interrupt controller can arbitrate among 8 different devices.

As it can be seen from the diagram above the PIC device has 8 inputs IRQ0-IRQ7. IRQ0 has the highest priority and IRQ7 has the lowest. Each IRQ input is attached to an I/O device whenever the device requires an I/O operation it sends a signal to the PIC. The PIC on the basis of its priority and presence of other requests decides which request to serve. Whenever a request is to be served by PIC it interrupt the processor with the INT output connected to the INTR input of the processor and send the interrupt # to be generated the data lines connected to the lower 8 datelines of the data bus to inform the processor about the interrupt number. In case no higher priority signal is available to the processor and the processor is successfully interrupted the microprocessor sends back an INTA (interrupt Acknowledge) signal to inform the PIC that the processor has been interrupted.

The following diagram also shows the typical connectivity of the IRQ lines with various devices
In standard PCs there maybe more than 8 devices so generally two PIC are used for INTR line arbitration. These 2 PICs are cascaded such that they collectively are able to arbitrate among 16 devices in all as shown in the following diagram.

The PICs are cascaded such that a total of 16 IRQ levels can be provided number IRQ0-IRQ15. The IRQ level 2 is used to cascade both of the PIC devices. The Data lines are multiplexed such that the interrupt number is issued by the concerned PIC. The IRQ 2 input of the Master PIC is connected to the INT output of the Slave PIC. If the slave PIC is interrupted by a device its request ins propagated to the master PIC and the master PIC ultimately interrupts the processor on INTR line according to the priorities.

In a standard PC the PICs are programmed such that the master PIC generated the interrupt number 8-15 for IRQ0–IRQ7 respectively and the slave PIC generates interrupt number 70-77H for IRQ8-IRQ15

**Hardware Interrupts are Non-Preemptive**

As described earlier IRQ 0 has the highest priority and IRQ 15 has the lowest priority. If a number of requests are available instantaneously the request with higher priority will be sent for service first by the PIC. Now what will happen if a lower priority interrupt is being service and a higher priority interrupt request occurs, will the lower priority interrupt be preempted? The answer is that the interrupt being serviced will not be preempted no matter what. The reason for this non-preemptive can be understood by the example illustrated as below. Lets first consider that the hardware interrupts are preemptive for argument sake. If a character ‘A’ is input a H/W interrupt will occur to process it, while this interrupt is being processed another character is input say ‘B’ in case the interrupts have been preemptive the previous instance will be preempted and another instance for the H/W interrupt call will be generated, and similarly consider another character is input ‘C’ and the same happened for this input as well. In this case the character first to be fully processed and received will be ‘C’ and then ‘B’ will be
processed and then ‘A’. So the sequence of input will change to CBA while the correct sequence would be ABC.

The input will be received in correct sequence only if the H/W interrupts are non-preemptive as illustrated in the diagram below.

Hardware interrupts requires something more to be programmed into them as compared with software interrupts. The major difference is because of the reason given above that the H/W interrupts are non-preemptive. To make them non-preemptive the PIC should know when the previously issued interrupt returns. The PIC cannot issue the next pending interrupt unless it is notified that the previous interrupt has returned.

Who Notifies EOI (End of interrupt)
The PIC has to be notified about the return of the previous interrupt by the ISR routine. From programmer point of view this is the major difference between H/W and software interrupt. A software interrupt will not require any such notification. As the diagram below illustrates that every interrupt returns with an IRET instruction. This instruction is executed by the microprocessor and has no linkage with the PIC. So there has to be a different method to notify the PIC about the End of interrupt.
Pending Hardware interrupts.
While a hardware interrupt is being processed a number of various other interrupt maybe pending. For the subtle working of the system it is necessary for the In-service hardware interrupt to return early and notify the PIC on return. If this operation takes long and the pending interrupt requests occur repeated there is a chance of loosing data.

Programming the PIC
To understand how the PIC is notified about the end of interrupt lets take a look into the internal registers of PIC and their significance. A PIC has a number of initialization control words (ICW) and operation control words (OCW), following is characteristic of ICW and OCWs
- ICW programmed at time of boot up
- ICW are used to program the operation mode like cascade mode or not also it is used to program the function of PIC i.e if it is to invoke interrupts 08~ 0FH or 70-77H on IRQ request.
- OCW are used at run-time.
- OCW is used signal PIC EOI
- OCW are also used to read/write the value of ISR(In-service register), IMR(interrupt mask register), IRR(interrupt request register).

To understand the ISR, IMR and IRR lets take a look at the following diagram illustrating an example.

The values shown in the various registers illustrate that the currently in-service interrupt is that generated through IRQ4 of the PIC (int 0CH in case of mater PIC), also the interrupt through IRQ1 has been masked (int 9h (keyboard interrupt) in case of master PIC) which means that even though a request for this interrupt is received by the PIC but this request is ignored by the PIC until this bit is cleared. And the requests through IRQ7, IRQ6, IRQ2 and IRQ0 are pending and waiting for the previously issued interrupt to return.

Port Addresses
Few of the operation control words can be altered after boot time. The addresses for these OCW are listed as below
- Master PIC has two ports
  20H=OCW for EOI
  21H=OCW for IMR
• Slave PIC has two ports as well
  A0H=OCW for EOI code
  A1H=OCW for IMR

Let’s now discuss an example that accesses these ports to control the PIC

```
#include <stdio.h>
#include <bios.h>

void main()
{
  outport(0x21,0x02);
}
```

This example simply accesses the bit # 1 of IMR in the master PIC. It sets the bit #1 in IMR which masks the keyboard interrupt. As a result no input could be received from the keyboard after running this program.

Let’s now look at another example

```
#include <dos.h>
#include <stdio.h>
#include <bios.h>

void interrupt (*oldints)();
void interrupt newint8();
int t=0; //corrected

void main()
{
  oldints=getvect(0x08);
  setvect(0x08,newint8);
  keep(0,1000);
}

void interrupt newint8()
{
  t++:
  if (t==182)
    { outport(0x21,2); }
  else{
    if (t==364)
    {
      outport(0x21,0);
      t=0;
    }
  }
  (*oldints)();
}
```
The example above is also an interesting example. This program intercepts the timer interrupt. The timer interrupt makes use of a variable to keep track of how much time has passed; \( t \) is incremented each time int 8 occurs. If it reaches the 182 after 10 second, at this point the keyboard interrupt is masked and remains masked for subsequent 10 second at which point the value of \( t \) will be 364, also \( t \) is cleared to 0 for another such round.

```c
#include <dos.h>
void interrupt (*old)();
void interrupt newint9();
char far *scr=(char far *) 0x00400017;

void main()
{
  old=getvect(0x09);
  setvect(0x09,newint9);
  keep(0,1000);
}

void interrupt newint9()
{
  if (inportb(0x60)==83
    && (((*scr)&12)==12)) //corrected
  {
    outportb(0X20,0x20);
    return;
  }

  (*old)();
}
```

The above program disables the CTRL+ALT+DEL combination in the DOS environment (if windows OS is also running this combination will not be disabled for its environment). The keyboard interrupt has been intercepted, whenever the keyboard interrupt occurs the newint9 function receives the scan key code from the keyboard port 0x60, 83 is the scan key code of DEL key. Also the program checks if the ALT and CTRL button has been pressed as well from the status of the 40:17H keyboard status byte. If it confirms that the combination pressed is CTRL+ALT+DEL then it does not invoke the real int 9 ( *oldint() which will make the computer reboot in DOS environment had the computer been booted through DOS) and simply returns. But notice that before returning it notifies the PIC that the interrupt has ended. The EOI code sent to the OCW at the address 0x20 is also 0x20. This is being done because int 9 is a hardware interrupt, had this been a software interrupt this would have not been required.
```c
#include <dos.h>

void interrupt (*old)();

void interrupt newint9();

void main()
{

old=getvect(0x09);
setvect(0x09, newint9);
keep(0,1000);
}

void interrupt newint9()
{

if (inportb(0x60)==0x1F) //corrected
{

    outportb(0X20,0x20);
    return;
}

(*old)();
}
```

The above C language program suppresses the ‘s’ input from the keyboard. The keyboard interrupt has been intercepted. When a key is pressed newint9 is invoked. This service checks the value through the import statement of the keyboard port numbered 0x60. If he scan code (and not the ASCII code) is 0x1F then it indicates that the ‘s’ key was pressed. This program in this case simply returns the newint9 hence suppressing this input by not calling the real int 9. Before return it also notifies the PIC about the end of interrupt.
The keyboard buffer

Keyboard Buffer

- Keyboard Buffer is located in BIOS Data Area.
- Starts at 40:IEH
- Ends at 40:3DH
- Has 32 bytes of memory 2 bytes for each character.
- Head pointer is located at address 40:1A to 40:IBH
- Tail pointer located at address 40:IC to 40:IDH

The keyboard buffer is a memory area reserved in the BIOS data area. This area stores the ASCII or special key codes pressed from the keyboard. It works as a circular buffer and two bytes are reserved for each character, moreover 2 bytes are used to store a single character. The first character stores the ASCII code and the second byte stores 0 in case an ASCII key is pressed. In case a extended key like F1- F12 or arrow key is pressed the first byte stores a 0 indicating a extended key and the second byte stores its extended key code.
Circular buffer

The circular keyboard buffer starts at the address 40:1EH and contains 32 bytes. The address 40:1AH stores the head of this circular buffer while the address 40:1CH stores the tail of this buffer. If the buffer is empty the head and tail points at the same location as shown in the diagram above.

Storing characters in the keyboard buffer

The above slide shows how characters are stored in the buffer. If ‘A; is to be stored then the first byte in the buffer will store its ASCII code and the second will store 0, and if
extended key like DEL is to be stored the first byte will store 0 and the second byte will store its scan code i.e 83. The diagram also shows that head points to the next byte where the next input character can be stored. Also notice that head contain the offset from the address 40:00H and not from address 40:1EH. i.e it contain 0x24 which is the address of the next byte to be stored relative to the start of BIOS data area and not the keyboard buffer.

Position of tail

As discussed earlier the keyboard buffer is a circular buffer therefore the tail need to be placed appropriately. In the given example the input ‘A’ stored in the buffer is consumed. On consumption of this character the tail index is updated so that it points to the next character in the buffer. In brief the tail would point to the next byte to be consumed in the buffer while head points to the place where next character can be stored.
• So KBD buffer acts as a circular buffer.
• The tail value should be examined to get to the start of the buffer.

Example

```c
#include <dos.h>
void interrupt (*old)();
void interrupt new1()!
unsigned char far *scr = (unsigned char far *) 0x0040001C
void main()
{
    old=getvect(0x09);
    setvect(0x09,new1);
    keep(0,100);
}
```
The program listed in the slides above intercepts interrupt 9. Whenever the interrupt 9 occurs it reads the keyboard port 0x60. If the port contains 83 then it means DEL was pressed, if so it places the code 25 in the buffer and then updates the head in circular manner. The code 25 placed instead of 83 represents the combinations CTRL+Y. The program when resident will cause the program to receive CTRL+Y combination whenever DEL is pressed by the user. i.e in Borland C environment CTRL+Y combination is used to delete a line, if this program is resident then in Borland C environment a line will be deleted whenever DEL is pressed by the user. But the thing worth noticing is that the interrupt function returns and does not call the real interrupt 9 after placing 25 in the buffer, rather it returns directly. But before returning as it has intercepted a hardware interrupt it needs to notify the PIC, this is done by outportb(0x20,0x20); statement. 0x20 is the address of the OCW that receives the EOI code which incidentally is also 0x20.
As discussed earlier the slave PIC is cascaded with the master PIC. If the hardware interrupt to be processed is issued by the master PIC then the ISR needs to send the EOI code to the master PIC but if the interrupt request is issued by the slave PIC then the ISR needs to inform both master and slave PICs as both of them are cascaded as shown in the slide.
Reading OCW

The same port i.e 0x20 is used to access the OCWs. 00 is placed in bits number 4 and 3 to indicate an EOI is being received and 01 is placed to indicate that a internal register is to be accessed.

The value in bits number 1 and 0 indicate which Register is to accessed. 10 is for IRR and 11 is for ISR.
Accessing the ISR and IRR.

A value is placed in the port 0x20 as shown in the above slide to indicate that a register is to be accessed.

Then again a value in that same port is placed to indicate which register is to be accessed, as in the above slide IRR is to be accessed.
And in this slide a value is formed which can be programmed in the port 0x20 to access the ISR.

A sample program

```c
#include <stdio.h>
#include <dos.h>
#include <bios.h>
void main (void)
{
    char a;
    outport(0x20,8);
    outport(0x20,0x0A);
    a=inport(0x20);
    printf ("value of IRR is %x\n",a);
    outport(0x20,0x08);
    outport(0x20,0x0B);
    a=inport(0x20);
    printf ("value of ISR is %x\n",a);
}
```

The above program makes use of the technique described to access the ISR and IRR. Firstly 0x08 is placed to specify that a register is to be accessed then 0x0A is placed to indicate that IRR is to be accessed. Now this port 0x20 can be read which will contain the value in IRR. Similarly it is done again by placing the 0x0B in port 0x20 to access the ISR.
More about TSR Programs

- A TSR need to be loaded once in memory
- Multiple loading will leave redundant copies in memory

- So we need to have some check which will load the program only once
Using a Global Variable as a flag

as shown in the slide below

```
int flag;
flag = 1;
keep(0,1000);
if (flag==1)
    Make TSR
else
    exit Program
```

This will not work as this global variable is only global for this instance of the program. Other instances in memory will have their own memory space. So the
Answers is to use a memory area as flag that is global to all programs. 
 i.e. IVT

int 65H is empty, we can use its vector as a flag. 
Address of vector 
    seg = 0 
    offset = 65H * 4

Example:

```c
#include<stdio.h>
#include<BIOS.H>
#include<DOS.H>
unsigned int far * int65vec = MK_FP(0,0x65*4)
void interrupt (*oldint) ( );
void interrupt newfunc ( );
void main()
{
    if(*int65vec) != 0xF00F) //corrected
        { oldint = getvect(0x08);
          setvect(0x08, newint);
          (*int65vec) = 0xF00F;
          keep (0,1000);
        }else
            {
              puts("Program Already Resident");
            }
    void interrupt newfunc()
    {
        :::::::
        :::;
        (*oldint) ( );
    }
```
The above template shows how the vector of int 0x65 can be used as a flag. This template shows that a far pointer is maintained which is assigned the address of the int 0x65 vector. Before calling the keep() function i.e making the program resident a value of 0xf00f is placed at this vector( this vector can be tempered as it is not being used by the OS or device drivers). Now if another instance of the program attempts to run the if statement at the start of the program will check the presence of 0x0f00f at the int vector of 0x65, if found the program will simply exit otherwise it will make itself resident. Or in other word we can say that 0xf00f at the int 0x65 vector in this case indicate that the program is already resident.

➢ But what if another program is resident or using this vector.

Another Method
➢ Service # 0xFF usually does not exist for ISR’s.
➢ Key is to create another service # 0xFF for the ISR interrupt besides other processing.

Example:
#include<stdio.h>
#include<BIOS.H>
#include<DOS.H>
void interrupt (*oldint) ( );
void interrupt newfunc ( unsigned int BP,……,flags);
void main()
{
  _DI = 0;
  _AH = 0xFF;
  geninterrupt (0x13);
  if (_DI = = 0xF00F) {
    puts (“Program Already Resident”);
    exit (0);
  }
The implements the service 0xff of interrupt 0x13 such that whenever this service is called it returns 0xf00f in DI and if this value does not return then it means that this program is not resident.

```c
Example:

else
{
    oldint = getvect (0x13);
    setvect (0x13, newint);
    keep (0, 1000);
}

void interrupt newint ( )
{
    if (_AH == 0xFF)
    {
        DI = 0xF00F;
        return;
    }
    else
    {
        (*oldint) ( );
    }
}
```
The interval timer is used to divide an input frequency. The input frequency used by the interval timer is the PCLK signal generated by the clock generator. The interval timer has three different each with an individual output and memory for storing the divisor value.
Dividing Clock signal

**Counter Registers:**

- Counter registers can be used to divide frequency.

<table>
<thead>
<tr>
<th>count</th>
<th>/16</th>
<th>/8</th>
<th>/4</th>
<th>/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A counter register can be used to divide the clock signal. As shown in the slide above the but 0 of the clock register is used to divide the clock frequency by 2 subsequently bit 1 is used to divide it by 4 and so on.

**The Division mechanism**

<table>
<thead>
<tr>
<th>0000 0000</th>
<th>0000 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001</td>
<td>0000 1001</td>
</tr>
<tr>
<td>0000 0010</td>
<td>0000 1010</td>
</tr>
<tr>
<td>0000 0011</td>
<td>0000 1011</td>
</tr>
<tr>
<td>0000 0100</td>
<td>0000 1100</td>
</tr>
<tr>
<td>0000 0101</td>
<td>0000 1101</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000 1110</td>
</tr>
<tr>
<td>0000 0111</td>
<td>0000 1111</td>
</tr>
</tbody>
</table>

The above slide shows a sequence of output that a 8bit clock register will generate in sequence whenever it receives the clock signal. Observe bit #1, its value changes between 0 and 1 between two clock cycles so it can be used to divide the basic frequency by 2.
Similarly observe bit #2 its value transits between 0 and 1 within 4 clock cycles hence it divides the frequency by 4 and so on.

Timing diagram

Here is the timing diagram for above example. Bit #1 performs one cycle in between 2 clock cycles. Similarly bit #2 performs one cycle in between 4 clock cycles and so on.
Command registers within the programmable interval timer

**Interval Timer Programming: Command Registers**

- 8-bit Command port
- Need to be programmed before loading the divisor value for a channel.
- 3 channels, each requires a 16-bit divisor value to generate the output frequency.

Command register and the channels need to be programmed for the interval timer to generate a wanted frequency.

**Command Register**

Command register is an 8 bit register which should be programmed as described in the slide above before loading the divisor value. It signifies the channel to be programmed, size of divisor value, the mode in which the channel is to be operated and also whether the counter is to be used a binary or BCD counter.
Mode Description

The interval timer can operate in six modes. Each mode has a different square wave pattern according to need of the application. Some modes might be suitable to control a motor and some might be suitable to control the speaker.

Binary counter

The interval timer channels can be used as a binary as well as a BCD counter. In case its used in binary mode its counter registers will count in binary sequence and if its used as a BCD counter its registers will count in BCD sequence as described above.
Ports and Channels

Ports & Channels:

- 3-Channels 16-bit wide divisor value
  i.e 0~65535

8-bit port for each channel therefore the divisor word is loaded serially byte by byte.

Port Addresses
- 43H = Command Port
- 40H = 8-bit port for Channel 0
- 41H = 8-bit port for Channel 1
- 42H = 8-bit port for Channel 2

The interval timer has 3 channels each channel is 16 bit wide. The port 43H is an 8 bit port used as the command register. Ports 40h, 41H and 42H are associated with the channels 0, 1 and 2 respectively. Channels are 16 bit wide whereas the ports are 8 bit wide. A 16 bit value can be loaded serially through the ports into the register.

Steps for programming the interval timer

Programming Concepts for Interval Timer:

- Load the Command byte into command register required to program the specific channel.
- The divisor word is then Serially loaded byte by byte.
The port 61H

![Diagram of 61H Port]

the port 61h is used to control the speaker only the least significant 2 bits are important. Bit 0 is used to connect the interval timer to the speaker and the bit #1 is used to turn the speaker on off. Rest of the bits are used by other devices.

Example

```c
//Program loads divisor value of 0x21FF
//Turns ON the speaker and connects it to Interval Timer
#include<BIOS.H>
#include<DOS.H>
void main()
{
    outportb (0x43,0xB4);
    outportb (0x42,0xFF);
    outportb (0x42,0x21);
    outportb (0x61,inportb(0x61) | 3);
    getch();
    outportb (0x61,inportb(0x61) & 0xFC);
}
```

The above programs the interval timer and then turns it on. A value of 0xb4 is loaded into the command register 0x43. This value signifies that the channel 2 is to programmed,
both the bytes of divisor value are to loaded, the interval timer is to be programmed in mode 2 and is to be used as a binary counter. 

Then the divisor value say 0x21ffH, is loaded serially. First 0xFF low byte and then the high byte 0x21 is loaded. Both the least significant bits of 0x61 port are set to turn on the speaker and connect it to the interval timer.

On a key press the speaker is again disconnected and turned off.

Producing a Delay in a Program

```c
Timer Count:
40:6CH
Incremented every 1/18.2 seconds. Whenever INT8

unsigned long int far *time = (unsigned long int far*) 0x0040006C
void main()
{
    unsigned long int tx;
    tx = (*time);
    tx = tx + 18;
    puts("Before");
    while(*time <= tx);
    puts("After");
}
```

Delay can be produced using double word variable in the BIOS Data area placed at the location 0040:006C. This value contains a timer count and is incremented every 1/18 th of a second. In this program the this double word is read, placed in a program variable and incremented by 18. The value of 40:6cH is compared with this variable in a loop. This loop iterates until the value of 40:6cH is not greater. In this way this loop will keep on iterating for a second approximately.
Lecture # 10
Sample Program

```c
unsigned long int * time = (unsigned long int *) 0x0040006C
void main()
{
    unsigned long int tx;
    unsigned int divisor = 0x21FF;
    while (divisor >= 0x50) {
        outportb(0x43,0xB4);
        outportb(0x42,*((char*)(&divisor)));
        outportb(0x42,*((char*)(&divisor))+1));
        outportb(0x61,inportb(0x61) | 3);
        tx = *time;
        tx = tx + 4;
        while (*time <= tx);
        divisor = divisor -30;
    }
}
```

The inner while loop in the program is used to induce delay. The outer loop simply reloads the divisor value each time it iterates after reducing this value by 30. In this way the output frequency of the interval timer changes after every quarter of a second approximately. The speaker will turn on with a low frequency pitch and this frequency will increase gradually producing a spectrum of various sound pitches.
Sample Program

```c
#include <dos.h>
#include <bios.h>

void interrupt (*oldint15) ();
void interrupt newint15 (unsigned int BP, unsigned int DI,
                         unsigned int SI, unsigned int DS, unsigned int ES,
                         unsigned int DX, unsigned int CX, unsigned int BX,
                         unsigned int AX, unsigned int IP, unsigned int CS,
                         unsigned int flags);

void main ( )
{
    oldint15 = getvect (0x15);
    setvect (0x15, newint15);
    keep (0, 1000);
}
```

```c
void interrupt newint15( unsigned int BP, unsigned int DI,
                         unsigned int SI, unsigned int DS, unsigned int ES,
                         unsigned int DX, unsigned int CX, unsigned int BX,
                         unsigned int AX, unsigned int CS, unsigned int IP,
                         unsigned int flags)
{
    if (_AH == 0x4F)
    {
        if (_AL == 0x1F)
        {
            outport (0x43, 0xB4);
            outport (0x42, 0xFF);
            outport (0x42, 0x21);
            outport (0x61, inport(0x61) ^ 3);
        }
    }
    else
    {
        (*oldint15) ();
    }
}
```

The above program is a TSR program that can be used to turn the speaker on/off. The above program intercepts the int 15h. Whenever this interrupt occurs it looks for service #0x4f (keyboard hook). If ‘S’ (0x1f scan code) has been pressed it toggles the speaker.
Sample Program

```c
#include <dos.h>
#include <bios.h>
unsigned int divisors[4]={0x21ff,0x1d45,0x1b8a,0x1e4c};
unsigned long int far *time = (unsigned long int far *)0x0040006C;
void main ()
{
    unsigned long int tx;
    int i=0;
    while (!kbhit())
    {
        while (i<4)
        {
            outport(0x43,0xB4);
            outport(0x42,*((char *)&divisor[i]));
            outport(0x42,*(((char *)&divisor[i])+1));
            outport(0x61, inport(0x61)|3);
            tx=*time;
            tx=tx+4;
            while (tx >= (*time));
            i++;
        }
        i=0;
    }
    outport(0x61,inport(0x61)&0xFC);
}
```

This program generates a tune with 4 different pitches. This program is quite similar to the one discussed earlier. The only major difference is that in that program the pitch was gradually altered from low to high in this the pitches change periodically until a key is pressed to terminate the outer loop. Four various pitches are maintained and their divisor values are placed in the divisors[] array. All these divisor values are loaded one by one after a delay of approximately quarter of a second and this continues until a key is pressed.
Sample Program

```c
#include <stdio.h>
#include <dos.h>
#include <bios.h>

struct tagTones
{
    unsigned int divisor;
    unsigned int delay;
};
struct tagTones Tones[4]={{0x21ff,3},{0x1d45,2},{0x1b8a,3},{0x1e4c,4}};
int i, ticks, flag=0;

void interrupt (*oldint15)();
void interrupt (*oldint8)();
void interrupt newint15();
void interrupt newint8();

unsigned char far *scr = (unsigned char far*)(0x00400017);

void main()
{
    oldint15=getvect(0x15);
    setvect(0x15,newint15);
    oldint8=getvect(0x08);
    setvect(0x08,newint8);
    keep(0,1000);
}
```

This is an interrupt driven version of the previous program. This program makes use of the timer interrupt rather than a loop to vary the divisor value. Moreover interrupt 15 is used to turn the speaker on /off.
The speaker turns on whenever ‘S’ (scan code 0x1f) is pressed and turns off whenever ‘A’ (scan code 0x1E) is pressed.

The timer interrupt shifts the divisor value stored in the tones structure whenever the required number of ticks (timer counts) have passed as required by the value stored in the delay field of the tone structure.

More such divisor values and their delays can be initialized in the tones structure to generate an alluring tune.
Peripheral Programmable interface (PPI)

PPI is used to perform parallel communication. Devices like printer are generally based on parallel communication. The principle of parallel communication is explained in the slide above. It's called parallel because a number of bits are transferred from one point to another parallel on various lines simultaneously.
Advantages of Parallel communication

- Faster
- Only Economically Feasible For Small Distances
Lecture # 11

Programmable Peripheral Interface (PPI)

- Device Used as Parallel port Interface (I/O controller) is PPI

The PPI acts as an interface between the CPU and a parallel I/O device. A I/O device cannot be directly connected to the buses so they generally require a controller to be placed between the CPU and I/O device. One such controller is the PPI. Here we will see how we can program the PPI to control the device connected to the PPI which generally is the printer.
Int 17H is used to control the printer via the BIOS. The BIOS functions that perform the printer I/O are listed in the slide above with its other parameter i.e DX which contains the LPT number. A standard PC can have 4 PPI named LPT1, LPT2, LPT3 and LPT4.
### Status Byte

The above listed function returns a status byte in the AH register whose meaning is described in the slide above. Various bits of the byte describe the status of the printer.

### Time out Byte

The BIOS service once invoked will try to perform the requested operation on the printer repeated for a certain time period. In case if the operation is rendered unsuccessful due to any reason BIOS will not quit trying and will try again and again until the number of tries specified in the timeout bytes shown above runs out.
Accessing the Parallel Port Through BIOS Functions

- Specify the number of Attempts BIOS perform before giving a time out Error
- This byte Varies Depending upon the speed of the PC
- Busy = 0 Printer is Busy
- Busy = 1 Printer is not Busy

Importance of Status Byte

Importance of the Status Byte

```
If((pstate&0x29)!=0) or 
((pstate&0x80)==0) or 
((pstate&0x10)==0) 
{printerok=FALSE;}
else
{printerok=TRUE;}
```

The status of the printer can be used in the above described manner to check if the printer can perform printing or not. In case there is a transfer error, the printer is out of paper or there is a timeout the printer could not be accessed. Or if the printer is busy or if the printer is offline the printer cannot be accessed. The pseudo is just performing these checks.
Importance of the Status Byte

17H/00H Write a character on entry
AH=00
AL=ASCII code
DX=Interface#
On exit
AH=Status Byte

17H/01H Initialize Printer on entry
AH=01
DX=Interface#
On exit
AH=Status Byte

17H/02H Get Printer Status on entry
AH=02, DX=Interface# On exit AH=Status Byte

Printing Programs
Sample Program

### Printing Program

```c
union REGS regs; FILE *fptr;
void main(void)
{
  fptr=fopen("c:\temp\abc.txt","rb");
  regs.h.ah=1;
  regs.x.dx=0;
  int86(0x17,&regs,&regs);
  while(!feof(fptr))
  {
    regs.h.ah=2;
    regs.x.dx=0;
    int86(0x17,&regs,&regs);
    if ((regs.h.ah & 0x80)==0x80)
    {
      regs.h.ah=0;
      regs.h.al=getc(fptr);
      int86(0x17,&regs,&regs);
    }
  }
}
```

The above program performs programmed I/O on the printer using BIOS services. The program firstly initializes the printer int 17H/01. The while loop will end when the end of file is reached, in the loop it checks the printer status (int 17h/02) and write the next byte in the file if the printer is found idle by checking the most significant bit of the status byte.

Sample Program

```c
#include <dos.h>
void interrupt (*old)( );
void interrupt newint ( );
main( )
{
  old = getvect(0x17);
  setvect(0x17,newint);
  keep(0,1000);
}
void interrupt new ()
{
  if (_AH==0)
  {
    if ((_AL=='A') || (_AL=='Z')) //corrected
    {
      (*old);
    }
    else
    {
      //corrected
      (*old);
    }
  }
}
```

The above program intercepts int 17H. Whenever a certain program issues int 17H to print a character the above TSR program will intercept the service and do nothing if A or
Z is to be printed rest of the characters will be printed normally. Only the As and the Zs in
the printing document will be omitted.

Sample Program

```c
#include <dos.h>
void interrupt (*old)( );
void interrupt newfunc ( );
main( )
{
    old=getvect(0x17);
    setvect(0x17,newfunc);
    keep(0,1000);
}
void interrupt newfunc( )
{
    if (_AH==0)
    {
        if ( _AL != '  ' )
            (*old);
    }
}
```

In this sample program again int 17H is intercepted. The new interrupt function will
ignore all the spaces in the print document.
Sample Program

```c
#include <dos.h>
void interrupt (*old)();
void interrupt newfunc();
main()
{
    old=getvect(0x17);
    setvect(0x17,newfunc);
    keep(0,1000);
}
void interrupt newfunc()
{
    if (_AH == 0 ) {
        (*old);
        _AH=0;
        (*old);
        _AH=0;
        (*old);
    }
    (*old);
}
```

In this program interrupt 17h is again intercepted. Whenever a character is to printed the new function call the old function thrice. As a result a single character in the print document will be repeated 4 times.

Direct Parallel Port Programming

Now we will see how the register within the PPI can be accessed directly to control the printer.
Direct Parallel Port Programming

- BIOS support up to three parallel ports
- Address of these LPT ports is stored in BIOS Data Area

<table>
<thead>
<tr>
<th>40:08</th>
<th>word</th>
<th>LPT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>40:0A</td>
<td>word</td>
<td>LPT2</td>
</tr>
<tr>
<td>40:0C</td>
<td>word</td>
<td>LPT3</td>
</tr>
<tr>
<td>40:0E</td>
<td>word</td>
<td>LPT4</td>
</tr>
</tbody>
</table>

Above slide list the addresses within the BIOS data area where the base address (starting port number) of LPT devices is stored.

Dump of BIOS data area

The dump of BIOS data area address specified in the previous slide for a certain computer shows that the base port address of LPT1 is 0x03bc, for LPT2 it is 0x0378, for LPT3 it is 0x0278. These values need not be the same for all the computer and can vary from computer to computer.
Swapping LPTs

Direct Parallel Port Programming

```c
unsigned int far * lpt =
    (unsigned int far *) 0x00400008 ;
unsigned int temp;
    temp=*(lpt);
    *lpt=*(lpt + 1);
    *(lpt + 1)=temp;
```

The LPTs can be swapped i.e LPT1 can be made LPT2 and vice versa for LPT2. This can be accomplished simply by swapping their addresses in the BIOS data area as shown in the slide above.

Direct Parallel Port Programming

Port Registers

- 40:08 store the base address for lpt1
- The parallel port interface has 3 ports internally
- If the Base address is 0X378 then the three Ports will be 0x378,0x379 0x37A

© Copyright Virtual University of Pakistan
LPT Ports

Direct Parallel Port Programming

Port Registers

Base +0 = Data Port

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Base +1 = Printer Status

| Busy=0 | ACK=0 | PE=1 | SL=1 | ERR=0 | 0 | 0 | 0 |

- **Printer Online**
- **Out of Paper**
- **Printer is ready for Next Character**
- **Printer is Busy**

The first port (Base +0) is the data port. Data to be sent/received is placed in this port. In case of printer the (Base + 1) is the printer status port as described in the slide. Each bit represents the various status of the printer quite similar to the status byte in case of BIOS service.

Printer Control Register

Direct Parallel Port Programming

Port Registers

Printer Control Register = Base + 2

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IRQ</td>
<td>SI</td>
<td>IN</td>
<td>ALF</td>
<td>ST</td>
</tr>
</tbody>
</table>

- **IRQ**: Execute Interrupt When ACK=0;
- **SELECT InLine**
- **Turn Computer on line**
- **initialize**
- **Auto Line Field**
- **STROB**

(Base +2) is the printer control register it is used to pass on some control information to the printer as described in the slide.
Direct Parallel Port Programming

```c
file *fptr;
unsigned far *base=(unsigned int far *)0x00400008
void main (void)
{
    fptr=fopen("c:\abc.txt","rb");
    while( ! feof (fptr) )
        {  
            if( ! ( inport (*base + 1 ) & 0x80)  
            {  
                outport(*base,getc(fptr));
                outport ((*base+2,&inport(*base+2) | 0x01);
                outport((base+2,inport(*base+2) & 0xFE);
            }  
        }
}
```

The above program directly accesses the registers of the PPI to print a file. The while loop terminates when the file ends. The if statement only checks if the printer is busy or not. If the printer is idle the program writes the next byte in file on to the data port and then turns the strobe bit to 1 and then 0 to indicate that a byte has been sent to the printer. The loop then again starts checking the busy status of the printer and the process continue.
Lecture # 12

Printer Interface and IRQ7

The printer interface uses the IRQ 7 as shown in the slide above. Therefore if interrupt driven I/O is to be performed int 0x0F need to be programmed as an hardware interrupt.

```
Interrupt Driven Printer I/O
char buf[1024]; int i = 0;
void interrupt (*oldint)(
void interrupt newint ();
void main (void)
{
outport(( *lpt), inport( *lpt) | 4);
outport(( *lpt), inport( *lpt) | 0x10);
oldint =getvect (0x0F);
setvect (0x0F, newint);
outport(0x21, inport( 0x21) & 0x7F);//corrected
keep(0,1000);
}
```
Above is a listing of a program that uses int 0x0f to perform interrupt driven I/O. To enable the interrupt 0x0f three things are required to be done. The interrupt should be enabled in the printer control register, secondly it should also be unmasked in the IMR in PIC. The program can then intercept or set the vector of interrupt 0x0f by placing the address of its function newint();

The newint() will now be called whenever the printer can perform output. This newint() function writes the next byte in buffer to the data registers and then send a pulse on the strobe signal to tell the printer that data has been sent to it. When whole of the buffer has been sent the int 0x0f vector is restored, interrupt is masked and the memory for the program is de-allocated.

The above listing might not work. Not all of the printer interfaces are designed as described above. Some modifications in the printer interface will not allow the interrupt driven I/O to work in this manner. If this does not work the following strategy can be adopted to send printing to the printer in background.
#include <stdio.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <stdlib.h>
void interrupt (*oldint());
void interrupt newint();
unsigned int far * lpt = (unsigned int far *)0x00400008;
char st[80] = "this is a test print string !!!!!!!!!!!."
int i;

void main ()
{
    oldint = getvect(0x08);
    setvect(0x08,newint);
    keep(0,1000);
}

void interrupt newint()
{
    if (((inport(*lpt)+1)) & 0x80) == 0x80)
    {
        outport(*lpt,st[i++]);
        outport((*lpt)+2, inport((*lpt)+2) & 0xfe);
        outport((*lpt)+2, inport((*lpt)+2) | 1);
    }
    if (i==32)
    {
        setvect (0x08,oldint);
        freemem(_psp);
    }
    (*oldint)();
}

This program uses the timer interrupt to send printing to the printer in the back ground. Whenever the timer interrupt occurs the interrupt function checks if the printer is idle or not. If it’s the printer is idle it takes a byte from the buffer and sends it to the data port of the printer interface and then sends a pulse through the strobe signal. When the buffer is full the program restores the int 8 vector and the relinquishes the memory occupied by the program.
Not all the bits of the internal registers of the PPI are available in standard PCs. In standard PCs the PPI is connected to a DB25 connector. And some of the bits of its internal registers are available as pin outs as describes in the slide above.
Computer to Computer communication

It might be desirable to connect one computer to another via PPIs to transfer data. One might desire to connect them such that one port of PPI at one end is connected to another port of the other PPI at the other end. But interconnecting the whole 8 bits of PPI cannot be made possible as all the bits of the internal ports are not available as pinouts. So the answer is to connect a nibble (4-bits) at one end to the nibble at the other. In this way two way communication can be performed. The nibbles are connected as shown in the slide above.
PPI Interconnection

The pins that are interconnected are shown in the slide above. Another thing worth
noticing is that the 4th bit of the data port is connected to the BUSY and vice versa. The
BUSY is inverted before it can be read from the status port. So the 4th bit in data port at
PC1 will be inverted before it can be read at the 7th bit of status register at PC2.

Flow Control

An algorithm should be devised to control the flow of data so the receiver and sender may
know when the data is to be received and when it is to be sent. The following slides
illustrate one such algorithm.
First the low nibble of the byte is sent from the sender in bit D0 to D3 of the data port. D4 bit is cleared to indicate the low nibble is being sent. The receiver will know the arrival of the low nibble when its checks BUSY bit which should be set (by the interface) on arrival.

The receiver then sends back the nibble turning its D4 bit to 0 as an acknowledgement of the receipt of the low nibble. This will turn the BUSY bit to 1 at the sender side.

The sender then send the high nibble and turns its D4 bit to 1 indicating the transmission of high nibble. On the receiver side the BUSY bit will turn to 0 indicating the receipt of high nibble.
The receiver then sends back the high nibble to the sender as an acknowledgment.
Lecture # 13

Program implementing the described protocol

```c
int i = 0; char Buf[1024];
while (1)
{
    ch = Buf[i];
    if ((inport(*lpt) + 1) & 0x80) == 0)
    {
        ch = Buf[i];
        ch = ch & 0xEF;
        while((inport(*lpt) + 1) &0x80) == 0);
    }
    else
    {
        ch = Buf[i];
        ch = ch >> 4;
        ch = ch | 0x10;
        outport(*lpt, ch);
        i++;
        while((inport(*lpt) + 1) &0x80) == 80);
    }
}
```

This is the sender program. This program if find the BUSY bit clear sends the low nibble but turns the D4 bit to 0 before sending. Similarly it right shifts the byte 4 times sets the D4 bit and then sends the high nibble and waits for acknowledgment until the BUSY is cleared.

```c
int i;
while (1)
{
    if ((inport(*lpt + 1)& 0x80) == 0x80)
    {
        x = inport ((*lpt) + 1);
        x = x >> 3;
        x = x & 0x0F;
        outport(*lpt, x);
        while((inport(*lpt) + 1) &0x80) == 0x80);
    }
    else
    {
        y = inport ((*lpt) + 1);
        y = y << 1;
temp = y;
        y = y & 0xF0; //instruction added
        y = y | x;
    }
```
temp = temp >> 3;
temp = temp | 0x10;
i++;
outport (*lpt, temp);
Buf [i] = y;
while((inport((*lpt) + 1) & 0x80) == 0);
}

This is receiver program. If the BUSY bit is clear it receives the low nibble and stores it in x. Similarly if the BUSY bit is 0 it receives the high nibble and concatenates the botth nibble to form a byte.
Serial Communication

- Advantages
- Disadvantages

**Types Of Serial Communication**

- Synchronous
- Asynchronous

In case of serial communication the bits travel one after the other in serial pattern. The advantage of this technique is that in this case the cost is reduced as only 1 or 2 lines maybe required to transfer data.

The major disadvantage of Serial communication is that the speed of data transfer maybe reduced as data is transferred in serial pattern.

There are two kinds of serial communications.

**Synchronous Communication**

- Timing signal is used to identify start and end of a bit.

In case of synchronous communication as shown in the slide a timing signal is required to identify the start and end of a bit.
Synchronous Communication

- Sampling may be edge triggered.
- Special line may be required for timing signal (requires another line).
- Or the timing signal may be encoded within the original signal (requires double the bandwidth).

Asynchronous Communication

- Does not use make use of timing signal.
- Each byte (word) needs to be encapsulated in start and end bit.

In case of asynchronous communication no timing signal is required but each byte needs to be encapsulated to know the end and start of a byte.
The UART is a device used for asynchronous communications. UART is capable of encapsulating a byte that might be 5, 6, 7 or 8 bits wide in start and stop bits. Moreover, it can attach an extra parity bit with the data for error detection. The width of stop bits may also vary.

**Sampling Rate**

Bit rate = 9600

A bit is sampled after $= \frac{1}{9600}$

-- But start and end bits of a particular Byte cannot be recognized.

-- So 1.5 stop bit (high) is used to encapsulate a byte. A low start bit at the start of Byte is used to identify the start of a Byte.
--- Bit rate and other settings should be the same at both ends i.e.
- Data bits per Byte. (5 – 8)
- Parity check
- Parity Even/Odd
- No. of stop bits.

Sampling Rate

A = 41H = 0100 0001 B
Parity = Odd
Data = 8
Stop bit =1
Data rate = 300 bits/sec
RS232C is a standard for physical dimension of the connector interconnecting a DTE (Data terminal equipment) and DCE (Data communication equipment).

The pin outs of the DB25 connector used with RS232C is shown in the slide above.
Lecture # 14

RS – 232C Standard
- Standard for physical dimensions of the connectors.

PC (DTE)

Modem (DCE)
Connected via serial port

RS – 232C Connectors and Signals
DB25 (25 pin connector)

- 13
- 12
- 11
- 10
- 9
- CD
- GND
- DSR
- CTS
- RTS
- RD
- TXD
- RXD
- 25
- 24
- 23
- 22
- RI
- 21
- DTR
- 20
- 19
- 18
- 17
- 16
- 15
- 14

25 pin connector on PC
Flow Control using RS232C

Data is received through the RxD line. Data is send through the TxD line. DTR (data terminal ready) indicates that the data terminal is live and kicking. DSR (data set ready) indicates that the data set is live. Whenever the sender can send data it sends the signal RTS (Request to send) if as a result the receiver is free and can receive data it send the sender an acknowledge through CTS (clear to send) indicating that its clear to send now.

DB 9 Connector for UART

The above slide shows the pinouts of the DB 9 connector.
UART internals

This slide shows the various internal registers within a UART device. The programmer only needs to program these registers efficiently in order to perform asynchronous communication.

Register summary

<table>
<thead>
<tr>
<th>Base +</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Holding Register</td>
</tr>
<tr>
<td>Receiver Data</td>
</tr>
<tr>
<td>Band Rate Divisor (Low Byte)</td>
</tr>
<tr>
<td>Band Rate Divisor (High Byte)</td>
</tr>
<tr>
<td>Interrupt Enable</td>
</tr>
<tr>
<td>FIFO Control Register</td>
</tr>
<tr>
<td>Interrupt ID</td>
</tr>
<tr>
<td>Line Control</td>
</tr>
<tr>
<td>Mode Control</td>
</tr>
<tr>
<td>Line Status</td>
</tr>
<tr>
<td>Modern Status</td>
</tr>
<tr>
<td>Scratch Pad</td>
</tr>
</tbody>
</table>

The above table lists the registers within the UART and also shows their abbreviation. Also it shows there offsets with respect to the base register.
## Served Ports in Standard PC

BIOS supports 4 UARTS as COM Ports
COM1, COM2, COM3, COM4

<table>
<thead>
<tr>
<th>Ports</th>
<th>Memory Address</th>
<th>Port Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM1</td>
<td>0040:0000</td>
<td>03F8H</td>
</tr>
<tr>
<td>COM2</td>
<td>0040:0002</td>
<td>02F8H</td>
</tr>
<tr>
<td>COM3</td>
<td>0040:0004</td>
<td>03E8H</td>
</tr>
<tr>
<td>COM4</td>
<td>0040:0006</td>
<td>02E8H</td>
</tr>
</tbody>
</table>

## BIOS Data Area

The above dump of the BIOS data area for a certain computer shows that the address of COM1 is 03F8, the address of COM2 is 02F8 and the address of COM3 is 03E8. These addresses may not be same for all the computers and may vary computer to computer.
Setting the Baud rate

Setting the Baud Rate

1.8432 MHZ=frequency generating by UARTS internally

\[ \text{Baud rate} = \frac{1.8432 \text{ MHZ}}{16 \times \text{Divisor}} \]

Divisor value loaded in DLL (Base +0) and DLM (Base +1)

Divisor = 1, Baud Rate = 115200
Divisor = 0CH, Baud Rate = 9600
Divisor = 180H, Baud Rate = 300

The baud rate is set in accordance with the divisor value loaded within the UART internal registers base +0 and base +1.

Line Control Register

The line control register contains important information about the behaviour of the line through which the data will be transferred. In it various bits signify the word size, length of stop bits, parity check, parity type and also the a control bit to load the divisor value. The bit 7 if set indicates that the base +0 and base +1 will act as the divisor register otherwise if cleared will indicate that base +0 is the data register.
Line status register illustrates the status of the line. It indicates if the data can be sent or received. If bit 5 and 6 both are set then 2 consecutive bytes can be sent for output. Also this register indicates any error that might occur during communication.

Interrupt Enable Register

If interrupt driven output is to be performed then this register is used to enable interrupt for the UART. It can also used to select the events for which to generate interrupt as described in the slide.
Interrupt ID Register

Once an interrupt occurs it may be required to identify the case of the interrupt. This register is used to identify the cause of the interrupt.

Modem/Line
00 = Change in Modem Status
01 = THR is Empty
10 = Data is Ready
11 = Error in Data
Lecture # 15

Modem Control Register

In case software oriented flow control technique is used the bits 0 and 1 need to be set in that case. Bit #3 need to be set to enable interrupts. Moreover if a single computer is available to a developer the UART contains a self test mode which can be used by the programmer to self test the software. In self test mode the output of the UART is routed to its input. So you receive what you send.
Modem Status Register

This register indicates the status of the modem status line or any change in the status of these lines.

FIFO Queue

This feature is available in the newer version of the UART numbered 16500. A queue or a buffer of the input or output bytes is maintained within the UART in order to facilitate more efficient I/O. The size of the queue can be controlled through this register as shown by the slide.
Interrupt ID Register

### Interrupt ID Register (Revisited)

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Interrupt Triggered =1**
  - Any one of these BEING Set Indicates FIFO is ON.
  - 1= Interrupt Triggered
    - Because Buffer is not full
    - But other side has stop sending data.
    - (Time OUT)

- **Reasons of Interrupt**
  - 00=Change in Modem Line Status
  - 01=THR is Empty
  - 10=Data is ready
  - 11=Error in Data Transmit

---

BIOS Support for COM Ports

### INT # 14H

- **DX = Port # 0 for COM1**
  - 1 for COM2 etc.

---

**Service #0 = Set communication parameters**
**Service #01 = Output characters**
**Service #02 = Read in characters**
**Service #03 = Get port status**

---

The following slide shows how int 14H service 0 can be used to set the line parameter of the UART or COM port. The illustrates the various bits of AL that should be set according before calling this service.
The Service on return places the line status in AH register as shown in the slide below.

And places the modem status in the AL register as shown in slide below.
Other service of 14h include service #1 which is used to send a byte and service #2 which is used to receive a byte as shown in the slide below.

**Service # 01**

**ON ENTRY**
- AL = ASCII character to send

**ON RETURN**
- AH = Error Code
  - If 7th bit in AH = 1 = Unsuccessful
  - 0 = Successful

**Service # 02**

**ON RETURN**
- AL = ASCII character received
- AH = Error Code
Communication through Modem

Modem is generally used to send/receive data to/from an analog telephone. Had the telephone line been purely digital there would have been no need of a modem in this form. If data is to be transferred from one computer to another through some media which can carry digital data then the modem can be eliminated and the UART on both computers can be interconnected. Such arrangement is called a NULL modem.
NULL Modem Configuration

The above slide shows the configuration used to interconnect two UARTs. In this way a full duplex communication can be performed and moreover flow control can also be performed using DSR, DTS, RTS and CTS signals.

Sample Program

Example:
```
#include<BIOS.H>
#include<DOS.H>
char ch1, ch2;

void initialize (int pno)
{
    _AH=0;
    _AL=0x57;
    _DX=pno;
    geninterrupt(0x14);
}
```
char receivechar (int pno)
{
    char ch;
    _DX = pno;
    _AH = 2;
    geninterrupt (0x14);
    ch = _AL;
    return ch;
}

void sendchar (char ch, int pno)
{
    _DX = pno;
    _AH = 1;
    _AL = ch;
    geninterrupt (0x14);
}

unsigned int getcomstatus (int pno)
{
    unsigned int temp;
    _DX = pno;
    _AH = 03;
    geninterrupt (0x14);
    *((char*)(&temp)) = _AL;
    *((char*)(&temp)) + 1 = _AH;
    return temp;
}
Lecture # 16
Sample Program using BIOS routines

Example:
```
#include<BIOS.H>
#include<DOS.H>
char ch1, ch2;

void initialize (int pno) {
    _AH=0;
    _AL=0x57;
    _DX=pno;
    geninterrupt(0x14);
}
```

```
char receivechar (int pno) {
    char ch;
    _DX = pno;
    _AH = 2;
    geninterrupt (0x14);
    ch = _AL;
    return ch;
}
```

The initialize () function initializes the COM port whose number is passed as parameter using BIOS services. The recievechar() function uses the COM port number to receive a byte from the COM port using BIOS services.
the sendchar() function sends a character to the COM port using BIOS service whose number is passed as parameter. And the getcomstatus() function retrieves the status of the COM port whose number has been specified and returns the modem and line status in an unsigned int.

Let’s suppose two UARTs are interconnected using a NULL modem
In the main() function there is a while loop which retrieves the status of the COM port. Once the status has been retrieved it checks if a byte can be transmitted, if a key has been pressed and its is clear to send a byte the code within the if statement sends the input byte to the COM port using sendchar() function.
The second if statement checks if a byte can be read from the COM port. If the Data ready bit is set then it receives a byte from the data port and displays it on the screen. Moreover there is another check to end the program. The program looks for an escape character ASCII = 27 either in input or in output. If this is the case then it simply breaks the loop.

Sample Program

This program does more or less the same as the previous program but the only difference is that in this case the I/O is done directly using the ports and also that the Self Test facility is used to check the software.

```c
#include <dos.h>
#include <bios.h>
void initialize (unsigned int far *com)
{
    outportb ( (*com)+3, inport ((*com)+3) | 0x80);
    outportb ( (*com),0x80);
    outportb( (*com) +1, 0x01);
    outportb ( (*com)+3, 0x1b);
}
void SelfTestOn(unsigned int far * com)
{
    outportb((*com)+4,inport((*com)+4)|0x10);
}
```

The initialize() loads the divisor value of 0x0180 high byte in base +1 and low byte in base +0. It also program the line control register for all the required line parameters. The SelfTestOn() function simply enables the self test facility within the modem control register.
The SelfTestOff() function turns this facility off. The writechar() function writes the a byte passed to this function on the data port. The readchar() function reads a byte from the data port.

The main function after initializing and turning the self test mode on enters a loop which will terminate on input of the escape character. This loop also controls the position of the cursor such the cursor goes to the next line right after a full line has been typed.
```cpp
if (j==13)
    j=0;

gotoxy(i,j);
ch=getche();
writechar(ch,com);
ch2=readchar(com);
gotoxy(i,j+14);
putch(ch2);
i++;
}
SelfTestOff (com);
}
```

All the input from the keyboard is directed to the output of the UART and all the input from the UART is also directed to the lower part of the screen. As the UART is in self test mode the output becomes the input. And hence the user can see output send to the UART in the lower part of the screen as shown in the slide below.

```plaintext
hello how r u ? whats new about systems programming?

hello how r u ? whats new about systems programming?
```
Sample Program using interrupt driven I/O

```c
#include <dos.h>
#include <bios.h>

void initialize (unsigned int far *com)
{
    outportb ( (*com)+3, inport ((*com)+3) | 0x80);
    outportb ( (*com),0x80);
    outportb( (*com) +1, 0x01);
    outportb ( (*com)+3, 0x1b);
}

void SelfTestOn(unsigned int far * com)
{
    outportb((*com)+4,inport((*com)+4)|0x18);
}

void SelfTestOff(unsigned int far * com)
{
    outportb((*com)+4, inport((*com)+4) & 0xE7);
}

void writechar( char ch, unsigned int far * com)
{
    //while ( !((inportb((*com)+5) & 0x20) == 0x20));
    outport(*com,ch);
}

char readchar( unsigned int far *com)
{
    //while (!((inportb((*com)+5) & 0x01)==0x01));
    return inportb(*com);
}
```
unsigned int far *com=(unsigned int far*)0x00400000;
unsigned char far *scr=(unsigned char far*)0xB8000000;
int i =0,j=0;char ch;int k;
void interrupt (*oldint)();
void interrupt newint();
void main ()
{
    initialize(com);
    SelfTestOn(com);
    oldint = getvect(0x0c);
    setvect(0x0c,newint);
    outport(("com")+1,1);
    outport(0x21,inport(0x21)&0xEF);
    keep(0,1000);
}

This si program is also quite similar to the previous one. The only difference is that in this
the I/O is performed in an interrupt driven patter using the Int 0x0C as the COM1 uses
IRQ4. Also to use it in this way IRQ4 must be unmasked from the IMR register in PIC.
Also before returning from the ISR the PIC must be signaled an EOI code.

void interrupt newint()
{
    ch= readchar(com);
    if (i==80)
    {
        j++;
        i=0;
    }
    if (j==13)
    j=0;
    k = i*2+(j+14)*80*2;
    *(scr+k)=ch;
    i++;
    outport(0x20,0x20);
}
```c
#include <bios.h>
#include <dos.h>

void interrupt (*oldint)();
void interrupt newint();
unsigned char far *scr = (unsigned char far*)0xB8000000;

void initialize (unsigned int far *com)
{
    outportb (*com)+3, inport ((*com)+3) | 0x80);
    outportb (*com),0x80);
    outportb(*com)+1, 0x01);
    outportb (*com)+3, 0x1b);
}

void main (void)
{
    oldint = getvect(0x0C);
    vect ((0xC,newint);
    initialize (*com);
    outport ((*com)+4, inport ((com)+4) | 0x08);
    outport ((0x21, inport (0x21)&0x0EF);
    outport ((com) + 1, 1);
    keep(0,0x60);
}

void interrupt newint ()
{
    *scr = inport(*com);
    outport (0x20,0x20);
}
```
Lecture # 17
Sample Program

```
#include <dos.h>
#include <bios.h>
char ch1,ch2;

void initialize (unsigned int com)
{
    outportb ( com+3, inport (com+3) | 0x80);
    outportb ( com,0x80);
    outportb( com +1, 0x01);
    outportb ( com+3, 0x1b);
    outportb ( com+3, 0x1b);
}

void main ( )
{
    initialize(0x3f8);
    while (1)
    {
        if ( ((inport(0x3fd)&0x20)==0x20) && (kbhit()))
            ch1=getche();
            outport(0x3f8,ch1);
        }
    if ( ((inport(0x3fd)&0x01)==1))
        ch2= inport(0x3f8);
        putch(ch2);
    }
    if (( ch1==27) || (ch2==27))
        break;
}
```

This program is same functionally as one of the previous programs which used BIOS services to get the input data and send the output data. The only difference is that in this case it does the same directly accessing the ports.
NULL Modem (Revisited)

Only two or three of the lines are being used to send receive data rest of the lines are being used for flow control. The cost of these lines can be reduced by reducing the lines used to flow control and incorporating software oriented flow control rather than hardware oriented flow control as show in the slide below.

NULL Modem (Revisited)

The DTR, DSR, RTS and CTS lines have been eliminated to reduce cost but in this flow control will be performed in a software oriented manner.
Software Oriented Flow Control

Makes use of Two Control characters.
- XON (^S)
- XOFF (^T)

XON whenever received indicates the start of communication and XOFF whenever received indicates a temporary pause in the communication.

Following is a pseudo code which can be used to implement the software oriented flow control.

```
while (1)
{
    receivedchar = readchar (com);
    if (receivedchar == XON)
    {
        ReadStatus = TRUE;
        continue;
    }
    if (receivedchar == XOFF)
    {
        ReadStatus = FALSE;
        continue;
    }
    if (ReadStatus == TRUE)
    Buf [i++] = receivedchar;
}
```

the received character is firstly analysed for XON or XOFF character. If XON is received the status is set to TRUE and if XOFF is received the status is set to FALSE. The characters will only be received if the status is TRUE otherwise they will be discarded.
Real Time Clock

<table>
<thead>
<tr>
<th>Time Updation Through INT8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Real Time Clock Device</strong></td>
</tr>
<tr>
<td>• Battery powered device</td>
</tr>
<tr>
<td>• Updates time even if PC is shutdown</td>
</tr>
<tr>
<td>• RTC has 64 byte battery powered RAM</td>
</tr>
<tr>
<td>• INT 1AH used to get/set time.</td>
</tr>
</tbody>
</table>

Real time clock is a device incorporated into the PC to update time even if the computer is off. It has the characteristics shown in the slide above which enables it to update time even if the computer is off.

The BIOS interrupt 0x1Ah can be used to configure this clock as shown in the slide below it has various service for getting/setting time/date and alarm.

<table>
<thead>
<tr>
<th>Clock Counter 1AH/00</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hours<em>60</em>60 + min*60 + sec) * 18.2065</td>
</tr>
</tbody>
</table>

**ON ENTRY**
AH = 00

**ON EXIT**
AL = Midnight flag
CX = Clock count (Hi word)
DX = Clock count (Low word)

1573040 Times Increment
1573040/18.2065 = 86399.9121 sec
Whereas 86400 sec represent 24 hrs.
ON ENTRY
AH = 01
CX = Clock count (Hi word)
DX = Clock count (Low word)

AL = 1  if Midnight passed
AL = 0  if Midnight not passed
Set Clock Counter 1AH/01

Read Time 1AH/02

ON ENTRY
AH = 02

ON EXIT
CH = Hours (BCD)
CL = Minutes (BCD)
DH = Seconds (BCD)
Set Time 1AH/03

ON ENTRY

AH = 03
CH = Hours (BCD)
CL = Minutes (BCD)
DH = Seconds (BCD)
DL = Day Light saving = 1
Standard Time = 0

Read Date 1AH/04

ON ENTRY

AH = 04

ON EXIT

CH = Century (BCD)
CL = Year (BCD)
DH = Month (BCD)
DL = Day (BCD)
Set Date 1AH/05

ON ENTRY
AH = 05
CH = Century (BCD)
CL = Year (BCD)
DH = Month (BCD)
DL = Day (BCD)

Set Alarm 1AH/06

ON ENTRY
AH = 06
CH = Hours (BCD)
CL = Minutes (BCD)
DH = Seconds (BCD)
Disable Alarm  1AH/07

ON ENTRY
AH = 07

Read Alarm  1AH/09

ON ENTRY
AH = 09

ON EXIT
CH = Hours (BCD)
CL = Minutes (BCD)
DH = Seconds (BCD)
DL = Alarm Status (00 = Not Enable
     01 = Enable)
RTC internals

The RTC internally has an array of registers which can be used to access the 64 byte battery powered CMOS RAM.

Internal Ports
70 – 7FH (16 ports)
Only 70 & 71H are important from programming point of view
The following slide shows the function of some of the bytes in the battery powered RAM used to store the units of time and date.

<table>
<thead>
<tr>
<th>64 Byte Battery Powered RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H = Current Second</td>
</tr>
<tr>
<td>01H = Alarm Second</td>
</tr>
<tr>
<td>02H = Current Minute</td>
</tr>
<tr>
<td>03H = Alarm Minute</td>
</tr>
<tr>
<td>04H = Current Hour</td>
</tr>
<tr>
<td>05H = Alarm Hour</td>
</tr>
<tr>
<td>06H = Day Of the Week</td>
</tr>
<tr>
<td>07H = Number Of Day</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64 Byte Battery Powered RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>08H = Month</td>
</tr>
<tr>
<td>09H = Year</td>
</tr>
<tr>
<td>0AH = Clock Status Register A</td>
</tr>
<tr>
<td>0BH = Clock Status Register B</td>
</tr>
<tr>
<td>0CH = Clock Status Register C</td>
</tr>
<tr>
<td>0DH = Clock Status Register D</td>
</tr>
<tr>
<td>32H = Century</td>
</tr>
</tbody>
</table>
Day of the week

<table>
<thead>
<tr>
<th>Week Day</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01H = Sunday</td>
<td></td>
</tr>
<tr>
<td>02H = Monday</td>
<td></td>
</tr>
<tr>
<td>03H = Tuesday</td>
<td></td>
</tr>
<tr>
<td>04H = Wednesday</td>
<td></td>
</tr>
<tr>
<td>05H = Thursday</td>
<td></td>
</tr>
<tr>
<td>06H = Friday</td>
<td></td>
</tr>
<tr>
<td>07H = Saturday</td>
<td></td>
</tr>
</tbody>
</table>

The value in the days of the week byte indicates the day according to slide shown above.

Generally BCD values are used to represent the units of time and date.

<table>
<thead>
<tr>
<th>Year</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No of Century and Year are in BCD.</td>
<td></td>
</tr>
</tbody>
</table>
Accessing the Battery Powered RAM

Battery Powered RAM is accessed in two steps
• Specify the Byte no. in 70H port.
• Read/write port 71H to get/set the value of specified byte.

Following slide shown a fragment of code that can be used to read or write onto any byte within the 64 byte battery powered RAM.

```
outport (0x70, 0);             outport (0x70, 4);
sec = inport (0x71);          outport (0x71,hrs);
```
Clock Status Registers

The lower 4 bits of this register stores a code indicating the frequency with which the RTC hardware interrupt can interrupt the processor. The next field is used to specify the time frequency i.e. the frequency with the time is sampled and hence updated. The most significant bit indicates that after time sampling if the time has been updated in to the 64 byte RAM or not.
Lecture # 18

Clock Status Registers

The status register B is the main control register. It is used to specify the date time formats and is also used to enable interrupt on various events like alarm time and time updation. Another feature of RTC is periodic interrupt which occur with a frequency specified in the A register.

Status register is used to identify the reason of interrupt generation as described in the slide above.
Only the most significant byte in status register D is important which on being 0 indicates that the battery is dead.

Sample Program.

```c
void main()
{
    unsigned int hours, months, seconds;
    _AH = 2;
    geninterrupt(0x1a);
    hours = _CH;
    minutes = _CL;
    seconds = _DH;
    hours = hours << 4;
    *((unsigned char *)(& hours)) = *((unsigned char *)(& hours)) >> 4;
    hours = hours + 0x3030;
    seconds = seconds << 4;
    *((unsigned char *)(& seconds)) = *((unsigned char *)(& seconds)) >> 4;
    seconds = seconds + 0x3030;
}```
The above program uses the service int 1Ah/02H to read the time from the real time clock. It reads the time and converts the packed BCD values into unpacked BCD values. These values are then converted into ASCII and displayed using the printf() statement.
Read time from RTC (Sample Program)

This sample program directly accesses the 64 byte RAM to access the units of time. Before reading the time it makes sure by checking the value of Status register A and checking its most significant bit for time update completion. If the updation is complete time can be read from the respective registers in the 64 byte RAM.

```c
#include <bios.h>
#include <dos.h>
void main ()
{
    int hrs,mins,secs;
    char temp;
    do {
        outportb(0x70,0x0a);
        temp=inportb(0x71);
    }while ((temp & 0x80) == 0);
    outportb(0x70,0);
    secs=inport(0x71);
    outportb(0x70,2);
    mins=inport(0x71);
    outportb(0x70,4);
    hrs=inport(0x71);

    hrs = hrs <<4;
    *((unsigned char *)(&hrs)) =
    *((unsigned char *)(&hrs)) >>4;
    hrs = hrs + 0x3030;

    mins = mins <<4;
    *((unsigned char *)(&mins)) =
    *((unsigned char *)(&mins)) >>4;
    mins = mins + 0x3030;

    secs = secs <<4;
    *((unsigned char *)(&secs)) =
    *((unsigned char *)(&secs)) >>4;
    secs = secs + 0x3030;
    clrscr();
```
printf("%c%c:%c%c:%c%c",
    *((unsigned char*)(&hrs))+1),
    *((unsigned char*)(&hrs)),
    *((unsigned char*)(&mins))+1),
    *((unsigned char*)(&mins)),
    *((unsigned char*)(&secs))+1),
    *((unsigned char*)(&secs)));
getch();
}

The time units are similarly read and converted to ASCII and displayed.

**Write the Time on RTC**

```c
#include <bios.h>
#include <dos.h>

unsigned char ASCIItoBCD(char hi, char lo) {
    hi = hi - 0x30;
    lo = lo - 0x30;
    hi = hi << 4;
    hi = hi | lo;
    return hi;
}

unsigned long int far *tm = 
    (unsigned long int far *)0x0040006c;
```
The above listing of the program inputs the time from the user which is in ASCII format. It converts the ASCII in packed BCD and uses BIOS services to update the time. In DOS or Windows this time change may not remain effective after the completion of the program as the DOS or Windows device drivers will revert the time to original even if it has been changed using this method.
Sample Program

```c
#include <bios.h>
#include <dos.h>

unsigned char ASCIItoBCD (unsigned char hi, unsigned char lo)
{
    hi = hi - 0x30;
    lo = lo - 0x30;
    hi = hi << 4;
    hi = hi | lo;
    return hi;
}

void main()
{
    unsigned int hrs, mins, secs;
    char ch1, ch2;
    int temp;

    puts("Enter the hours to update: ");
    ch1 = getche();
    ch2 = getche();
    hrs = ASCIItoBCD(ch1, ch2);

    puts("Enter the minutes to update: ");
    ch1 = getche();
    ch2 = getche();
    mins = ASCIItoBCD(ch1, ch2);

    puts("Enter the seconds to update: ");
    ch1 = getche();
    ch2 = getche();
    secs = ASCIItoBCD(ch1, ch2);

    outportb(0x70, 0x0b);
    temp = inport(0x71);
```
To elaborate more on the problem posed by the OS device drivers here is another program. This program first updates the time accessing the 64 byte RAM directly and taking the new time as input from the user. After updating the program produces a delay of 30 seconds and then reads time to display it. A difference of 30 seconds will be noticed in the time entered and the time displayed. This shows that during the execution of the program the time was successfully changed and was being updated accordingly.
mins = mins <<4;
*(unsigned char *)&mins = 
  (*((unsigned char *)&mins)) >>4;
mins = mins + 0x3030;
secs = secs <<4;
*(unsigned char *)&secs = 
  (*((unsigned char *)&secs)) >>4;
secs = secs + 0x3030;
printf("Updated time is = %c%c:%c%c:%c%c",
  (*((unsigned char *)&hrs)+1),
  *((unsigned char *)&hrs),
  *((unsigned char *)&mins),
  (*((unsigned char *)&mins)+1),
  *((unsigned char *)&secs),
  *((unsigned char *)&secs));
getch();
#include <bios.h>
#include <dos.h>
void main ()
{
    unsigned int cen, yrs, mons, days;
    _AH = 4;
    geninterrupt(0x1a);
    cen = _CH;
    yrs = _CL;
    mons = _DH;
    days = _DL;
    cen = cen << 4;
    *(unsigned char *)&cen = *(unsigned char *)&cen >> 4;
    cen = cen + 0x3030;
    mons = mons << 4;
    *(unsigned char *)&mons = *(unsigned char *)&mons >> 4;
    mons = mons + 0x3030;
    yrs = yrs << 4;
    *(unsigned char *)&yrs = *(unsigned char *)&yrs >> 4;
    yrs = yrs + 0x3030;
    days = days << 4;
    *(unsigned char *)&days = *(unsigned char *)&days >> 4;
    days = days + 0x3030;
    clrscr();
}
Setting the Date

unsigned char ASCIItoBCD(char hi, char lo)
{
    hi = hi - 0x30;
    lo = lo - 0x30;
    hi = hi << 4;
    hi = hi | lo;
    return hi;
}

void main()
{
    unsigned char yrs,mons,days,cen;
    char ch1, ch2;
    puts("Enter the century to update: ");
    ch1=getche();
    ch2=getche();
    cen = ASCIItoBCD(ch1, ch2);
    puts("Enter the yrs to update: ");
    ch1=getche();
    ch2=getche();
    yrs = ASCIItoBCD(ch1, ch2);
    puts("Enter the month to update: ");
    ch1=getche();
    ch2=getche();
    mons = ASCIItoBCD(ch1, ch2);
    puts("Enter the days to update: ");
    ch1=getche();
    ch2=getche();
    days = ASCIItoBCD(ch1, ch2);
    _CH = cen; _CL=yrs; _DH= mons;
    _DL=days; _AH =5;
    geninterrupt(0x1a);
    puts("Date Updated");
}
The above sample program takes ASCII input from the user for the new date. After taking all the date units as input the program sets the new date using the BIOS service 1Ah/05H.

Setting the Alarm

```c
void interrupt (*oldint)();
void interrupt newint();
unsigned int far * scr = (unsigned int far *)0xb8000000;
void main ()
{
    oldint = getvect(0x4a);
    setvect(0x4a, newint);
    _AH=6;
    _CH =0x23;
    _CL=0x50;
    _DH=0;
    geninterrupt(0x1a);
    keep(0,1000);
}
void interrupt newint()
{
    *scr=0x7041;
    sound(0x21ff);
}
```

The alarm can be set using BIOS function 1Ah/06h. Once the alarm is set BIOS will generate the interrupt 4Ah when the alarm time is reached. The above program intercepts the interrupt 4Ah such that newint() function is invoked at the time of alarm. The newint() function will just display a character ‘A’ on the upper left corner of the screen. But this program may not work in the presence of DOS or Windows drivers.

Another way to set Alarm

```c
#include <bios.h>
#include <dos.h>
void interrupt newint70();
void interrupt (*oldint70)();
unsigned int far *scr =
    (unsigned int far *)0xb8000000;
unsigned char ASCIItoBCD(char hi, char lo)
{
    hi = hi - 0x30;
    lo = lo - 0x30;
    hi = hi << 4;
    hi = hi | lo;
    return hi;
}
```
void main (void)
{
    int temp;
    unsigned char hrs, mins, secs;
    char ch1, ch2;

    puts("Enter the hours to update: ");
    ch1 = getche();
    ch2 = getch();
    hrs = ASCIItoBCD(ch1, ch2);

    puts("Enter the minutes to update: ");
    ch1 = getche();
    ch2 = getch();
    mins = ASCIItoBCD(ch1, ch2);

    puts("Enter the seconds to update: ");
    ch1 = getche();
    ch2 = getch();
    secs = ASCIItoBCD(ch1, ch2);

    outportb(0x70, 1);
    outportb(0x71, secs);
    outportb(0x70, 3);
    outportb(0x71, mins);
    outportb(0x70, 5);
    outportb(0x71, hrs);
    outportb(0x70, 0x0b);

    temp = inport(0x71);
    temp = temp | 0x70;
    outportb(0x70, 0x0b);
    outportb(0x71, temp);

    oldint70 = getvect(0x70);
    setvect(0x70, newint70);
    keep(0, 1000);
}

void interrupt newint70()
{
    outportb(0x70, 0x0c);
    if ((inport(0x71) & 0x20) == 0x20)
        sound(0x21ff);
    *scr = 0x7041;
    (*oldint70)();
}
This program takes the time of alarm as ASCII input which is firstly converted into BCD. This BCD time is placed in the 64 byte RAM at the bytes which hold the alarm time. Once the alarm time is loaded the register is accessed to enable the interrupts such that other bits are not disturbed. Whenever the RTC generates an interrupt, the reason of the interrupt needs to be established. This can be done by checking the value of status register C, if the 5th bit of register C is set it indicates that the interrupt was generated because the alarm time has been reached. The reason of interrupt generation is established in the function newint70(). If the interrupt was generated because of alarm then speaker is turned on by the sound() function and a character ‘A’ is displayed on the upper left corner of the screen.

Other Configuration Bytes of Battery Powered RAM

**Configuration Data And Battery Operated RAM**

Besides the date and time information, the 64 battery backed memory registers also contain configuration data. Of the memory locations of various BIOS manufacturers, only those that are designated as unreserved contain the same information. Since all the other locations are used at the discretion of BIOS and hardware designers, these locations shouldn't be overwritten by a program.

<table>
<thead>
<tr>
<th>Addr</th>
<th>Content</th>
<th>Addr</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0EH</td>
<td>Diagnostic byte (see below)</td>
<td>18H</td>
<td>High byte in K of an expansion board’s main memory size</td>
</tr>
<tr>
<td>0FH</td>
<td>Status at system power-down</td>
<td>19H</td>
<td>Reserved</td>
</tr>
<tr>
<td>10H</td>
<td>Write to disk (see below)</td>
<td>20H</td>
<td>Reserved</td>
</tr>
<tr>
<td>11H</td>
<td>Hard drive 1 type</td>
<td>2EH</td>
<td>Checksum high byte (memory locations 10H - 2DH)</td>
</tr>
<tr>
<td>12H</td>
<td>Hard drive 2 type</td>
<td>2FH</td>
<td>Checksum low byte (memory locations 10H - 2DH)</td>
</tr>
<tr>
<td>13H</td>
<td>Reserved</td>
<td>30H</td>
<td>Low byte in K of expansion memory size</td>
</tr>
<tr>
<td>14H</td>
<td>Configuration byte (see below)</td>
<td>31H</td>
<td>High byte in K of expansion memory size</td>
</tr>
<tr>
<td>15H</td>
<td>Low byte in K of hard drive main memory size</td>
<td>32H</td>
<td>First two century digits in BCD notation</td>
</tr>
<tr>
<td>16H</td>
<td>High byte in K of motherboard main memory size</td>
<td>33H-3FH</td>
<td>Reserved</td>
</tr>
<tr>
<td>17H</td>
<td>Low byte in K of an expansion board’s main memory size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Diagnostic byte (0EH)

The diagnostic byte documents various errors that may occur during the Power-On Self Test (POST).

```
7 6 5 4 3 2 1 0
```

- 1 = Date or time incorrect (memory locations 00h-09H)
- 1 = Hard drive or controller error
- 1 = Memory size incorrect (memory locations 15H and 16H)
- 1 = Configuration byte incorrect
- 1 = Checksum incorrect (memory locations 2EH - 2FH)
- 1 = Realtime battery clock dead

Disk description (10H)

Memory address 10H of the battery backed RAM contains information identifying the first and the second disk drive formats (5.25-inch or 3.5-inch) and their capacities.

```
7 6 5 4 3 2 1 0
```

- Second drive type:
  - 000b = no second drive
  - 001b = 6.25-inch 320/360K
  - 010b = 5.25-inch 1.2 Meg
  - 011b = 3.5-inch 720K
  - 100b = 3.5-inch 1.44 Meg

- First drive type:
  - 0000b = no disk drive
  - 0001b = 5.25-inch 320/360K
  - 0010b = 5.25-inch 1.2 Meg
  - 0011b = 3.5-inch 720K
  - 0100b = 3.5-inch 1.44 Meg

Configuration byte (14H)

Memory address 14H of the battery backed RAM contains configuration data that specifies the number of disk drives, the video mode at system startup, and the availability of a math coprocessor.

```
7 6 5 4 3 2 1 0
```

- 1 = No disk drive installed
- 1 = Math coprocessor installed
- 1 = Hard drive or controller error

Number of disk drives:
- 00b = reserved
- 01b = CGA / EGA / VGA, 40 columns
- 10b = CGA / EGA / VGA, 80 columns
- 11b = MDA/Hercules, 80 columns
- Unused (0)
Determining Systems Information

Interrupt 11H is used to determine the systems information. On return this service returns the systems info in AX register. The detail of the information in AX register is shown in the slide above.
Int 12H is used to determine the amount of conventional memory interfaced with the processor in kilobytes. The amount of memory above conventional memory (extended memory) can be determined using the service 15H/88H.
Lecture # 20

Types of Processor

Determining the Processor Type

Flags register test to identify **8086**

Pushing or Poping the flags register will set these 4-bits in 8086.

Determining the Processor Type

mov AX, 0
push AX
popf
pushf
pop AX
Test the bits 15 – 12 of AX if all set, the processor is 8086 else higher processor.

The above slides show the test that can be used to determine if the underlying processor is 8086 or not. If its not 8086 some test for it to be 80286 should be performed.
Checking for 80286

<table>
<thead>
<tr>
<th>Flags test for 80286</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov AX, 7000H</td>
</tr>
<tr>
<td>push AX</td>
</tr>
<tr>
<td>popf</td>
</tr>
<tr>
<td>pushf</td>
</tr>
<tr>
<td>pop AX</td>
</tr>
</tbody>
</table>

If the bits 14 – 12 are cleared the processor is 286 only.

if the bits 14-12 are cleared on pushing the flags register then the processor is 80286. This can be checked as shown in the slide above.

<table>
<thead>
<tr>
<th>Alignment Test (If Not 286)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eflags</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>Alignment Check</td>
</tr>
</tbody>
</table>

Alignment Check:

mov dword ptr [12], EDX

In 32-bit processors it is more optimal in terms of speed if double word are placed at addresses which are multiples of 4. If data items are placed at odd addresses the access to such data items is slower by the virtue of the memory interface of such PCs. So it more optimal to assign such variables addresses which are multiple of 4. The 386 and 486 are both 32 bit processors but 486 has alignment check which 386 does not have. This
property can be used to distinguish between 386 and 486. If the previous tests have failed then there is a possibility that the processor is not 8086 or 286. To eliminate the possibility of it being a 386 we perform the alignment test. As shown in the slide above the 18th bit of the EFLAGS register is the alignment bit, it sets if a double word is moved onto a odd address or an address which does not lie on a 4 byte boundary.

![Alignment Test]

```
pushfd
pop eax
mov ecx, eax
mov dword ptr [13], EDX
pushfd
pop eax
```

In the above slide a double word is moved into a odd address. If the processor is 386 then the 18th bit of the EFLAGS register will not be set, it will be set if the processor is higher than 386.
Distinguishing between 486 and Pentium processors

CPUID Test

• 486 will pass the alignment test.
• To distinguish 486 with Pentium CPUID Test is used.

A Pentium and 486 both will pass the alignment test. But a 486 does not support the CPUID instruction. We will next incorporate the CPUID instruction support test to find if the processor is 486 or a Pentium as Pentium does support CPUID instruction.

CPUID Test

If a program can set and also clear bit 21 of Eflags, then processor supports CPUID instructions.
• Set bit 21 of Eflags and read value of Eflags and store it.
• Clear bit 21 of Eflags, read the value of Eflags.
• Compare both the value if bit 21 has changed the CPUID instruction is available.

If the CPUID instruction is available the processor is a Pentium processor otherwise it’s a 486.
More about CPUID Instruction

The CPUID instruction, if available, returns the vendor name and information about the model as shown in the slide above. Beside rest of the test the CPUID instruction can also be used by the software to identify the vendor name.
Testing for Coprocessor

Coprocessor control word

The coprocessor control word contains some control information about the coprocessor. The bit number 7 of coprocessor control word is the Interrupt Enable Flag and bit number 8 & 9 should contain 11 on initialization.

The coprocessor status register stores the status of the coprocessor. Very much like the flags register in the microprocessor, the Coprocessor status word can be used to determine the result of a comparison as shown in the slide.
Following test can be performed to test the presence of coprocessor.

**To Check Coprocessor is Present**
- Initialize
- Read Hi – Byte of Control register.
- If value in Hi – Byte is 3, then coprocessor is available, otherwise its absent.

Once it's established that the coprocessor is present then the model of the coprocessor should be determined. In case an invalid numerical operation is requested the 8087 coprocessor generates an interrupt while the higher coprocessors do not use interrupts in fact they make use of exceptions. This feature can be used to distinguish between 8087 and higher processor as shown in the slide above. The higher processor will not respond to an attempt made to set the IEM flag while 8087 will respond.

**Check for 8087 Coprocessor**
- IEM can be set in 8087.
- IEM cannot be set in 80287, 80387 as they use exception to inform the software about any invalid instruction.
- If an attempt to set this bit using FDISI fails then it implies, its not a 8087 coprocessor.
Distinguishing between 80287 and 80387

Distinguish between 80287 & 80387

• 80387 only allows to reverse the sign of infinity.
• Perform a division by zero.
• If the sign of result can be reversed then the coprocessor is 80387.

if the sign of infinity can be reversed than the coprocessor is 80387 otherwise its 80387
Reading the Computer configuration

```c
void PrintConfig( void )
{
    union REGS Register;
    BYTE AT;
    clrscr();
    AT = (peekb(0xF000, 0xFFFF) == 0xFC);
    printf("Your PC Configuration 

c----------------------------------------------
cPC type              : ");
    switch( peekb(0xF000, 0xF FFE) )
    {
        case 0xFF : printf("PC
"); break;
        case 0xFE : printf("XT
"); break;
        default   : printf("AT or higher
"); break;
    }
    printf("Conventional RAM     : ");
    int86(0x12, &Register, &Register);
    printf(" %d K",Register.x.ax);
    if ( AT )
    {
        Register.h.ah = 0x88;
        int86(0x15, &Register, &Register);
        printf("Additional RAM       :
%d K over 1 megabyte
", Register.x.ax);
    }
    int86(0x11, &Register, &Register);
    printf("Default video mode   :
%dn" , Register.ax >> 6 & 3);
    printf("Serial interfaces    :
%dn" , Register.ax >> 9 & 0x03);
    printf("Parallel interfaces  :
%dn" , Register.ax >> 14);
}

void main()
{
    PrintConfig();
}
```

In this program the general configurations of the computer are read using interrupt 11H, 12H and 15H. First its determined if the Processor is and AT (advanced technology all processors above 8086) type computer or not. This can be done easily by checking its signature byte placed at the location F000:FFFEH which will contain neither 0xFF nor 0xFE if its an AT computer. The program shows the size of conventional RAM using the interrupt 12H, then if the computer is an AT computer then the program checks the extended memory size using int 15H/88H and reports its size. And ultimately the program calls int 11H to show the number and kind of I/O interfaces available.
Lecture # 21

Processor Identification

```Assembly
_getproc proc near
    pushfl ;Secure flag register contents
    push dl

    ;== Determine whether model came before or after 80286 =====
    xor ax,ax ;Set AX to 0
    push ax ;and push onto stack
    popf ;Pop flag register off of stack
    pushf ;Push back onto stack
    pop ax ;and pop off of AX
    and ax,0f000h ;Do not clear the upper four bits
    cmp ax,0f000h ;Are bits 12 - 15 all equal to 1?
    je not_286_386 ;YES --> Not 80386 or 80286

;-- Test for determining whether 80486, 80386 or 80286 -------
    mov dl,p_80286 ;In any case, it's one of the
    mov ax,07000h ;three processors
    push ax ;Push 07000h onto stack
    popf ;Pop flag register off
    pushf ;and push back onto the stack
    pop ax ;Pop into AX register
    and ax,07000h ;Mask everything except bits 12-14
    je pende ;Are bits 12 - 14 all equal to 0?
    inc dl ;No --> it's either an 80386 or an
    ;80486. First set to 386

    ;-- The following test to differentiate between 80386 and ---
    ;-- 80486 is based on an extension of the EFlag register on
    ;-- the 80486 in bit position 18.
    ;-- The 80386 doesn't have this flag, which is why you
    ;-- cannot use software to change its contents.
```

In the above slide the test for 8086 or not is performed by clearing all the bits of flags register then reading its value by pushing flags and then poping it in AX, the bits 15-12 of ax are checked if they have been set then it’s a 8086.
The above slide further performs the test for 80286 if the previous test fails. It sets the bit 14-12 of flags and then again reads back the value of flags through stack. If the bits 14-12 have been cleared then it’s a 80486.

```
cli ;No interrupts now
mov ebx,offset array
mov [ebx],eax
pushfd
pop eax
mov first,eax;

mov [ebx+1],eax
pushfd
pop eax
shr first,18
shr eax,18
and first,1
and eax,1
cmp first,eax
inc dl
sti
jne pende
```

The above code performs the alignment test as discussed before by test the 18th bit after addressing a double word at an odd address.

```
pushfd
pop eax
mov temp, eax
mov eax,1
shl eax,21
push eax
popfd
pushfd
pop eax
shr eax,21
shr temp,21
cmp temp, eax
inc dl
je pende
jmp pende ;Test is ended
```

the above code performs a test to see if CPUID instruction is available or not for which the bit number 21 of flags is set and then read back.
A CPUID Program

#include "stdafx.h"
#include <stdio.h>
#include <dos.h>
unsigned long int id[3];
unsigned char chn="0":
unsigned int steppingid ;
unsigned int model,family,type1 ;
unsigned int cpcw;
int main(int argc, char* argv[])
{
  _asm xor eax,eax
  _asm cpuid
  _asm mov id[0], ebx;
  _asm mov id[4], edx ;
  _asm mov id[8], ecx;
  printf("%s
", (char *) (id));
  _asm mov eax,1
  _asm cpuid
  _asm mov ecx,eax
  _asm AND eax,0xf;
  _asm mov steppingid,eax;
  _asm mov eax, ecx
The above program places 0 in eax register before issuing the CPUID instruction. The string returned by the instruction is then stored and printed moreover other information about family, model etc is also printed.
Detecting a Co Processor

```asm
.asm finit
.asm mov byte ptr cpcw+1, 0;
.asm fstcw cpcw
if ("((char *)&cpcw)+1)==3)
    puts("Coprocessor found");
else    puts("Coprocessor not found");
```

After initialization the control word is read if the higher byte contains the value 3.

```asm
.getco proc near
    mov dx,co_none ;First assume there is no CP
    mov byte ptr cs:wait1,NOP_CODE ;WAIT-instruction on 8087
    mov byte ptr cs:wait2,NOP_CODE ;Replace by NOP
    wait1:    finit ;Initialize Cop
    mov byte ptr cpz+1,0  ;Move high byte control word to 0
    wait2:    fstcw cpz ;Store control word
    cmp byte ptr cpz+1,3 ;High byte control word = 3?
    jne gcende ;No --- No coprocessor
    ; Coprocessor exists. Test for 8087 -----------------------
    inc   dx
    and   cpz,0FF7Fh ;Mask interrupt enable mask flag
    fldcw cpz ;Load in the control word
    fdisi ;Set IEM flag
    fstcw cpz ;Store control word
    test cpz,80h ;IEM flag set?
    jne gcende ;YES --- 8087, end test
```

In the code above the IEM bit is set and then the value of control word is read to analyse change in the control word. If the most significant bit is set then it’s a 8087 co processor otherwise other tests must be performed.
An operation (like division by zero is performed) which results in infinity. Then the sign of the result is reversed, if it can be reversed then its 80387 co-processor otherwise its certainly 80287.

The keyboard interface as discussed earlier uses the IRQ1 and the port 60H as data port, it also uses another port number 64H as a status port. The keyboard can perform synchronous serial I/O.
The above slide shows the detailed meaning of bits in port 64H.

The typematic rate of the keyboard can be controlled by a control word as depicted in the slide above. The delay and typematic rates need to be specified in this control word. The delay indicates the delay between first and second character input whenever a key is pressed. The timing of rest of the successive character inputs for the same key is determined by the typematic rate.
The input character scan code is received at port 60H. A certain bit in the port 64H or keyboard controller is used as the IBF (input buffer full) bit. A device driver can check this bit to see if a character has been received from the keyboard on which this bit will turn to 1.

Similarly some data (as control information) can be send to the keyboard. The processor will write on the port 60H. The device driver will check the OBF (output buffer full) bit of port 64H which remains set as long as the byte is not received by the keyboard. On receipt of the byte from the port 60H the keyboard device write a code 0xFA on the port 60H to indicate that the byte has been received properly.
Lecture # 22

Using the described information we can design a protocol for correctly writing on the keyboard device as described below.

**Keyboard writing Protocol**

- Wait till input buffer is full
- Write on buffer
- Wait till output buffer is full
- Check the acknowledgement byte
- Repeat the process if it was previously unsuccessful.

Keyboard is a typically an input device but some data can also be send to the keyboard device. This data is used as some control information by the keyboard. One such information is the typematic rate. This type matic rate can be conveyed to the keyboard as described by the slide below.

**Command for writing Typematic rate**

0xF3

Means Typematic rate will be sent in the next byte.

Other such control information is the LED status. Every keyboard has three LEDs for representing the status of Num Lock, Caps Lock and the Scroll Lock. If the device driver
needs to change the status then the LED status byte should be written on the keyboard as described below. But before writing this byte the keyboard should be told that the control byte is to be written. This is done by sending the code 0XED before sending the status byte using the above described protocol.

Keyboard LEDs

LED Status byte

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

LED Control byte = 0xED

Changing Typematic Rate

```c
#include <dos.h>
#include <conio.h>
char st [80];
int SendKbdRate(unsigned char data , int maxtry)
{
    unsigned char ch;
    do{
        do{
            ch=inport(0x64);
        }while (ch&0x02);
        outport(0x60,data);
        do{
            ch = inport(0x64);
        }while (ch&0x01);
    }while (maxtry);
}
```
if (ch==0xfa) {
    puts("success\n");
    break;
} 
maxtry = maxtry - 1;
} while (maxtry != 0);
if (maxtry==0)
    return 1;
else
    return 0;
}

The above program has function SendKbdRate(). This function takes 2 parameters, first one is value to be sent and the second one is the maximum number of retries it performs if the byte cannot be sent. This function implements the described protocol. It first waits for the IBF to be cleared and then starts trying to send the byte. The functions stops trying either if 0xFA is received (success) or if the number of retries end (failure).

void main ()
{
    //clrscr();
    SendKbdRate(0xf3,3);
    SendKbdRate(0x7f,3);
    gets(st);
    SendKbdRate(0xf3,3);
    SendKbdRate(0,3);
    gets(st);
}

Now this function is used to change the typematic rate. Firstly 0XF3 is written to indicate that the typematic rate is to be changed then the typematic rate is set to 0x7F and a string can be type to experience the new typematic rate. Again this rate is set to 0. This program will not work if you have booted the system in windows. First boot the system in DOS and then run this program.
Changing LEDs Status

```c
#include <bios.h>
#include <dos.h>
char st[80];
unsigned char far *kbd =
    (unsigned char far *) 0x00400017;
int SendKbdRate(unsigned char data, int maxtry)
{
    unsigned char ch;
    do{
        do{
            ch = inport(0x64);
        } while (ch & 0x02);
        outport(0x60, data);
        do{
            ch = inport(0x64);
        } while (ch & 0x01);
        ch = inport(0x60);
        if (ch == 0xfa)
        { puts("success\n");
            break;
        }
        maxtry = maxtry - 1;
    } while (maxtry != 0);
    if (maxtry == 0)
        return 1;
    else
        return 0;
}
```
void main ()
{
    //clrscr();
    SendKbdRate(0xed,3);
    SendKbdRate(0x7,3);
    puts("Enter a string ");
    gets(st);
    *kbd=(*kbd )|0x70;
    puts("Enter a string ");
    gets(st);
}

Again the same function is being used in this program to turn on the keyboard LEDs. Firstly 0xED is sent to indicate the operation and then 7 is written to turn on all the LEDs. But tuning on the LEDs like this will not change the keyboard status indicated by the byte at 40:17H. If the status for the device driver usage is to change as well then the corresponding at 40:17H can be set by ORing it with 0x70. This program will not work if you have booted the system in windows. First boot the system in DOS and then run this program.
DMA is a device which can acquire complete control of the buses and hence can be used to transfer data directly from port to memory or vice versa. Transferring data like this can prove faster because a transfer will consume 2 bus cycles if it is performed using the processor. So in this approach the processor is bypasses and its cycles are stolen and are used by the DMA controller.
Lecture # 23

The latch B of the DMA interface is used to hold the higher 4 or 8 bits of the 20 or 24 bit absolute address respectively. The lower 16bits are loaded in the base address register and the number of bytes to be loaded are placed in the count register. The DMA requests to acquire buses through the HOLD signal, it receives a HLDA (Hold Acknowledge ) signal if no higher priority signal is available. On acknowledgment the DMA acquires control of the buses and can issue signals for read and write operations to memory and I/O ports simultaneously. The DREQ signals are used by various devices to request a DMA operation. And if the DMA controller is successful in acquiring the bus it sends back the DACK signal to signify that the request is being serviced. For the request to be serviced properly the DMA channel must the programmed accurately before the request.

![DMA Cascading](image)

A single DMA can transfer 8bit operands to and from memory in a single a bus cycle. If 16bit values are to transmitted then two DMA controllers are required and should be cascaded as shown above.
DMA Programming Model

- DMA has 4 – Channels
- Each Channel can be programmed to transfer a block of maximum size of 64k.
- For each Channel there is a
  - Base Register
  - Count Register
  - Higher AddressNibble/Byte is placed in Latch B.
- The Mode register is conveyed which Channel is to be programmed and for what purpose i.e. Read Cycle, Write Cycle, Memory to memory transfer.
- A request to DMA is made to start it’s transfer.

Internal Registers

- No of 16 & 8 bit Internal registers
- Total of 27 internal registers in DMA

<table>
<thead>
<tr>
<th>Register</th>
<th>Number</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Address</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Counter</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Current Address</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Current Counter</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Temporary Address</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Temporary Counter</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Status</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Command</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Intermediate Memory</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Mode</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Mask</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Request</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

The above slides shows the characteristics of each register when a DMA channel is to be programmed and also shows the total number of registers in the DMA controller. Some of the registers are common for all channels and some are individual for each channel.
DMA Modes

- Block Transfer
- Single Transfer
- Demand Transfer

The DMA can work in above listed modes. In block transfer mode the DMA is programmed to transfer a block and does not pause or halt until the whole block is transferred irrespective of the requests received meanwhile.

In Single transfer mode the DMA transfers a single byte on each request and updates the counter registers on each transfer and the registers need not be programmed again. On the next request the DMA will again transfer a single byte beginning from the location it last ended.

Demand transfer is same as block transfer, only difference is that the DREQ signal remains active throughout the transfer and as soon as the signal deactivates the transfer stops and on reactivation of the DREQ signal the transfer may start from the point it left.
Programming the DMA

Programming the DMA controller

The following table shows the different DMA controller registers which are used to determine the status of the controller or define the parameters:

<table>
<thead>
<tr>
<th>Register</th>
<th>Port*</th>
<th>Port**</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>00h</td>
<td>0DCh</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>00h</td>
<td>0DCh</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Request</td>
<td>09h</td>
<td>0D2h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Masking</td>
<td>10Ah</td>
<td>0D4h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>08h</td>
<td>0D5h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>ByteWord-FlipFlip</td>
<td>0Ch</td>
<td>0D8h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Intermediate memory</td>
<td>0Dh</td>
<td>0DAh</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>0Dh</td>
<td>0DAh</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Masking reset</td>
<td>0Eh</td>
<td>0DCh</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Masking</td>
<td>0Fh</td>
<td>0DEh</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

*slave in an AT / only one DMA in a PC/XT
**master in an AT / not present in a PC/XT

Before you access one of these registers, decide if you're addressing the master or the slave. If you have a PC/XT that has only one DMA controller, it's not possible to access a second DMA controller (master in the AT).

The above table shows the addresses of all the registers that should be programmed to perform a transfer. These registers act as status and control registers and are common for all the channels.
DMA Status Register

Terminal count if reached signifies that the whole of the block as requested through some DMA channel has been transferred. The above status register maintains the status of Terminal count (TC) and DREQ for each channel within the DMA.
This is the command register. It is used to program various common parameters of transfer for all the channels.
This register can be used to simulate a DMA request through software (in case of memory to memory transfer). The lower 2 bits contains the channel number to be requested and the bit # 2 is set to indicate a request.
DMA Mask–1 Register

Another way to mask or make a channel receptive to DMA requests is provided by mask register 2. In contrast to the mask register 1, all four channels are affected. Use this register only to change the status for all four channels simultaneously.

As its name suggests, the mode register determines a channel's operating mode. You can specify if the next DMA transfer will happen as a single transfer, a block transfer, or a demand transfer. It also specifies if the channel is to cascade two DMA controllers. In most cases you won't have to change this later setting since this happens when the computer is booted.

This register can be used to mask/unmask requests from a device for a certain DMA channel. The lower 2 bits contains the channel number and the bit #2 is set if the channel is to be masked.
This register can also be used to mask the DMA channels. It contains a single bit for each channel. The corresponding bit is set to mask the requests for that channel.
This register can be used to set the mode on each. The slide shows the detail of the values and bits which should be placed in the register in order to program a required mode.

Bit 5 of the mode register, determine the "direction" of a transfer. This "direction" isn't to or from a peripheral, rather it's forward or backward direction in memory. So you can decrement instead of increment the memory address during a DMA transfer. In this case, a data block is read backwards to forwards by the peripheral. Also the ending address of the buffer is loaded into the proper register before starting the transfer.
A channel is programmed for a start address and the count of bytes to be transferred before the transfer can take place. Both these values are placed in various registers according to the channel number as shown by the slide above. Once the transfer starts these values start changing. The start address is updated in to the current address and the count is also updates as bytes are transferred. During the transfer the status of the transfer can be analyzed by getting the values of these registers listed In the slide above for the channel(s) involved in the transfer.
The above slide shows the port number for each channel in which the higher 4 or 8 bits of the absolute address is stored in case of 20 or 24 bit address bus.

```
#include <dos.h>
#include <bios.h>
char st[80];
unsigned long int temp;
unsigned int i;
unsigned int count=48;

void main (void)
{
    temp=(unsigned long int)_DS;
    temp = temp << 4L;
    i = *((unsigned int *)&temp);
    temp = temp>>16L;
```

This program, programs the DMA channel 3 for read cycle by placing 0x0B in mode register (0x0B). Before the channel is unmasked and the channel mode is programmed the base address the count and the higher 4 or 8 bits of the address should be placed in base register, count register and Latch B respectively. The 20 (or 24) bit address is calculated. The higher 4 (or 8) bits are placed in the Latch B for channel 3, then the rest of the 16 bits of the base address are placed in base register for channe3 and ultimately the count is
loaded into the count register for channel 3.

```c
outportb(0x81, *((unsigned char *)&temp));
outportb(0x06, *((unsigned char *)&i));
outportb(0x06, *((unsigned char *)&i)+1));
count--;
outportb(0x07, *((unsigned char *)&count));
outportb(0x07, *((unsigned char *)&count)+1));
outportb(0x0b, 0x0b);
outportb(0x08, 0);
outport(0x0a, 3);
getch();
```
Lecture # 25

This program attempts to perform memory to memory transfer operation. This program will only work for a 8086 processor, higher processors’ DMA may not support memory transfer.

```c
#include <dos.h>
#include <bios.h>

char st[2048]="hello u whats up?0";
char st1[2048]="xyz";
unsigned long int temp;
unsigned int i;

void main (void)
{
    temp=_DS;
    temp = temp<<4;
    i = *((unsigned int *)&temp);
    temp = temp >>16;
    outportb(0x87,*((unsigned char *)&temp));
    outportb(0,*((unsigned char *)&i));
    outportb(0,*(((unsigned char *)&i)+1));
    outportb(1,0xff);
    outportb(1,0x07);
    outportb(0x0b,0x88);
    temp=_DS;
    temp= temp+128;
    temp=temp<<4;
    i = *((unsigned int *)&temp);
    temp=temp >>16;
    outportb(0x83,*((unsigned char *)&temp));
    outportb(2,*((unsigned char *)&i));
    outportb(2,*(((unsigned char *)&i)+1));
}
```
This program, programs the channel 0 and channel 1 of the DMA. It loads the address of Source string st in base register and the Latch B and loads the count register for channel 0 and does the same for st1. It then programs the mode, mask and command register for memory to memory transfer and to unmask channel 0 and channel 1.

```c
outportb(3,0xff);
outportb(3,0x07);
outportb(0x0b,0x85);
outportb(0x08,1);
outportb(0x0f,0x0c);
outportb(0x09,0x04);
while (!kbhit())
{
    printf("Channel 0 = %x,%x",inportb(0x01),inportb(0x01));
    printf("Channel 1 = %x,%x",inportb(0x03),inportb(0x03));
    printf("Status = %x\n",inportb(0x08));
}
puts(st1);
```

File Systems

**File System**

- Disk Architecture
- Disk Partitioning
- File systems
Disk Architecture

- Disk is a circular which is hollow from the center
- This shape is inherently useful for random access.

On tracks and sectors

Tracks are the circular division of the disk and the sectors are the longitudinal division of the disk as shown in the diagram above.
**Addressable unit Parameters**

- Heads
- Sectors
- Tracks

An addressable unit on disk can be addressed by three parameters i.e. head #, sector # and track #. The disk rotates and changing sectors and a head can move to and fro changing tracks. Each addressable unit has a unique combination of sec#, head# and track# as its physical address.

**Blocks**

- Blocks are the sectors per track
- Smallest addressable unit in memory
- Address of block is specified as a unique combination of three parameters (i.e. track, head, sec)
Density of Magnetic media

- Density of magnetic media is the determinant of the amount of data that can reside stably on the disk for example floppy disk come with different densities.
  - Double Density
  - High Density

Effect of surface area on disk size

- Increasing the surface area clearly increases the amount of data that can reside on the disk as more magnetic media no resides on disk but it might have some drawbacks like increased seek time in case only one disk platter is being used

Hard Disks

- Greater amounts of data can reside on hard disk
  - As greater amount of magnetic media can reside on the hard surface of the disk
  - Also because the surface area of the disk is increased by increasing the number of platters as shown in the diagram
Hard Disk Architecture

- Cylinders
  - In case of hard disk where there are number of platters the term track is replaced by cylinder
  - Cylinder is a collection of corresponding tracks if track on platter changes so will the tracks on rest of the platters as all the heads move simultaneously

Rotational Delay

- While accessing a selected block the time required by the disk to rotate to the specified sector is called rotational delay
Seek Time

- While accessing a selected block Time required by the head to reach the particular track/cylinder is called seek time

Access Time

- The accumulative time that is required to access the selected block is called access time
- Access time includes the delay required by disk rotation as well as head movement.

Head is like Electric Coil

- Disk follow the basic principle of magnetism of dynamo.
- When ever a magnetized portion of disk runs along the coil like head electricity is produced in the head which is interpreted as a logic 1
- And whenever a demagnetized portion on the disk runs through the head no electricity is produced in head which is interpreted as logic 0
Head position and precautions
- The head is touching the surface of floppy disk which rotates at a low speed of 300 RPM
- The head is not touching the surface of hard disk which run at high speeds up to 9600 RPM but is at a few microns distance from the surface
- All the magnetic disk are made out of magnetic media and hence data may be lost by exposing them to sunlight, heat, radiation, magnetic or electric fields.
- Dust is harmful and even fatal in case of head disk by the virtue of its speed and its distance of head from the surface
Lecture # 26

Reading/Writing a physical Block

- biosdisk(int cmd, int drive, int head, int track,
  int sector, int nsects, void * buffer);

Cmd 0 = disk reset
   1 = disk status (status of last disk operation)
   2 = disk read
   3 = disk write
   4 = disk verify
   5 = disk format

Now we establish how a block can be read or written if its address is known. The function biosdisk() can be used to read or write a physical block. The slide shows its parameter. It takes the command (cmd), drive number, head number, track number, number of sectors to be read or written and the memory from where the data is to read from or written to. Command signifies the operation that is to be performed.

Reading Writing a physical Block

- Drive 0x80 = first fixed disk (physical drive)
   0x81 = second fixed disk
   0x82 = third fixed disks
   ....
   0x00 = first removable disk
   0x01 = second removable disk
   ....

Drive number is described in the slide below it start from 0 for first removable disk and starts from 0x80 for first fixed disk.
Reading Writing a physical Block

```c
#include <bios.h>
#include <dos.h>
FILE *fp;
unsigned char buf[512];
unsigned char st[60];
unsigned char headno[10];
unsigned char secno[10];
unsigned char trackno[10];
void main (void)
{
    int i ;
    for (i=0;i<512;i++)
        buf[i]=0;
    gets(st);
    fp=fopen(st,"wb");
    printf("Head ");
    gets(headno);
    puts (headno);
    printf("sector ");
    gets(secno);
    puts(secno);
    printf("track ");
    gets(trackno);
    puts(trackno);

Cont...
```
The above program reads a physically addressed block from disk using bios disk() function for first fixed disk. The program after reading the specified block writes it to a file and then closes the file.

**Limitation of biosdisk**
- Biosdisk() calls int 13H/0/1/2/3/4/5
- Details of 13H services used by Biosdisk()
- On Entry
  - AH = service #
  - AL = No. of sectors
  - BX = offset address of data buffer
  - CH = track #
  - CL = sector #
  - DH = head/side #
  - DL = Drive #
  - ES = Segment Address of buffer.

However there are some limitation of this biosdisk() while using large disks. This function uses the int 13H services listed in the slide above.
**Limitation of biosdisk()**

- Large sized disk are available now with thousands of tracks
- But this BIOS routine only is capable of accessing a max. of 1024 tracks.
- Hence if a large disk is being used not whole of the disk can be accessed using this routine.

The parameter sizes provided by these services may not be sufficient to hold the track number of block to be accessed.

**Extended BIOS functions**

- Extended BIOS functions of int 13h can be used for operations on higher tracks
- As discussed later usual BIOS functions can access a maximum of 504MB of disk approx.

Above slide shows for which disks extended services are required to access the block efficiently.
Extended BIOS functions

<table>
<thead>
<tr>
<th></th>
<th>BIOS</th>
<th>IDE</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Sec</td>
<td>63</td>
<td>255</td>
<td>63</td>
</tr>
<tr>
<td>Max heads</td>
<td>256</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Max Capacity</td>
<td>1024</td>
<td>65536</td>
<td>1024</td>
</tr>
</tbody>
</table>

Highest biosdisk() capacity

• Hence the highest capacity of disk can be accessed using bios functions is
• $63 \times 16 \times 1024 \times 512 = 504 \text{ MB approx.}$

But IDE disk interface can support disks with memory space larger than 504MB as shown in the next slide.
Highest IDE capacity

• Hence highest physical capacity of the disk according to the IDE interface is
  \[255 \times 16 \times 65536 \times 512 = 127 \text{GB}\]

• Extended BIOS functions allow to access disk with sizes greater than 504 MB through LBA translation.

Extended services require that the address of the block is specified as a LBA address.

LBA Translation Method

• Each unique combination of three parameters is assigned a unique index as shown below

  • Firstly all the sectors of a cylinder are indexed for head = 0, then when whole track has been indexed the sector in the track of same cylinder with head = 1 are indexed and so on up till the end of all heads
  
  When done with one cylinder the same is repeated for the next cylinder till the end of cylinders
LBA translation is done by numbering the blocks with a single index. The indexes are assigned to blocks as shown in the slide below. In terms of the disk geometry firstly all the sectors of a tracks will be indexed sequentially, then the track exhausts the next track is chosen on the other side of the disk and so on all the tracks in a cylinder are indexed. When all the blocks within a cylinder has been indexed the same is done with the next cylinder.

<table>
<thead>
<tr>
<th>Cylinder</th>
<th>head</th>
<th>sec</th>
<th>LBA address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>63</td>
<td>62</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>126</td>
</tr>
</tbody>
</table>

if the CHS (cylinder, head, sector) address of a disk is known it can be translated in to the LBA address and vice versa. For this purpose the total number of cylinders, heads and sectors must also be known.

**Mathematical Notation for LBA translation**

- LBA address = \((C \times H^\prime + H) \times S^\prime + S - 1\)

Where
- \(C\) = Selected cylinder number
- \(H^\prime\) = No. of heads
- \(H\) = Selected head number
- \(S^\prime\) = Maximum Sector number
- \(S\) = Selected Sector number
Also conversely LBA to CHS translation can also be done using the formulae discussed in the following slide but for this the total number of cylinders, heads and sectors within the disk geometry should be known.

**LBA to CHS translation**

- Conversely LBA address can be translated into CHS address
  
  \[
  \text{cylinder} = \frac{\text{LBA}}{(\text{heads\_per\_cylinder} \times \text{sectors\_per\_track})}
  \]

  \[
  \text{temp} = \text{LBA} \mod (\text{heads\_per\_cylinder} \times \text{sectors\_per\_track})
  \]

  \[
  \text{head} = \frac{\text{temp}}{\text{sectors\_per\_track}}
  \]

  \[
  \text{sector} = (\text{temp} \mod \text{sectors\_per\_track}) + 1
  \]
Disk Address Packet is a data structure used by extended int 13H services to address a block and other information for accessing the block. Its structure is defined in the slide below.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Byte</td>
<td>Size, Should not be less than 16</td>
</tr>
<tr>
<td>1</td>
<td>Byte</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Byte</td>
<td>No. of blocks to transfer, Max value no greater than 7FH</td>
</tr>
<tr>
<td>3</td>
<td>Byte</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Double Word</td>
<td>Far address of buffer</td>
</tr>
<tr>
<td>8</td>
<td>Quad word</td>
<td>LBA address</td>
</tr>
</tbody>
</table>
Lecture # 27

Extended Read

- Service used for extended read is int 13h/42h
  
  **On Entry**
  
  AH=42H
  
  DL=drive #
  
  DS:SI= far address of Disk address packet
  
  **On Exit**
  
  If CF=0
  
  AH=0= Success
  
  If CF=1
  
  AH= Error code

Interrupt 13H/42H can be used to read a LBA addressed block whose LBA address is placed in the Disk Address packet as described in the slide above.

Extended Write

- Service used for extended write is int 13h/43h
  
  **On Entry**
  
  AH=43H
  
  AL=0,1 write with verify off
  
  2 write with verify on
  
  DL=drive #
  
  DS:SI= far address of Disk address packet
  
  **On Exit**
  
  If CF=0
  
  AH=0= Success
  
  If CF=1
  
  AH= Error code

Similarly int 13H / 43H can be used to write onto to LBA addressed block as described in the slide above.
Reading a LBA block

```c
#include <dos.h>
#include <bios.h>
struct DAP {
    unsigned char size;
    unsigned char reserved1;
    unsigned char blocks;
    unsigned char reserved2;
    unsigned char far *buffer;
    unsigned long int lbalod;
    unsigned long int lbahid;
} dap;
char st[80];
unsigned char buf[512];
FILE *fptr;

void main (void)
{
    puts ("enter the lba low double word: ");
    gets (st);
    dap.lbalod = atol (st);
    puts ("enter the lba high double word: ");
    gets (st);
    dap.lbahid = atol (st);
    dap.size = 16;
    dap.reserved1 = 0;
    dap.blocks = 1;
    dap.reserved2 = 0;
    dap.buffer = (unsigned char far *)MK_FP(_DS,buf);
    _AH = 0x42;
    _DL = 0x80;
    _SI = (unsigned int)&dap;
    geninterrupt(0x13);
    puts ("enter the path: ");
    gets (st);
    fptr = fopen(st,"wb");
    fwrite(buf,512,1,fptr);
    fclose (fptr);
}
```
The above slides list a program that performs a block read operation using the interrupt 13H/42H. A structure of type DAP is created, and appropriate values are placed into it which includes its LBA address. The offset address of dap is placed in SI register and the DS already contains its segment address as it has been declared a global variable. The drive number is also specified and the interrupt is invoked. The interrupt service reads the contents of the block and places it in a buffer whose address was specified in dap. The contents of this buffer are then written on to a file. Slide 7

**Disk Partitioning**
- Partition Table contains information pertaining to disk partitions.
- Partition Table is the first physical sector
  - Head = 0
  - Track/Cylinder = 0
  - Sec = 1 or LBA = 0
- Partition Table at CHS = 001 is also called MBR (Master Boot Record).

**Structure of Partitioning Table**
- Total size of Partition Table is 512 bytes.
- First 446 bytes contains code which loads the boot block of active partition and is executed at Boot Time.
- Rest of the 66 bytes is the Data part.
- Last two bytes of the Data part is the Partition table signature.
File System for Each O.S.

- On a single disk there can be 4 different file systems and hence 4 different O.S.
- Each O.S. will have its individual partition on disk.
- Data related to each partition is stored in a 16-bytes chunk within the Data Part of Partition Table.

Structure of Data Part of P.T.

<table>
<thead>
<tr>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Bytes</td>
<td>Partition into of 1st partition.</td>
</tr>
<tr>
<td>16 Bytes</td>
<td>Partition into of 2nd partition.</td>
</tr>
<tr>
<td>16 Bytes</td>
<td>Partition into of 3rd partition.</td>
</tr>
<tr>
<td>16 Bytes</td>
<td>Partition into of 4th partition.</td>
</tr>
<tr>
<td>02 Bytes</td>
<td>Signature</td>
</tr>
</tbody>
</table>

The data part can contain information about four different partitions for different Operating systems. Each partition information chunk is 16 bytes long and the last two bytes at the end of the partition table data part is the partition table signature whose value should be AA55 indicating that the code part contains valid executable code.
The structure of the information stored in each 16 byte for partition is shown in the slides below.

<table>
<thead>
<tr>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>80H if Bootable, 0 if Not</td>
</tr>
<tr>
<td>Byte</td>
<td>Head # for first block in the partition</td>
</tr>
<tr>
<td>Byte</td>
<td>0 – 5 bits are sector # for first block within the partition and bits 6 -7 are higher bits of cylinder #</td>
</tr>
<tr>
<td>Byte</td>
<td>Low 8-bits of cylinder # for last block within the partition.</td>
</tr>
<tr>
<td>Byte</td>
<td>File System ID</td>
</tr>
</tbody>
</table>

The byte at the offset 4 in the 16 byte data part contains the file system ID which can have various values depending upon the type of OS in use as described by the slides below.
### File System ID

0 ~ FF for various O.S.

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Empty</td>
</tr>
<tr>
<td>01</td>
<td>DOS 12-bit partition</td>
</tr>
<tr>
<td>02</td>
<td>Xenix root</td>
</tr>
<tr>
<td>03</td>
<td>Xenix/usr</td>
</tr>
<tr>
<td>04</td>
<td>MS-DOS 16-bits &lt; 32MB</td>
</tr>
<tr>
<td>05</td>
<td>MS-DOS extended partition can manage disks of sizes up to 8.4 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>MS-DOS 16-bits FAT &gt;= 32MB</td>
</tr>
<tr>
<td>07</td>
<td>OS/2, 1FS = Installable file system</td>
</tr>
<tr>
<td></td>
<td>Advanced Unix</td>
</tr>
<tr>
<td></td>
<td>Windows NT NTFS</td>
</tr>
<tr>
<td>08</td>
<td>AIX Boot partitions</td>
</tr>
<tr>
<td>09</td>
<td>AIX Data partitions</td>
</tr>
<tr>
<td>0A</td>
<td>OS/2 Boot Manager</td>
</tr>
<tr>
<td>0B</td>
<td>Win 95 FAT 32</td>
</tr>
<tr>
<td>0C</td>
<td>Win 95 FAT 32 LBA Mapped</td>
</tr>
<tr>
<td>0E</td>
<td>Win 95 FAT 16 LBA Mapped</td>
</tr>
<tr>
<td>0F</td>
<td>Extended partitions LBA Mapped</td>
</tr>
</tbody>
</table>
**Primary Partition**
- Partition defined in the MBR (Master Boot Record) are primary partition.
- Each Primary Partition contains information about its respective O.S.
- However if only one O.S. is to be installed then extended partitions.

**Extended Partitions**

However if a single operating system is to be kept for instance, then the disk can be divided into primary and extended partitions. Information about primary and extended partition is kept in the first physical block. The extended partition may again be divided into a number of partitions, information about further partitions will be kept in extended partition table which will be the first physical block within extended partition (i.e. it will not the first block of primary partition.). Moreover there can be extended partitions within extended partitions and such that in then end there are number of logical partitions this can go on till the last drive number in DOS.
Lecture # 28

Extended Partitions

1st Block of Extended Partitions

Primary Partition Logical Drive Extended Partition
Here it can be seen that the first partition table maintains information about the primary and extended partitions. The second partition table similarly stores information about a logical and a extended partition within the previous extended partition. Similarly for each such extended partition there will be a partition table that stores information about the logical partition and may also contain information about any further extended partition. In this way the partition tables form a chain as depicted in the slide below. The last partition table within the chain contains just a single entry signifying the logical drive.
The above dump shows the dump of the data part of the first partition table which has two 16 bytes entries, the values in the entries can be interpreted as the values shown in the following slide.
First Partition
System ID = 0c = Windows FAT32 partition (LBA MApped)
first block = 3F
No. of blocks = 01388afc
end cylinder# = 1023
end sec # = 63 indicating a LBA disk

Second Partition
System ID = 0f = Extended windows partition
Start block (relative to the start) = 01388b3b = 20482875
No. of blocks = 0390620a = 59793930

Here is the information found in the partition table of first extended partition table which implies another extended and a logical partition.
First Partition
System ID = 0b = Windows FAT32 partition
first block = 3F
First block physical address = 3F + 01388b3b
No. of blocks = 01388afc
end cylinder# = 1023
end sec # = 63 indicating a LBA disk

Second Partition
System ID = 05 = Extended DOS partition
Start block (relative to the start) = 01388b3b = 20482875
Start block (physical) = 01388b3b + 01388b3b = 2711676H = 40965750
No. of blocks = 01388b3b = 20482875

Here is information in the partition table of the second extended partition that implies yet another extended partition and a logical partition with the details shown in the following slide.
First Partition
System ID = 0b = Windows FAT32 partition
first block = 3F
First block physical address = 3F + 01388b3b +1388b3b
No. of blocks = 01388fc
end cylinder# = 1023
end sec # = 63 indicating a LBA disk

Second Partition
System ID = 05 = Extended DOS partition
Start block (relative to the start of extended partition) = 2711676H = 40965750
Start block (physical) = 01388b3bH + 2711676H = 3A9A1B1H = 61448625
No. of blocks = 11f4b94 = 18828180
Lecture # 29

LBA=0 (MBR)  LBA =1388B3B

<table>
<thead>
<tr>
<th>Logical Drive</th>
<th>Extended Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F 1388AFC</td>
<td>0390620A</td>
</tr>
</tbody>
</table>

LBA=1388B3B

<table>
<thead>
<tr>
<th>Logical Drive</th>
<th>Logical Drive</th>
<th>Extended Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd Partition Table</td>
<td>Relative start = 1388B3B</td>
<td></td>
</tr>
<tr>
<td>1388B3B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2711676H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Above slides shows the information collected as yet which indicates the logical drive there starting LBA blocks, the number of block, the hidden blocks etc. The following slide shows the contents of the data part of partition table of the last extended partition.

It has just one data entry for the logical drive indicating that this is the partition table in the chain. The detail of the contents this partition table are shown below.
First Partition
System ID = 07 = Windows FAT32 partition
first block = 3F
First block physical address = 3F + 01388b3b + 1388b3b + 1388b3b (Blocks in previous partitions)
No. of blocks = 011f4b55
end cylinder# = 1023
end sec # = 63 indicating a LBA disk

Second Partition
System ID = 0 = Unused

The following slide shows the summary of all the data collected as yet depicting 4 logical drives and the number of blocks within.
```c
#include <dos.h>
#include <bios.h>

struct D AP {
    unsigned char size;
    unsigned char reserved1;
    unsigned char blocks;
    unsigned char reserved2;
    unsigned char far *buffer;
    unsigned long int lbalod;
    unsigned long int lbahid;
} dap;

struct PartEntry {
    unsigned char BootableFlag;
    unsigned char StartHead;
    unsigned char StartSector;
    unsigned char StartCylinder;
    unsigned char SystemID;
    unsigned char EndHead;
    unsigned char EndSector;
    unsigned char EndCylinder;
    unsigned long AbsBegin;
    unsigned long SectCount;
};
```
struct PartTable
{
    unsigned char code[446];
    struct PartEntry e[4];
    unsigned int signature;
};

struct DAP dap;

void ReadLBA(unsigned int drive,
             unsigned long int lbalo, unsigned long int lbahi,
             unsigned char * buffer, int nsects)
{
    dap.lbalod = lbalo;
    dap.lbahid = lbahi;
    dap.size = 16;
    dap.reserved1 = 0;
    dap.blocks = nsects;
    dap.reserved2 = 0;
    dap.buffer = buffer; /*MK_FP(_DS,buf);*/
    _A=0x42;
    _DL=drive;
    _SI=(unsigned int)&dap;
    geninterrupt(0x13);
}

void GetPart(unsigned char drive,
             unsigned long int low, unsigned long int high)
{
    struct PartTable p;
    unsigned int Ssect, Esect;
    unsigned int Scyl, Ecyl;
    unsigned long int Bsect;
    int i;
    ReadLBA(drive, low, high, (unsigned char *)&p, 1);
    if (p.signature == 0xaa55)
    {
        for (i = 0; i < 4; i++)
        {
            if (p.e[i].SystemID != 0)
            {
                Ssect = p.e[i].StartSector;
                Ssect = Ssect << 2;
                Scyl = p.e[i].StartCylinder;
                Scyl = *(((unsigned char *)&Scyl) + 1);
                Esect = p.e[i].EndSector;
                Esect = Esect << 2;
                Ecyl = p.e[i].EndCylinder;
                /* ... */
            }
        }
    }
}
Above is a listing of a simple program that reads the partition table using the extended 13H services. It then displays the contents of the data part of the partition table read. For this purpose it uses various data structures designed in reflection of the partition table and 16 bytes data entries within. The program uses recursion and calls the getpart() function recursively whenever it finds an extended partition to read the data within the extended partition table.
Get Drive Parameters

On Entry:
- AH – 48
- DL – Drive number
- DS:SI – Address of result buffer

On Exit:
- Carry Clear
  - AH – 0
  - DS:SI – result buffer
- Carry Set
  - AH – Error Code

The partition table data entry also stores the CHS address of the starting block. But this address is left insignificant if a LBA enable disk is in question. However LBA address can be used in place of the CHS address, and in case CHS address is required it can be calculated if the total number of tracks, sectors and heads are known. To get the total number of tracks, sectors and head the above described service can be used.
### Lecture # 30

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>Buffer Size, must be 26 or greater. <em>The caller sets this value to the maximum buffer size.</em> If the length of the buffer is less than 30, this function does not return the pointer to the Enhanced Disk Drive structure (EDD). If the Buffer Size is 30 or greater on entry, it is set to exactly 30 on exit. If the Buffer Size is between 26 and 29, it is set to exactly 26 on exit. If the Buffer Size is less than 26 on entry an error is returned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Information Flags</th>
</tr>
</thead>
</table>

In the following table, a 1 bit indicates that the feature is available, a 0 bit indicates the feature is not available and will operate in a manner consistent with the conventional 16-bit interface.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DMA boundary errors are handled transparently</td>
</tr>
<tr>
<td>1</td>
<td>The geometry supplied in bytes 8-12 is valid</td>
</tr>
<tr>
<td>2</td>
<td>Device is removable</td>
</tr>
<tr>
<td>3</td>
<td>Device supports write with verify</td>
</tr>
<tr>
<td>4</td>
<td>Device has change line support (bit 2 must be set)</td>
</tr>
<tr>
<td>5</td>
<td>Device is lockable (bit 2 must be set).</td>
</tr>
<tr>
<td>6</td>
<td>Device geometry is set to maximum, no media is present (bit 2 must be set). This bit is turned off when media is present in a removable media device.</td>
</tr>
<tr>
<td>7-15</td>
<td>Reserved, must be 0</td>
</tr>
</tbody>
</table>
Above slides shows the structure of result buffer used by extended 13H services. If a
extended service returns a value it will be stored in the result buffer as described above.

```c
#include <bios.h>
#include <dos.h>
struct RESULTBUFFER
{
    unsigned int size;
    unsigned int infoflags;
    unsigned long int cylinders;
    unsigned long int heads;
    unsigned long int sectors;
    unsigned long int locount;
    unsigned int hicount;
    unsigned int bytespersector;
    unsigned long int configptr;
} rb;
```
The above program uses a RESULTBUFFER data structure in reflection of the result buffer described in previous slides. It uses the interrupt 13H/48H to get the drive parameters and then displays the received total number of sectors, heads and cylinders.
This is also a quite similar program only difference is that it also translates a LBA address into CHS address and displays it, for this purpose it gets the drive parameters to know the total number of heads, sectors and cylinders.

LSN (Logical Sector Number)

<table>
<thead>
<tr>
<th>C</th>
<th>H</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 = Partition Table</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 = Boot Block</td>
</tr>
</tbody>
</table>

Boot Block has LSN = 0
- If the blocks are indexed from the boot block such that the boot block has index = 0, then this index is called LSN.
- LSN is relative index from the start of logical drive, not the physical drive.

LSN is also indexed like LBA the only difference is that LBA is the address relative to the start of physical drive (i.e. absolute), whereas LSN address is the address from the start of logical partition i.e relative.
As in the above example it can be noticed that the LBA = 0 is not the same as LSN=0. The LBA=0 block is the first block on disk. Whereas each logical partition has LSN=0 block which is the first block in logical drive and is not necessarily the first block on physical drive. Also notice the hidden blocks between the first physical block on each partition and its first LSN block. These hidden blocks are not used by the operating system for storing any kind of data.

**Conclusion**

- LBA is physical or absolute address.
- LSN is relative address with respect to the start of Logical Drive.
File System Data Structures
- BIOS Parameter Block (BPB)
- Drive Parameter Block (DPB)
- File Control Block (FCB)
- FAT 12, FAT 16, FAT 32
- Master File Table (MFT)

To understand the file systems of DOS and Windows the above given data structure should be understood which are used by the operating system for file management. In the coming lecture these data structures will be discussed in detail.

Anatomy of a FAT based file system

Above slide shows the overall anatomy of a FAT based system. Starting block(s) is /are the boot block(s), immediately after which the FAT (File allocation table) starts. A typical volume will contain two copies of FAT. After FAT the root directory is situated which contain information about the files and folders in the root directory. Whole of this area constitutes the systems area rest of the area is used to store user data and folders.
Clusters
- A cluster is a collection of contiguous blocks.
- User Data is divided into clusters
- Number of blocks within a cluster is in power of 2.
- Cluster size can vary depending upon the size of the disk.
- DOS has a built in limit of 128 blocks per cluster.
- But practically limit of 64 blocks per cluster has been established.
- We will learn more about the size of clusters, later.

BPB (BIOS Parameter Block)
- Situated within the Boot Block.
- Contains vital information about the file system.

BIOS parameter block is a data structure maintained by DOS in the boot block for each drive. The boot block is typically a 512 byte block which as seen the previous slides is the first logical block i.e. LSN = 0. It contains some code and data. The data part constitutes the BPB. Details for FAT 12 and 16 are shown in following slides.
**BPB (BIOS Parameter Block)**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field Length</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0B</td>
<td>WORD</td>
<td>Bytes per Sector. The size of a hardware sector. Usually 512.</td>
</tr>
<tr>
<td>0x0D</td>
<td>BYTE</td>
<td>Sectors Per Cluster. The number of sectors in a cluster. The default cluster size for a volume depends on the disk size and the file system.</td>
</tr>
<tr>
<td>0x0E</td>
<td>WORD</td>
<td>Reserved Sectors. The number of sectors from the Partition Boot Sector to the start of the first file allocation table, including the Partition Boot Sector. The minimum value is 1.</td>
</tr>
<tr>
<td>0x10</td>
<td>BYTE</td>
<td>Number of file allocation tables (FATs). The number of copies of the file allocation table on the volume. Typically, the value of this field is 2.</td>
</tr>
<tr>
<td>0x11</td>
<td>WORD</td>
<td>Root Entries. The total number of file name entries that can be stored in the root folder of the volume.</td>
</tr>
<tr>
<td>0x13</td>
<td>WORD</td>
<td>Small Sectors. The number of sectors on the volume if the number fits in 16 bits (65535). For volumes larger than 65536 sectors, this field has a value of 0 and the Large Sectors field is used instead.</td>
</tr>
<tr>
<td>0x15</td>
<td>BYTE</td>
<td>Media Type. Provides information about the media being used. A value of 0xF8 indicates a hard disk.</td>
</tr>
<tr>
<td>0x16</td>
<td>WORD</td>
<td>Sectors per file allocation table (FAT). Number of sectors occupied by each of the file allocation tables on the volume.</td>
</tr>
<tr>
<td>0x18</td>
<td>WORD</td>
<td>Sectors per Track.</td>
</tr>
<tr>
<td>0x1A</td>
<td>WORD</td>
<td>Number of Heads.</td>
</tr>
<tr>
<td>0x1C</td>
<td>DWORD</td>
<td>Hidden Sectors.</td>
</tr>
<tr>
<td>0x20</td>
<td>DWORD</td>
<td>Large Sectors. If the Small Sectors field is zero, this field contains the total number of sectors in the volume. If Small Sectors is nonzero, this field contains zero.</td>
</tr>
</tbody>
</table>

**Physical Disk Number.** Physical Disk Number. This is related to the BIOS physical disk number. Floppy drives are numbered starting with 0x00 for the A disk. Physical hard disks are numbered starting with 0x80. The value is typically 0x80 for hard disks, regardless of how many physical disk drives exist, because the value is only relevant if the device is the startup disk.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field Length</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x24</td>
<td>BYTE</td>
<td>Current Head. Not used by the FAT file system. (Reserved)</td>
</tr>
<tr>
<td>0x25</td>
<td>BYTE</td>
<td>Signature. Must be either 0x27, 0x28 or 0x29 in order to be recognized by Windows.</td>
</tr>
<tr>
<td>0x26</td>
<td>BYTE</td>
<td>Volume Serial Number. A unique number that is created when you format the volume.</td>
</tr>
<tr>
<td>0x27</td>
<td>4 bytes</td>
<td>Volume Label.</td>
</tr>
<tr>
<td>0x28</td>
<td>11 bytes</td>
<td>System ID. Either FAT12 or FAT16, depending on the format of the disk.</td>
</tr>
</tbody>
</table>
The LSN of the boot block is 0. The information contained within the BPB in boot block can be used to calculate the LSN of the block from where the user data starts. It can be simply calculated by adding the number of reserved sector, sectors occupied by FAT copies * number of FAT copies and the the number of blocks reserved for root dir.

Inside a Boot Block
- Contains Code and Data
  jmp codepart
  OSName
  BIOS
  Parameter Block
  
  codepart:
  codepart:
  codepart:

- Boot Block executes at Booting time.

The above slide shows the location of BPB within the boot block. A jump instruction (near jump of 3 bytes size) is used to jump to the code part and skip the data part so that it is not interpreted as instructions by the processor.
Above is the dump of the boot block for a FAT 12 system. The contents of the BPB can be read from it the following slide shows the detail of the information obtained from the above BPB.

Logical drive: A
Size: 1 Mb (popularly 1 Mb)
Logical sectors: b40h = 2880
Bytes per sector: 512
Sectors per Cluster: 1
Cluster size: 512
File system: FAT12
Number of copies of FAT: 2
Sectors per FAT: 9
Start sector for FAT1: reserved sectors = 1
Start sector for FAT2: reserved sectors + size of FAT = 1 + 9 = 10
Root DIR Sector: reserved sectors + 2 * (size of FAT) = 1 + 2 * 9 = 19
Root DIR Entries: E0 = 224
Size of Root dir : 224 * 32 = 7168
Blocks occupied by root dir = 7168 / 512 = 14
2-nd Cluster Start Sector: root dir sectors +
    size of root dir in blocks = 19 + 14 = 33
Ending Cluster: 2880 - 33 / sector per cluster + 1
    = 2880/33 + 1 = 2848
Media Descriptor: F0
Heads: 2
Hidden sectors: 0
SerialVolumeID: 2A876CE1
Volume Label:

Following is another dump showing the boot block for a FAT 16 system.
Following is the detail of information read from the above dump which describes the volume in question.

Logical sectors: = 00065f5b = 417627  
Bytes per sector: 200h = 512  
Sectors per Cluster: 8  
Cluster size: 8*512 = 4096  
File system: FAT16  
Number of copies of FAT: 2  
Sectors per FAT: CCH = 204  
Start sector for FAT1: reserved blocks = 8  
Start sector for FAT2: reserved blocks + blocks per FAT = 8 + CCH = D4 = 212  
Root DIR Sector: reserved blocks + 2*(size of FAT) = 8 +2*CCH = 1A0 = 416  
Root DIR Entries: 200H = 512  
Size of Root Dir 512 * 32 = 16384 = 32 blocks

Size of Root Dir 512 * 32 = 16384 = 32 blocks  
2-nd Cluster Start Sector: root dir start blocks + blocks in root dir = 1A0 + 20H = 416 + 32 = 448  
Ending Cluster: ((logical blocks - start of user data blocks)/blocks per cluster) + 1 = (417627 - 448 )/8 + 1 = 52148  
Media Descriptor: F8

Heads: 255  
Hidden sectors: 63  
SerialVolumeID: 2CA5BC35  
Volume Label: NO NAME

Besides the LBA address a LSN address can also be used to address a block. If the LSN address is known the absread() function can be used to read a block and abswrite() can be used to write on it as described in the slide below where nsect is the number of sector to be read/written.
Reading/ Writing a Block

- `absread()` is used to read a block given its LSN
  
  ```c
  absread(int drive, int nsects, long lsec, void *buffer);
  ```

- `abswrite()` is used to write a block given its LSN
  
  ```c
  abswrite(int drive, int nsects, long lsec, void *buffer);
  ```
Lecture # 32

Besides the BPB another data structure can be used equivalently called the DPB (Drive parameter block). The operating system translates the information in BPB on disk into the DPB which is maintained main memory. This data structure can be accessed using the undocumented service 21H/32H. Its detail is shown in the slide below.

**Undocumented Services (INT 21H/32H)**

**On Entry:**
- AH = 32h
- DL – 0 for current Drive
  - 1 for A: Drive
  - 2 for B: Drive
  - 3 for C: Drive

**On Exit:**
- DS:BX = far address of DPB

The DPB contains the information shown in the table below. This information can be derived from the BPB but is placed in memory in the form of DPB.

**DPB (Drive Parameter Block)**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>BYTE</td>
<td>Drive number (00h = A:, 01h = B:, etc)</td>
</tr>
<tr>
<td>01h</td>
<td>BYTE</td>
<td>Unit number within device driver</td>
</tr>
<tr>
<td>02h</td>
<td>WORD</td>
<td>Bytes per sector</td>
</tr>
<tr>
<td>04h</td>
<td>BYTE</td>
<td>Highest sector number within a cluster</td>
</tr>
<tr>
<td>05h</td>
<td>BYTE</td>
<td>Shift count to convert clusters into sectors</td>
</tr>
<tr>
<td>06h</td>
<td>WORD</td>
<td>Number of reserved sectors at beginning of drive</td>
</tr>
<tr>
<td>08h</td>
<td>BYTE</td>
<td>Number of FAT’s</td>
</tr>
<tr>
<td>09h</td>
<td>WORD</td>
<td>Number of root directory entries</td>
</tr>
<tr>
<td>00h</td>
<td>WORD</td>
<td>Number of first sector containing user data</td>
</tr>
<tr>
<td>08h</td>
<td>WORD</td>
<td>Highest cluster number (number of data cluster +1)</td>
</tr>
</tbody>
</table>
The following code shows how the service 21H/32H is invoked and the registers in which it returns a value. It also shows the contents of the DPB by taking the dump at the location returned by the service for a FAT 12 volume (i.e. Floppy disk).
The details of the information read from the dump of the DPB are shown below.

```
Drive # = 0
unit # = 0
Bytes per sector = 0200H = 512 bytes
highest sec no within a cluster = 0
Shift count to convert sec to clust = 0
Reserved sectors at the begining of drive = 0001
FAT copies = 02
Root directory entries = E0 = 224
First sector containing user data = 21H = 33
Highest cluster number = 0b20 = 2848
Number of sectors per fat = 0009 = 9
Sector number of first directory = 0013 = 19
```

The following code shows how the service 21H/32H is invoked and the registers in which it returns a value. It also shows the contents of the DPB by taking the dump at the location returned by the service for a FAT 16 volume (i.e. hard disk partition smaller than 2 GB approx.).
The details of the information read from the dump of the DPB are shown below.

```
#include <bios.h>
#include <dos.h>

struct BPB
{
    unsigned int bytespersec;
    unsigned char secperclust;
    unsigned int reservedsecs;
    unsigned char fats;
    unsigned int rootdirents;
    unsigned int smallsecs;
    unsigned char media;
    unsigned int fatsecs;
    unsigned int secspertrack;
    unsigned int heads;
```

unsigned long int hiddensecs;
unsigned long int hugesecs;
unsigned char driveno;
unsigned char reserved;
unsigned char bootsignature;
unsigned long int volumeid;
unsigned char volumelabel[11];
unsigned char filesystem[8];

};
struct bootblock
{
    unsigned char jumpinstruction[3];
    unsigned char osname[8];
    struct BPB bpb;
    unsigned char code[448];
};

void nputs(char *p, int n)
{
    int i;
    for (i = 0; i < n; i++)
        putchar(p[i]);
}

void main(void)
{
    struct bootblock bb;
    clrscr();
    absread(0, 1, 0, &bb);
    printf("jump instruction = %x\n", bb.jumpinstruction);
    printf("OS = ");
    nputs(bb.osname, 8);
    puts("\n");
    printf("No of bytes per sector = %d\n", bb.bpb.bytespersec,
           bb.bpb.secsperclust, bb.bpb.reservedsecs);
    printf("No of FATs = %d\n", bb.bpb.fats,
           bb.bpb.roottreents, bb.bpb.smallssecs,
           bb.bpb.media, bb.bpb.fatsecs,
           bb.bpb.secspertrack, bb.bpb.heads,
           bb.bpb.hiddensecs, bb.bpb.hugesecs,
           bb.bpb.driver, bb.bpb.reserved,
           bb.bpb.bootsignature, bb.bpb.volumeid);
```c
puts (" Volume Name =");
puts (bb.bpb.volumelabel,11);
puts ("in");

puts (" File system =");
puts (bb.bpb.filesystem,8);
puts("in");
getch();
}
```

The above program creates a data structure in reflection of the BPB and reads the boot record of the volume using absread(). It extracts the data part of the boot block and displays all the values stored in it.

```c
#include <dos.h>
#include <bios.h>
struct DPB {
    unsigned char driveno;
    unsigned char unitno;
    unsigned int bytespersec;
    unsigned char highestsecinclust;
    unsigned char shiftcount;
    unsigned int reservedsecs;
    unsigned char fats;
    unsigned int rootentries;
    unsigned int firstuserdatasec;
    unsigned int highestclustnumber; //only for 16 and 12 bit FATs
    unsigned int sectsperfat;
    unsigned int firstdirsec;
    unsigned int ddheaderoff;
    unsigned int ddheaderseg;
};
```
unsigned char media;
unsigned char accessed;
unsigned int nextdpboff;
unsigned int nextdpbseg;
unsigned int searchstart;
unsigned int freeclust;
}

void main (void)
{
    struct DPB far *ptr;
    struct DPB dpb;
    clrscr();
    _asm push DS;
    _asm push BX;
    _AH=0x32;
    _DL=1;
    geninterrupt (0x21);
    ptr = (struct DPB far *)MK_FP(_DS,_BX);
    dpb=*ptr;
    _asm pop BX;
    _asm pop DS;
    printf("Drive No = %x\n",dpb.driveno);
    printf("Unit No = %x\n",dpb.unitno);
    printf("Bytes per sector = %d\n",dpb.bytespersec);
    printf("Highest sector number within a cluster =
           %d\n",dpb.highestsecinclust);
    printf("Shift Count = %d\n",dpb.shiftcount);
    printf("Reserved sectors = %d\n",dpb.reservedsecs);
    printf("Number of FATs = %d\n",dpb.fats);
    printf("Root entries = %d\n",dpb.rooteentries);
    printf("First User data sec = %d\n",dpb.firstuserdatasec);
    printf("Highest Cluster number =
           %d\n",dpb.highestclustnumber);
    printf("No of Sectors per FAT = %d\n",dpb.secsperfat);
    printf("First directory Sector = %d\n",dpb.firstdirsec);
    printf("DD header offset = %x\n",dpb.ddheaderoff);
    printf("DD header segment = %x\n",dpb.ddheaderseg);
    printf("Media ID= %d\n",dpb.media);
    printf("Disk accessed recently= %d\n",dpb.accessed);
    printf("Next DPB offset address = %d\n",dpb.nextdpboff);
    printf("Next DPB segment address =
           %d\n",dpb.nextdpbseg);
    printf("Point where to start the search for next cluster=
           %d\n",dpb.searchstart);
    printf("Free cluster= %d\n",dpb.freeclust);
    getch();
}

The above program is doing is the same using the DPB rather than BPB.
Lecture # 33

The Root Directory

The DOS directory structure is a Tree like structure. The top most level of the tree being the root directory. The root directory contains files and folders. Each folder can contains more files and folders and so on it continues recursively as shown by the slide.

File
- Is logically viewed as an organization of Data.
- Physically it can be a collection of clusters or blocks.
- O.S. needs to maintain information about the cluster numbers of which a file may be comprised of.

Control information about files are maintained in a data structure called the File control block (FCB). The FCB for each file is created and stored in the disk.
The root directory consists of FCBs for all the files and folders stored on the root directory. To obtain these FCBs, the portion on disk reserved for root directory can be read.
In the above two slides first the contents of DPB are read to find the start of the root directory. Using this block number the contents of root directory are read, as it can be seen they contain number of FCBs each containing information about a file within the directory.

The user data area is divided into clusters. The first cluster in user data area is numbered 2 in a FAT based systems. A cluster is not the same as block and also there are no system calls available which use the cluster number. All the system calls use the LSN address. If the cluster number is known it should be converted into LSN to access the blocks within the cluster. Moreover all the information about file management uses the cluster number.
rather than the LSN for simplicity and for the purpose of managing large disk space. So here we devise a formula to convert the cluster number into LSN.

The following memory dump extracts the starting cluster number from the FCB of a file and then converts the cluster number in sector to get its starting block.
The contents of the blocks/cluster at the start of file are then examined by loading the sectors within the first cluster to that file in the following slide. Here the contents of the file can be seen on the right side column.
Here is another example using a FAT 16 system.

![Another Example with FAT16 bit System]

```
DPB Dump

13AD:0100 mov ah,32
13AD:0102 int 21
13AD:0104

-AX=3200 BX=0000 CX=0000 DX=0000 SP=FFE8 BP=0000 SI=0000 DI=0000
-DS=13AD ES=13AD CS=13AD IP=0102
-13AD:0102 CD21          INT 21
-AX=3200 BX=13D2 CX=0000 DX=0000 SP=FFE8 BP=0000 SI=0000 DI=0000
-DS=00A7 ES=13AD CS=13AD IP=0104
-13AD:0104 D3E3          SHL BX,CL
-AX=3200 BX=0000 CX=0000 DX=0000 SP=FFE8 BP=0000 SI=0000 DI=0000
-DS=00A7 ES=13AD CS=13AD IP=0104
```

```
Lecture # 34

Here is another example which examines the contents of a file for a FAT 16 system. Firstly the DPB is read as shown in the following slide.

Once the DPB has been read the blocks reserved for root directory are determined and are then read to get the contents of the root directory.

The root directory contains a collection of FCBs. The FCB for the file in question is searched from where the first cluster of the file can be get.
After calculating the sector number for the cluster the contents of the file can be accessed by reading all the blocks within the cluster. In this way only the starting cluster will be read. If the file contains a number of cluster the subsequent clusters numbers within the file chain can be accessed from the FAT.
Larger File Contents
- Larger files would be comprised of numerous clusters.
- The first Cluster # can be read from FCB for rest of the Cluster, a chain is maintained within the FAT.

FAT12
- FAT is a simple table which contains cluster number of each file.
- FAT12 will have 12-bit wide entries and can have $2^{12}$ entries maximum.
- Although some of these entries may be reserved.
Above slides show how a cluster chain for a file is maintained in the FAT. The first cluster number is in the FCB. Subsequent clusters will be accessed from the FAT using the previous cluster number as index to look up into the FAT for the next cluster number.

A FAT theoretically will contain $2^n$ entries where $n$ is 12 for FAT 12 and 16 for FAT16. But all the entries are not used some of the entries are reserved following slide shows its detail.
Unused FAT Entries

- Reserved Entries = FF0H ~ FF6H
- EOF value = FF7H ~ FFFH
- First Two Clusters = 0,1
- Free Cluster = 0
- Max. range of Cluster # = 2 ~ FEFH
- Total # of Clusters of FAT12 = FEEH

Cluster Size Determination

tempo = size of disk / no. of entries in FAT (FEEH)
if ( temp > 32768)
    use higher FAT16 or FAT32
else
    choose the nearest value + temp greater than temp, which is a power of 2,
    Set this to be the Cluster size = bytes
    Size of Cluster in Blocks =
        Cluster size in Bytes / Bytes per Block
\}
No. of Entries of FAT =
    No. of Blocks in disk in User Data Area / Size of Cluster in Blocks.

There can various volume with various sizes with FAT12 or FAT16. The number of entries for FAT 12 or FAT16 are limited then the question arises how can a certain volume with moderate space and another volume with large space can be managed by the same FAT system. The answer is that the number of entries might be same but the size of cluster may be different. The cluster size can vary from 512 bytes to 32K in powers of 2 depending upon the volume size. The above slide shows how the cluster size and the exact number of required FAT entries can be determined.
Lecture # 35

There is no primitive data type of 12 bits. But the entries in 12 bit FAT are 12 bits wide. Three consecutive bytes in FAT 12 will contain two entries. The following slide shows an algorithm that can be used to extract these entries from a FAT 12 data structure.

### Selecting a 12-bit entry within FAT12

```plaintext
offset = cluster No * 3/2
temp = cluster No * 3%2
if (temp == 0)
{
    Then the entry is even, consider the word at this offset. Make a 12-bit value, by selecting the low Nibble of the high byte of this. Use this Nibble as the higher 4-bits. And use the low byte as the lower eight bits.
}
else
{
    The entry is odd, consider the word at this offset. Select the high Nibble of the low byte as lower 4-bits. And select high byte as the higher 8-bits.
}
```

### Example

```
184h * 3 / 2  =  246h
  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |
  |  |  |  |  |  |
  |  |  |  |  |
  |  |  |  |
  |  |  |
  |  |
  | |
  |

185h * 3 / 2 = 247h
  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |
  |  |  |  |  |  |
  |  |  |  |  |
  |  |  |  |
  |  |  |
  |  |
  |
```

Here a familiar operation has been performed. After reading the DPB the root directory is read to search for the entry of file CPAPER.TXT.
Following slide shows the dump for the FAT12 for the particular volume.

Contents of FAT

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>F0 FF FF D3 40 FF 6F 04 07 80 00 09 AD 00 08</td>
</tr>
<tr>
<td>1010</td>
<td>C0 0D 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1020</td>
<td>D0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1030</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1040</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1050</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1060</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1070</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1080</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1090</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
<tr>
<td>1100</td>
<td>C0 02 00 02 00 00 00 01 13 AD 02 15 AD</td>
</tr>
</tbody>
</table>

Using the root directory entry and the FAT contents the contents of the file can be accessed.
Cont...
-l 1000 0 b 1
-d 1000

13A6:1010 65 20 65 73 20 6E 6F 74-20 69 6E 63 6C 75 64 65 e is not include
13A6:1020 65 20 63 65 20 74 68 69 66-63 6C 75 64 65 20 68 65 d by #include, h
13A6:1030 72 20 65 76 74 20 6F 66-75 67 72 61 76 74 20 73 65 72 65 6E 65 6D 65 73 69 74 ve is included.
13A6:1040 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D r Error...

13A6:1050 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D

Cont...
-l 1000 0 b 1
-d 1000

13A6:1010 6E 67 20 70 6F 69 6E 74-65 6D 75 6C 61 74 69 6E 67 70 65 73 69 74 65 65 6D 65 73 ng point formats
13A6:1020 20 6E 6F 74 20 6C 6F 77 69 6E 67 3A 0D 0A 0D 0A 0D 0A 0D 0A 0D 0A not linked...Ab
13A6:1030 69 73 20 65 72 72 6F 72 20 6F 63 63 75 72 73 normal program...
13A6:1040 69 73 20 65 72 72 6F 72 20 74 68 65 20 66 6C 6F ating error occur
13A6:1050 69 73 20 65 72 72 6F 72 20 74 68 65 20 73 65 72 73 the following:
13A6:1060 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D

© Copyright Virtual University of Pakistan 270
Following slide shows the file organization in a FAT16 system. The starting cluster number is the FCB entry and the subsequent clusters can be determined by traversing the chain for that file in the FAT16 data structure.

Not all the entries in FAT16 will be used, the following shows which ones of the entry are reserved.
Unused FAT Entries
- Reserved Entries = FFF0H ~ FFF6H
- EOF value = FFF7H ~ FFFFH
- First Two Clusters = 0, 1
- Free Cluster = 0
- Max. range of Cluster # = 2 ~ FFEFH
- Total # of Clusters of FAT16 = FFEEH

The following slide shows the content of the DPB for a FAT16 volume. Same int 21h/32H is used to determine its address and the parameters can be determined by taking dump of the memory at the address returned.

Contents of DPB

Here the slide shows the contents of the root directory depicting the FCB of a file named CPAPER.TXT. The control information about this file can be read from the dump.
The above slides shows that the first cluster of the file is 0008. The following slide shows the contents of FAT. The FAT is loaded firstly in memory at the offset address 1000H. Each entry occupies 2 bytes. So the index 0008 will be located at 1000H + 0008*2 = 1010H. At 1010H, 0009 is stored which is the next cluster of the file, using 0009H as index we look up at 1012H, which stores 0055H, which means 0055H is the next cluster.

using 0055H as index we look up at 10AAH to find the next cluster number and so on.
Lecture # 36

Now converting all those cluster number in previous lecture into LSN we get the starting LSN for each cluster.

For cluster # 0009H and 55H

### Contents of FILE

```
-1 1000 5 1ff 8
-d 1000
```

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Contents</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0009H</td>
<td>65 65 66 20 64 65 6C 65-74 65 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6N</td>
<td>Content of FILE -l 1000 5 1ff 8 -d 1000</td>
</tr>
</tbody>
</table>

For cluster 56H and 57H

### Cont...

```
-1 1000 5 468 8
-d 1000
```

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Contents</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>56H</td>
<td>63 6F 6C 65 65 6E 20 64 65 6C 65-74 65 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6E 20 64 20 61 63 69 6N</td>
<td>Content of FILE -l 1000 5 468 8 -d 1000</td>
</tr>
</tbody>
</table>
Now let’s just analyse the contents of root directory of the same volume. If the DIR command is performed on the same volume its result will be as below. Note the entry for file named TEST.TXT

Now on the same volume the file TEST.TXT is deleted. Lets analyse the contents of the root directory now.

```
Volume in drive F is NEW VOLUME
Volume Serial Number is 2CA5-BC35
Directory of F:
22-09-05  03:00 PM             1,674 dPb1.txt
22-09-05  02:40 PM             1,631 dPb2.txt
28-09-05  10:55 AM            1,210 dirPb1.txt
28-09-05  10:55 AM <DIR>    second
28-09-05  10:55 AM <DIR>    third
28-09-05  01:44 PM             1,210 dPb5.txt
28-09-05  10:58 AM               248 tree.txt
28-09-05  02:25 PM            60,775 rootf.txt
28-09-05  02:24 PM             1,210 test.txt
28-09-05  02:28 PM               665 file.txt
29-09-05  01:05 PM        25,162 Qpaper.txt
29-09-05  01:13 PM            2,657 ffile.txt
29-09-05  02:49 PM            80,999 fdd.txt
29-09-05  03:02 PM                39 ffile.txt
29-09-05  03:06 PM            1,305 ffat.txt
29-09-05  03:52 PM            25,362 Cpaper.txt
13 File(s)        177,775 bytes
3 Dir(s)     213,278,720 bytes free
```

```
Volume in drive F is NEW VOLUME
Volume Serial Number is 2CA5-BC35
```
The entry for TEST.TXT still exists as can be seen from the next slide. The only difference that has occurred is that the first character has been replaced by a byte with the value 0xE5.

But when the DIR command execute on the same volume the file does not show.
Contents of Root Listing After Deleting

Volume in drive F is NEW VOLUME
Volume Serial Number is 2CA5-BC35

Directory of F:\

22-09-05  03:00 PM             1,674 dphl.txt
22-09-05  02:40 PM             1,631 dpb2.txt
28-09-05  10:55 AM  <DIR>     first
28-09-05  10:55 AM  <DIR>     second
28-09-05  10:55 AM  <DIR>     third
28-09-05  01:44 PM             1,210 drvpdf.txt
28-09-05  10:58 AM             248 tree.txt
28-09-05  02:25 PM             60,775 root.txt
28-09-05  02:28 PM             665 file.txt
29-09-05  01:05 PM             25,362 Qpaper.txt
29-09-05  01:13 PM             1,210 fdpb.txt
29-09-05  02:49 PM             80,999 fdd.txt
29-09-05  03:02 PM             1,305 ffat.txt
29-09-05  03:06 PM             2,657 ffile.txt
29-09-05  03:52 PM             940 dir.txt
29-09-05  03:54 PM               0 dird.txt
13 File(s)       178,676 bytes
3 Dir(s)     213,278,720 bytes free

Now lets see the contents of the file by converting the first cluster number in the FCB into LSN and taking its dump. We get the following slide.

Contents of File After Deleting
-1 1000 5 3ds l
-d 1000

The contents of the file are still there.
So we infer the following.

**Deleted Files**
- 0xE5 at the start of file entry is used to mark the file as deleted.
- The contents of file still remain on disk.
- The contents can be recovered by placing a valid file name, character in place of E5 and then recovering the chain of file in FAT.
- If somehow the clusters used by deleted file has been overwritten by some other file, it cannot be recovered.

Not only the file is marked for deletion but also the chain of its cluster in FAT is reclaimed by putting zeros in there place. This also indicate that these clusters are now free.

Now lets have some discussion about sub-directories. In the contents of the above given root directory notice an entry named SECOND. The attribute byte of this entry is 0x20 which indicates that it’s a directory, the size is 0 which shows that there is now user data in it, but even though the size 0 its has a first cluster which is 0x12. Converting 0x12 into LSN and then reading its contents we get the following dump. This shows that this cluster contains the FCBs for all the file and folders within this directory.
In the entries within SECOND we see an entry SUB2. Its starting cluster is 0017H. This value is converted into LSN and the contents read. The slide above also show the contents of SUB2.

Similarly the following slide shows the contents of SUB3 within SUB2 and the contents of SUB4 within SUB3.

The following slide shows the contents of SUB5 and also the contents of file MYFILE.TXT in SUB5.
In all the sub-directories one thing is worth noticing. The first two entries are the . and .. entries. These two are special entries as described in the slide below.

<table>
<thead>
<tr>
<th>The . and .. Directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd .</td>
</tr>
<tr>
<td>gives the current path</td>
</tr>
<tr>
<td>cd . .</td>
</tr>
<tr>
<td>goes one level backwards.</td>
</tr>
</tbody>
</table>

Notice the contents of SECOND directory. The . entry has the cluster number 0012H which is the cluster number for the SECOND directory and the .. entry has cluster number which indicates the higher level directory which is the root directory.

<table>
<thead>
<tr>
<th>. and .. Sub-Directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 1000 5 240 8</td>
</tr>
<tr>
<td>-d 1000</td>
</tr>
</tbody>
</table>

Also examine the contents of SUB2 directory the . directory has cluster number 0017h which is the cluster number for SUB2 and the .. entry has the cluster number 0012H which is the number of its parent directory SECOND.

Same can be observed for SUB3, SUB4, SUB5 or any other sub-directory in question.
So this how CD. command gives the current path and CD.. moves to the one higher level directory.
Lecture # 37

Lets now perform few more experiments to see how long file names are managed.
Windows can have long file names up to 255 characters. For this purpose a file is created with a long file name as shown in the slide below.

Following shows the result of DIR command on the same volume.

<table>
<thead>
<tr>
<th>Long File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume in drive H is NEW VOLUME</td>
</tr>
<tr>
<td>Volume Serial Number is 8033-3F79</td>
</tr>
</tbody>
</table>

Directory of H:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>09/14/2005</td>
<td>10:00 AM</td>
<td>&lt;DIR&gt;</td>
<td>New Folder</td>
</tr>
<tr>
<td>10/23/2005</td>
<td>11:20 AM</td>
<td></td>
<td>I am new file to experiment long file names.txt</td>
</tr>
</tbody>
</table>

1 File(s) 0 bytes
1 Dir(s) 409,944,064 bytes free
In the following slide the DPB of the volume is being read.

```
Drive Parameter Block

Drive Parameter Block

Drive Parameter Block

Using the information in DPB the root directory entries are read and are being shown in the slide below.

```

```
Directory Entry

Directory Entry

Directory Entry

```

© Copyright Virtual University of Pakistan 284
In the above slide it can be noticed that the long file name is also stored with the FCBs. Also the fragments of Unicode strings in the long file name forms a chain. The first byte in the chain will be 0x01, the first byte of the next fragment will be 0x02 and so on till the last fragment. If the last fragment is the nth fragment starting from 0 such that n is between 0 and 25 the first byte of the last fragment will be given as ASCII of ‘A’ plus n.
Now let's move our discussion to FAT32. In theory the major difference between FAT 16 and FAT 32 is of course the FAT size. FAT32 evidently will contain more entries and can hence manage a very large disk whereas FAT16 can manage a space of 2 GB maximum practically.

Following slide shows the structure of the BPB for FAT32. Clearly there would be some additional information required in FAT32 so the structure for BPB used in FAT32 is different.

<table>
<thead>
<tr>
<th>BPI_RsvnSecCnt</th>
<th>14</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of reserved sectors in the reserved region of the volume starting at the first sector of the volume. This field must not be 0. For FAT12 and FAT16 volumes, this value should never be anything other than 1. For FAT32 volumes, this value is typically 32. There is a lot of FAT code in the world “hard wired” to 1 reserved sector for FAT12 and FAT16 volumes and that doesn’t bother to check this field to make sure it is 1. Microsoft operating systems will properly support any non-zero value in this field.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Bits</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>BPB_NumFATs</td>
<td>16</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_RootEntCnt</td>
<td>17</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_TotSec16</td>
<td>19</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_Media</td>
<td>21</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_FATSz16</td>
<td>22</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_SecPerTrk</td>
<td>24</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_NumHeads</td>
<td>26</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_HiddSec</td>
<td>28</td>
<td>Word</td>
</tr>
<tr>
<td>BPB_TotSec32</td>
<td>32</td>
<td>Word</td>
</tr>
</tbody>
</table>
There can be different volumes with different volume sizes. The device driver for file handling would require knowing the FAT size. The following slide illustrates an algorithm that can be used to determine the FAT size in use after reading the BPB.
<table>
<thead>
<tr>
<th>Name</th>
<th>Offset (byte)</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPB_FATsz32</td>
<td>36</td>
<td>4</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This field is the FAT32 32-bit count of sectors occupied by ONE FAT. BPB_FATsz32 must be 0.</td>
</tr>
<tr>
<td>BPB_ExtFlags</td>
<td>40</td>
<td>2</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Bits 0-3 = Zero-based number of active FAT. Only valid if mirroring is disabled. Bits 4-6 = Reserved. Bit 7 = 0 means the FAT is mirrored at runtime into all FATs. = 1 means only one FAT is active; it is the one referenced in bits 0-3. Bits 8-15 = Reserved.</td>
</tr>
<tr>
<td>BPB_TSVers</td>
<td>42</td>
<td>2</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. High byte is major revision number. Low byte is minor revision number. This is the version number of the FAT32 volume. This supports the ability to extend the FAT32 media type in the future without worrying about old FAT32 drivers mounting the volume. This document defines the version as 0.0. If this field is non-zero, back-level Windows versions will not mount the volume. NOTE: Disk utilities should respect this field and not operate on volumes with a higher major or minor version number than that for which they were designed. FAT32 file system drivers must check this field and not mount the volume if it does not contain a version number that was defined at the time the driver was written.</td>
</tr>
<tr>
<td>BPB_RootChns</td>
<td>44</td>
<td>4</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This is set to the cluster number of the first cluster of the root directory, usually 2 but not required to be 2. NOTE: Disk utilities that change the location of the root directory should make every effort to place the first cluster of the root directory in the first non-bad cluster on the drive (i.e., in cluster 2, unless it’s marked bad). This is specified so that disk repair utilities can easily find the root directory if this field accidentally gets zeroed.</td>
</tr>
<tr>
<td>BPB_FSInfo</td>
<td>48</td>
<td>2</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Sector number of FSINFO structure in the reserved area of the FAT32 volume. Usually 1. NOTE: There will be a copy of the FSINFO structure in backupboot, but only the copy pointed to by this field will be kept up to date (i.e., both the primary and backup boot record will point to the same FSINFO sector).</td>
</tr>
<tr>
<td>BPB_BckBootSec</td>
<td>50</td>
<td>2</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. If non-zero, indicates the sector number is the reserved area of the volume of a copy of the boot record. Usually 6. No value other than 6 is recommended.</td>
</tr>
<tr>
<td>BPB_Reserved</td>
<td>52</td>
<td>12</td>
<td>This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Reserved for future expansion. Code that formats FAT32 volumes should always set all of the bytes of this field to 0.</td>
</tr>
<tr>
<td>BS_DrvNum</td>
<td>64</td>
<td>1</td>
<td>This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.</td>
</tr>
<tr>
<td>BS_RESERVED1</td>
<td>65</td>
<td>1</td>
<td>This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.</td>
</tr>
<tr>
<td>BS_BootSig</td>
<td>66</td>
<td>1</td>
<td>This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.</td>
</tr>
<tr>
<td>BS_VolID</td>
<td>67</td>
<td>4</td>
<td>This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.</td>
</tr>
<tr>
<td>BS_VolLab</td>
<td>71</td>
<td>11</td>
<td>This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.</td>
</tr>
<tr>
<td>BS_FiledsType</td>
<td>82</td>
<td>8</td>
<td>Always set to the string &quot;FAT32&quot;. Please see the note for this field in the FAT12/16 section earlier. This field has nothing to do with FAT type determination.</td>
</tr>
</tbody>
</table>
Fat Type Determination

```c
if(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
else
    FATSz = BPB_FATSz32;
if(BPB_TotSec16 != 0)
    TotSec = BPB_TotSec16;
else
    TotSec = BPB_TotSec32;
DataSec = TotSec - (BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
CountofClusters = DataSec / BPB_SecPerClus;
if(CountofClusters < 4085)
    /* Volume is FAT12 */
else if(CountofClusters < 65525)
    /* Volume is FAT16 */
else
    /* Volume is FAT32 */
```
Lecture # 38

Following slide shows how the chain of clusters is maintained in a FAT32 based system.
Fat32 Entry

- Each entry is of 32-bits size but only lower 28-bits are used.
- Higher 4-bits are not tempered.
- While reading higher 4-bits are ignored.
- While writing higher 4-bits are not changed.

FCB in FAT32 system has an enhanced meaning as shown by the slide below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset (byte)</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIR_Name</td>
<td>0</td>
<td>11</td>
<td>Short name.</td>
</tr>
<tr>
<td>DIR_Attr</td>
<td>11</td>
<td>1</td>
<td>File attributes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_READ_ONLY                  0x01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_HIDDEN                    0x02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_SYSTEM                    0x04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_VOLUME_ID                  0x08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_DIRECTORY                  0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_ARCHIVE                    0x20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_LONG_NAME                  0x80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ATTR_READ_ONLY, ATTR_HIDDEN, ATTR_SYSTEM, ATTR_VOLUME_ID</td>
</tr>
<tr>
<td>DIR_NTRes</td>
<td>12</td>
<td>1</td>
<td>Reserved for use by Windows NT. Set value to 0 when a file is created and never modified or looked at after that.</td>
</tr>
<tr>
<td>DIR_CrtTime</td>
<td>14</td>
<td>2</td>
<td>Time file was created.</td>
</tr>
<tr>
<td>DIR_CrtDate</td>
<td>16</td>
<td>2</td>
<td>Date file was created.</td>
</tr>
<tr>
<td>DIR_LastAccDate</td>
<td>18</td>
<td>2</td>
<td>Last access date. Note that there is no last access time, only a date. Thus the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.</td>
</tr>
<tr>
<td>DIR_FstClusH</td>
<td>20</td>
<td>2</td>
<td>High word of this entry’s first cluster number (always 0 for a FAT12 or FAT16 volume).</td>
</tr>
<tr>
<td>DIR_WrtTime</td>
<td>22</td>
<td>2</td>
<td>Time of last write. Note that file creation is considered a write.</td>
</tr>
<tr>
<td>DIR_WrtDate</td>
<td>24</td>
<td>2</td>
<td>Date of last write. Note that file creation is considered a write.</td>
</tr>
</tbody>
</table>

Anatomy of FAT32 based system differs from FAT16 based systems significantly as explained by the slide below.
Fat32 File System

- Reserved Blocks
  - No fixed space reserved for root directory.
  - FCB of root directory are saved in a cluster and the cluster # for root directory is saved in BPB as discussed earlier.

In reflection of the anatomy of FAT32 based system the method used to translate the cluster # into LSN also varies. The following formula is used for this purpose.

Starting Sector # for a Cluster

\[
\text{Starting Sector} = \text{Reserved Sect.} + \text{FatSize} \times \text{FatCopies} + (\text{cluster #} - 2) \times \text{size of cluster}
\]

Now we determine all the parameters in the above slide for a certain volume to translate a cluster number into LSN.
Reserved Blocks
Offset 14 = 0x0E = Reserved Sect. = 0x0024

Copies of FAT
Offset 16 = 0x10 = Count of FAT’S = 0x0002

Sector per FAT copy
Offset 36 = 0x24 = Sectors occupied by single FAT = 0x0000253E
Cluster # for Root Directory

Offset 44 = 0x2C = 0x0000 0002

Size of Cluster in Sectors

Offset 13 = 0x0D = 0x20 = 32 blocks

So using all this information we calculate the LSN for cluster number 2 as shown on the slide below for this particular volume.
Starting Sector for Cluster # 2

Starting Sector = Reserved Sect. + FatSize * FatCopies + (cluster # - 2) * size of cluster
= 0x0024 + 0x0000 253E * 0x0002 + (2 – 2) * 0x20
= 0x4AA0
= 19104D

To examine the contents of a file first a file is created whose contents are also shown in the slide.

Creation of a File on Root

and is stored as myfile.txt on the root directory.
Now we examine the contents of cluster number 2 which contains the root directory as already seen in the previous slides.
From the information from above slides the low and high words of the first cluster number is obtained and is shown in the slide below. The higher 4 bits of the cluster number should be ignored as discussed earlier.

**Cluster # within FCB**

Cluster # Low Word = 29C9  
Cluster # Hi Word = 0009H  
0009 29C9 & 0FFF FFFF = 009 29C9

Using the cluster number above the LSN for this cluster is determined as shown in the slide below.

**File Data Sectors**

Starting Sector = Reserved Sect. + FatSize * FatCopies + (cluster # - 2) * size of cluster  
= 0024 + 253E * 2 + (929C9 - 2) * 0x20  
= 4AA0 + 12538E0  
= 1258380  
= 19235712
The blocks within the cluster are read and their contents can be seen in the slide below.

In the FAT32 there is another special reserved block called FSInfo sector. The block contains some information required by the operating system while cluster allocation/deallocation to files. This information is also critical for FAT16 based systems. But in FAT12 and 16 this information is calculated when ever required. This calculation at the time of allocation is not feasible in FAT32 as the size of FAT32 is very large and such calculations will consume a lots of time, so to save time this information is stored in the FSInfo block and is updated at the time of allocation/deallocation.
<table>
<thead>
<tr>
<th>Name</th>
<th>Offset (byte)</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSL_LeadSig</td>
<td>0</td>
<td>4</td>
<td>Value 0x41615252. This lead signature is used to validate that this is in fact an FSI_INFO sector.</td>
</tr>
<tr>
<td>FSL_Reserved1</td>
<td>4</td>
<td>480</td>
<td>This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.</td>
</tr>
<tr>
<td>FSL_StructSig</td>
<td>484</td>
<td>4</td>
<td>Value 0x61417272. Another signature that is more localized in the sector to the location of the fields that are used.</td>
</tr>
<tr>
<td>FSL_Free_Count</td>
<td>488</td>
<td>4</td>
<td>Contains the last known free cluster count on the volume. If the value is 0xFFF0FFFF, then the free count is unknown and must be computed. Any other value can be used, but is not necessarily correct. It should be range checked at least to make sure it is &lt;= volume cluster count.</td>
</tr>
<tr>
<td>FSL_Next_Free</td>
<td>492</td>
<td>4</td>
<td>This is a hint for the FAT driver. It indicates the cluster number at which the driver should start looking for free clusters. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at cluster 2. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFF0FFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume.</td>
</tr>
<tr>
<td>FSI_Reserved2</td>
<td>496</td>
<td>12</td>
<td>This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.</td>
</tr>
<tr>
<td>FSI_TrailSig</td>
<td>508</td>
<td>4</td>
<td>Value 0xAAA55000. This trail signature is used to validate that this is in fact an FSI_INFO sector. Note that the high 2 bytes of this value—which go into the bytes at offsets 510 and 511—match the signature bytes used at the same offsets in sector 0.</td>
</tr>
</tbody>
</table>
Lecture # 39

The following slide shows the anatomy of an NTFS based system. The FAT and root directory has been replaced by the MFT. It will generally have two copies the other copy will be a mirror image of the original. Rests of the blocks are reserved for user data. In the middle of the volume is a copy of the first 16 MFT record which are very important to the system.
The following slides show the Boot sector structure for a NTFS based system.

### NTFS General Boot Sector Structure

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field Length</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>3 bytes</td>
<td>Jump Instruction</td>
</tr>
<tr>
<td>0x03</td>
<td>LONGLONG</td>
<td>OEM ID</td>
</tr>
<tr>
<td>0x0B</td>
<td>25 bytes</td>
<td>BPB</td>
</tr>
<tr>
<td>0x24</td>
<td>48 bytes</td>
<td>Extended BPB</td>
</tr>
<tr>
<td>0x54</td>
<td>426 bytes</td>
<td>Bootstrap Code</td>
</tr>
<tr>
<td>0x01FE</td>
<td>WORD</td>
<td>End of Sector Marker</td>
</tr>
</tbody>
</table>

### BPB of NTFS Boot Block

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field Length</th>
<th>Field Name</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>WORD</td>
<td>0x0002</td>
<td>Bytes Per Sector</td>
</tr>
<tr>
<td>0x00</td>
<td>BYTE</td>
<td>0x08</td>
<td>Sectors Per Cluster</td>
</tr>
<tr>
<td>0x0E</td>
<td>WORD</td>
<td>0x0000</td>
<td>Reserved Sectors</td>
</tr>
<tr>
<td>0x10</td>
<td>3 BYTES</td>
<td>0x00000000</td>
<td>always 0</td>
</tr>
<tr>
<td>0x13</td>
<td>WORD</td>
<td>0x0000</td>
<td>not used by NTFS</td>
</tr>
<tr>
<td>0x15</td>
<td>BYTE</td>
<td>0xFF</td>
<td>Media Descriptor</td>
</tr>
<tr>
<td>0x16</td>
<td>WORD</td>
<td>0x0000</td>
<td>always 0</td>
</tr>
<tr>
<td>0x18</td>
<td>WORD</td>
<td>0x3F00</td>
<td>Sectors Per Track</td>
</tr>
<tr>
<td>0x1A</td>
<td>WORD</td>
<td>0xFF00</td>
<td>Number Of Heads</td>
</tr>
<tr>
<td>0x1C</td>
<td>DWORD</td>
<td>0x3F000000</td>
<td>Hidden Sectors</td>
</tr>
<tr>
<td>0x20</td>
<td>DWORD</td>
<td>0x00000000</td>
<td>not used by NTFS</td>
</tr>
<tr>
<td>0x24</td>
<td>DWORD</td>
<td>0x00000000</td>
<td>not used by NTFS</td>
</tr>
<tr>
<td>0x28</td>
<td>LongLong</td>
<td>0x1AF57F000000000000</td>
<td>Total Sectors</td>
</tr>
<tr>
<td>0x30</td>
<td>LongLong</td>
<td>0x040000000000000000</td>
<td>Logical Cluster Number for the file $MFT</td>
</tr>
<tr>
<td>0x38</td>
<td>LongLong</td>
<td>0x54FF00000000000000</td>
<td>Logical Cluster Number for the file $MFTMIR</td>
</tr>
</tbody>
</table>
The following slide shows a sample of the boot block dump. The following slides depict various parameters placed in the Boot block.

Sample of NTFS Boot Block

<table>
<thead>
<tr>
<th>Physical Sector</th>
<th>Cylinder</th>
<th>Side</th>
<th>Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>05</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MFT File Cluster #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Sector</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>00000004</td>
</tr>
</tbody>
</table>

The first 16 entries of the MFT are reserved. Rests of the entries are used for user files. There is an entry for each file in the MFT. There can be difference in the way a file is managed depending upon the size of the file.
Following slide shows the detail about the first 16 system entries within the MFT.

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Information</td>
<td>Includes information such as timestamp and link count.</td>
</tr>
<tr>
<td>Attribute List</td>
<td>Lists the location of all attribute records that do not fit in the MFT record.</td>
</tr>
<tr>
<td>File Name</td>
<td>A repeatable attribute for both long and short file names. The long name of the file can be up to 255 Unicode characters. The short name is the 8.3, case-insensitive name for the file. Additional names, or hard links, required by POSIX can be included as additional file name attributes.</td>
</tr>
<tr>
<td>Security Descriptor</td>
<td>Describes who owns the file and who can access it.</td>
</tr>
<tr>
<td>Data</td>
<td>Contains file data. NTFS allows multiple data attributes per file. Each file typically has one unnamed data attribute. A file can also have one or more named data attributes, each using a particular syntax.</td>
</tr>
<tr>
<td>Object ID</td>
<td>A volume-unique file identifier. Used by the distributed link tracking service. Not all files have object identifiers.</td>
</tr>
<tr>
<td>Logged Tool Stream</td>
<td>Similar to a data stream, but operations are logged to the NTFS log file just like NTFS metadata changes. This is used by EFS.</td>
</tr>
<tr>
<td>Reparse Point</td>
<td>Used for volume mount points. They are also used by Installable File System (IFS) filter drivers to mark certain files as special to that driver.</td>
</tr>
<tr>
<td>Index Root</td>
<td>Used to implement folders and other indexes.</td>
</tr>
<tr>
<td>Index Allocation</td>
<td>Used to implement folders and other indexes.</td>
</tr>
<tr>
<td>Bitmap</td>
<td>Used to implement folders and other indexes.</td>
</tr>
<tr>
<td>Volume Information</td>
<td>Used only in the $Volume system file. Contains the volume version.</td>
</tr>
<tr>
<td>Volume Name</td>
<td>Used only in the $Volume system file. Contains the volume label.</td>
</tr>
</tbody>
</table>
# MFT System Entries

<table>
<thead>
<tr>
<th>System File</th>
<th>File Name</th>
<th>NFT Record</th>
<th>Purpose of the File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master file table</td>
<td>$Mft</td>
<td>0</td>
<td>Contains one base record for each file and folder on an NTFS volume. If the allocation information for a file or folder is too large to fit within a single record, other file records are allocated as well.</td>
</tr>
<tr>
<td>Master file table 2</td>
<td>$MftMirr</td>
<td>1</td>
<td>A duplicate image of the first four records of the MFT. This file guarantees access to the MFT in case of a single-sector failure.</td>
</tr>
<tr>
<td>Log file</td>
<td>$LogFile</td>
<td>2</td>
<td>Contains a list of transaction steps used for NTFS recovery. Log file size depends on the volume size and can be as large as 4 MB. It is used by Windows NT/2000 to restore consistency to NTFS after a system failure.</td>
</tr>
<tr>
<td>Volume</td>
<td>$Volume</td>
<td>3</td>
<td>Contains information about the volume, such as the volume label and the volume version.</td>
</tr>
<tr>
<td>Attribute definitions</td>
<td>$AttrDef</td>
<td>4</td>
<td>A table of attribute names, numbers, and descriptions.</td>
</tr>
<tr>
<td>Root file name index</td>
<td>$</td>
<td>5</td>
<td>The root folder.</td>
</tr>
<tr>
<td>Cluster bitmap</td>
<td>$Bitmap</td>
<td>6</td>
<td>A representation of the volume showing which clusters are in use.</td>
</tr>
<tr>
<td>Boot sector</td>
<td>$Boot</td>
<td>7</td>
<td>Includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable.</td>
</tr>
<tr>
<td>Bad cluster file</td>
<td>$BadClus</td>
<td>8</td>
<td>Contains bad clusters for the volume.</td>
</tr>
<tr>
<td>Security file</td>
<td>$Secure</td>
<td>9</td>
<td>Contains unique security descriptors for all files within a volume.</td>
</tr>
<tr>
<td>Uppercase table</td>
<td>$Uppcase</td>
<td>10</td>
<td>Converts lowercase characters to matching Unicode uppercase characters.</td>
</tr>
<tr>
<td>NTFS extension file</td>
<td>$Extend</td>
<td>11</td>
<td>Used for various optional extensions such as quotas, reparse point data, and object identifiers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12-15</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
Lecture # 40

Now in the following example a file is created and its entry is searched in the MFT. The following slide shows that the name of the file created is TEST.TXT.

This slide show the contents of the file created.

The first logical block is read to read the contents of the BPB in NTFS. Following shows the contents of boot block for this volume.
Following slides shows various parameters obtained from BPB.

**Sector Per Cluster = 0008**  **MFT Cluster # = 000c0000h=786432**
For NTFS simply the following formula will be used to translate the sector number into cluster number.

**Determining the Sector # from Cluster #**

Sector # = Cluster # * Sector Per Cluster

Following slide shows how the sector number for the MFT on this volume was calculated. The first block of MFT no this volume is 6291456.

**Disassembling the File**

MFT Cluster # * 8 = Sector

786432 * 8 = 6291456

6291520
From the block number 6291456 entries was searched for TEST.TXT and this file entry was found at the block number 6291520.

The above dump shows the file name as well as the contents of the file are stored in this entry. Has the file been larger it would not have been possible to store the content of the file in this entry so other clusters would have been used and there would indexes would have been kept in the entry.

As an exercise one can try to find out the sub folders and the contents of the files stored in it.
The following slides explain how the NTFS volume can be accessed in DOS. Normally it can not be accessed if the system has booted in DOS as the DOS device drivers do not understand NTFS.

**Accessing NTFS volume in DOS**
- NTFS volume can not be accessed in DOS using DOS based function like absread( ) etc.
- DOS device drivers does not understand the NTFS data structures like MFT etc.
- If NTFS volume is accessed in DOS, it will fire the error of Invalid Media.

**How to Access NTFS volume using BIOS Functions**
- If the system has booted in DOS then a NTFS volume can be accessed by an Indirect Method, using BIOS functions.
- This technique makes use of physical addresses.
- Sector can be accessed by converting their LSN into LBA address and then using the LBA address in extended BIOS functions to access the disk sectors.
Translating LSN to LBA

LBA = No. of Physical Blocks in other Partition + Hidden Blocks + LSN

• All this information can be retrieved from the Partition Table + Boot Block
# Lecture # 41

```c
#include <stdio.h>
#include <dos.h>
#include <bios.h>
#include <alloc.h>

typedef struct tagfcb
{
    unsigned char filename[8];
    unsigned char ext[3];
    unsigned char attrib;
    unsigned char reserved[10];
    unsigned int time, date;
    unsigned int firstcluster;
    unsigned long int size;
} FCB;

typedef struct tagBPB
{
    unsigned int bytespersec;
    unsigned char secperclust;
    unsigned int reservedsecs;
    unsigned char fats;
    unsigned int rootdirents;
    unsigned int smallsecs;
    unsigned char media;
    unsigned int fatsecs;
    unsigned int secspertrack;
    unsigned int heads;
    unsigned long int hiddensecs;
    unsigned long int hugesecs;
    unsigned char driveno;
    unsigned char reserved;
    unsigned char bootsignature;
    unsigned long int volumeid;
    unsigned char volumelabel[11];
    unsigned char filesystem[8];
} BPB;

struct bootblock
{
    unsigned char jumpinst[3];
    unsigned char osname[8];
    BPB bpb;
    unsigned char code[448];
};

DPB far * getdpb(int drive)
{
    DPB far *dpb=(DPB far *)0;
    _asm push ds
    _asm mov ah,0x32
    _asm mov dl,byte ptr drive;
    _asm mov dx,ds
    _asm int 0x21
    _asm pop ds
    _asm cmp al,0xff
    _asm je finish
    _asm mov word ptr dpb+2,dx
    _asm mov word ptr dpb,bx
    return dpb;
    finish:
        return ((DPB far *)(0));
}
```
```c
void main (void)
{
    unsigned char filename[9];
    struct bootblock bb;
    unsigned char ext[4];
    FCB * dirbuffer;
    unsigned int * FAT;
    DPB d;
    DPB far * dpbptr;
    int i,flag;
    unsigned int cluster;
    puts("Enter filename:");
    gets(filename);
    puts("Enter Extension");
    gets(ext);
    if   ((absread(0x05,1,0, &bb))==0)
    
    else{
    puts("Failure");
    exit(0);
    }
    FAT =malloc(512);
    dirbuffer=malloc((bb.bpb.rootdirents) * 32);
    absread(0x05,bb.bpb.fatsecs,bb.bpb.reservedsecs+1,FAT);
    absread(0x05,(bb.bpb.rootdirents*32)/bb.bpb.bytespersec ,bb.bpb.fatsecs*bb.bpb.fats+bb.bpb.reservedsecs ,dirbuffer);
    i = 0;    flag=0;
    while(i<bb.bpb.rootdirents)
    {
    if((strncmpi(filename,dirbuffer[i].filename,strlen(filename)))==0)
    {
    if ((strncmpi(ext,dirbuffer[i].ext,strlen(ext)))==0)
    {
    flag=1;
    cluster = dirbuffer[i].firstcluster;
    printf("First cluster = %x ",cluster);
    while (cluster < 0xFFF0)
    { absread(0x05,bb.bpb.reservedsecs+cluster/256,FAT);
        cluster = FAT[cluster%256];
        printf("Next Cluster is :%x",cluster);
    }
    if (flag ==1)
    break;
    i++;
    }else
    i++;
    }
    
    The above program uses the DPB to reach the clusters of a file. The getDPB() function gets the far address of the DPB. Using this address the drive parameters are used to determine the location of FAT and root directory. The file is firstly searched in the root directory through sequential search. If the file name and extension is found the first cluster number is used to look up into the FAT for subsequent clusters. The particular block containing the next cluster within the FAT is loaded and the entry is read, similarly the whole chain is traversed till the end of file is encountered.
```
Disk Utilities

Format
- Low Level Format
  -- sets the block size.
  -- sets the Initial values in the block.
  -- indexes the block for optimal usage.
  -- can be accomplished using BIOS routines for small disks or extended BIOS services for larger disks.

- Quick Format
  -- initializes the data structures for file management.
  -- initializes and sets the size of FAT, root directory etc, according to the drive size.
  -- initializes the data in boot block and places appropriate boot strap code for the boot block.

Disk Partitioning Software

- Write the code part of partition table to appropriately load the Boot Block of active partition in primary partition table.
- Places data in the partition table regarding primary and extended partitions.
- As per specification of the user assigns a appropriate size to primary and extended partition by modifying their data part.
Scan Disk
Surface Scan for Bad Sectors
• It attempts to write a block.
• After write it reads back the block contents.
• Performs the CRC test on data read back.
• If there is an error then the data on that block is not stable the cluster of that block should be marked bad.
• The cluster is marked bad by placing the appropriate code for bad cluster so that they may not be allocated to any file.

Lost Chains
• The disk scanning software may also look for lost chains.
• Lost chains are chains in FAT which apparently don’t belong to any file.
• They may occur due to some error in the system like power failure during deletion process.

Looking for Lost Chains
• For each file entry in the directory structure its chain in FAT is traversed.
• All the cluster in the file are marked.
• When done with all the files and folders, if some non-zero and non-reserved clusters are left then they belong to some lost chains.
• The lost chains are firstly discretely identified and then each chain can either be restored to newly named files or can be deleted.
Cross References

- If a cluster lies in more than one file chain, then it is said to be Cross Referenced.
- Cross references can pose great problems.
- Cross references can be detected easily by traversing through the chain of all files and marking the cluster # during traversal.
- If a cluster is referenced more than once then it indicates a cross reference.
- To solve the problem only one reference should be maintained.

Defragmenter

- Disk fragmentation is unwanted.
- Fragmentation means that clusters of a same file are not contiguously placed, rather they are far apart, increasing seek time hence access time.
- So it is desirable that files clusters may be placed contiguously, this can be done by compaction or defragmentation.
- Defragmentation Software reserves space for each file in contiguous block by moving the data in clusters and readjusting.
- As a result of defragmentation the FAT entries will change and data will move from one cluster to other localized cluster to reduce seek time.
- Defragmentation has high computation cost and thus cannot be performed frequently.

File Restoration

- FAT structure provides the possibility of recovering a file after deletion, if its clusters were contiguous and have not been over-written.
- DOS perform file deletion by placing 0xE5 at the first byte of it FCB entry and placing 0's (meaning available) in the entries for the file clusters in the FAT.
- Two task should be performed successfully to undelete a file
  - Replacing the 0xE5 entry in FCB by a valid file name character.
  - Placing the appropriate values in FAT for representation of file cluster chain.
- If any one of the above cannot be done then the file cannot be fully recovered.
Lecture # 42

Memory Management
- Understanding of the data structures and techniques used for memory management.
- Study of the overall memory areas used by operating system and applications.

The following slide shows the memory map of the first 1MB of RAM. The first 640KB is called conventional RAM and the higher 384KB is called system memory. Some of the memory areas are reserved for special purposes as described by the slide rest is user area where user application can reside.
In higher processors, the main memory may be greater than 1MB. In this slide it shows that the memory portion higher than 1MB is called extended memory and some unused portion in system memory is called the expanded memory.

**Expanded Memory**
- also called EMS
- can be accessed using a driver called EMM386.EXE
- this driver allows the use of unused memory within system memory.

**Extended Memory**
- also called XMS
- can be accessed by installing the driver HIMEM.SYS
- this driver enable the extended memory by shifting from Real to Protected Mode.
Dual Modes in Higher PCs
Higher PCs can operate in two modes
• REAL MODE
• PROTECTED MODE

Real Mode
• PCs initially boots up in Real Mode. It may be shifted to protected mode during the booting process using drivers like HIMEM.SYS
• Only first 1 MB of RAM can be accessed in Real Mode.
• The Real Mode address is a 20-bit address, stored and represented in the form of Segment : Offset
• OS like DOS has a memory management system in reflection of the Real Mode.

Protected Mode
• PC has to be shifted to Protected Mode if originally boots in Real Mode.
• In Protected Mode whole of the RAM is accessible that includes the Conventional, Expanded and Extended Memories.
• OS like Windows has a memory management system for Protected Mode.
• A privilege level can be assigned to a memory area restricting its access.
Memory Management in DOS
• DOS uses the conventional memory first 640 KB for its memory management.
• Additional 64 KB can be utilized by installing EMM386.EXE and additional 64 KB in the start of extended memory by installing HIMEM.SYS
• Smallest allocatable unit in DOS is a Paragraph, not a Byte.

Paragraph
• Whenever memory is to be allocated DOS allocates memory in form of Paragraph.
• A Paragraph can be understood from the following example consider two Physical Addresses
  1234 H : 0000 H
  1235 H : 0000 H
• Note there is a difference of 1 between the Segment address.
• Now lets calculate the Physical address
  12340 H
  12350 H
  Difference = 10 H
• A difference of 1 H in Segment address cause a difference of 10 H in Physical address.
• DOS loader assign a segment address whenever memory area is allocated, hence a change of 1 in Segment address will impart a difference of 16 D | 10 H in physical address.

Data Structures for Memory Management
• DOS makes use of various Data Structures for Memory Management:
  • MCB (Memory Control Block)
  • EB (Environment Block)
  • PSP (Program Segment Prefix)
**MCB or Arena Header**

- MCB is used to control an allocated block in memory.
- Every allocated block will have a MCB before the start of block.
- MCB is a 16-bytes large structure.

<table>
<thead>
<tr>
<th>Size</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>0</td>
<td>Contains ‘M’ if the MCB controls allocated memory and ‘Z’ if it controls free space.</td>
</tr>
<tr>
<td>Word</td>
<td>1</td>
<td>Contains the Segment address of the PSP and the program controlled by MCB.</td>
</tr>
<tr>
<td>Word</td>
<td>3</td>
<td>Contains number of Paragraphs controlled by the MCB.</td>
</tr>
<tr>
<td>Byte [11]</td>
<td>5</td>
<td>Reserved or contains the program name in case of higher versions of DOS.</td>
</tr>
</tbody>
</table>

**Environment Block**

- Contains Environment information like Environment variables and file paths for that program.

**PSP**

- is situated before the start of a process.
- contains control information like DTA (Disk Transfer Area) and command line parameters.
The following slide shows that two MCBs are allocated for each program typically. The first MCB controls the Environment Block the next MCB controls the PSP and the program. If this is the last program in memory then the MCB after the program has ‘Z’ in its first byte indicating that it is the last MCB in the chain.

Typical Memory Organization

All the MCB forms a chain. If the address of first MCB is known the segment address of next MCB can be determined by adding the number of paragraph controlled by MCB + ! into the segment address of the MCB. Same is true for all MCBs and hence the whole chain can be traversed.

How to Access the Start of Chain
• An documented service can be used to obtain the address of the first MCB.
• Service 21H/52H is used for this purpose.
• This service returns
  The address of DOS internal data structures in ES : BX
• 4-bytes behind the address returned lies the far address of the first MCB in memory.
• Using this address and hence traversing through the chain of MCBs using the information within MCBs.
The above slide shows how service 21H/52H is used to get the address of first MCB in memory.

In the following slide the dump of the first MCB is taken. ‘M’ in the first byte at the location read indicates the placement of MCB at this location. The address of next MCB can be calculated by adding the number of paragraphs controlled by MCB + 1 into the segment address. Using this method all the MCBs in memory are traversed till the last MCB with first byte ‘Z’ is encountered.
This program used the same method as discussed in previous lecture to get the address of first MCB, calculate the addresses of subsequent MCBs and traverse the MCBs to reach the last MCB.
Non-Contiguous Allocation

- Earlier Operating System like DOS has contiguous memory management system i.e. a program cannot be loaded in memory if a contiguous block of memory is not available to accommodate it.
- 80286 and higher processors support non-contiguous allocation.
- 80286 support Segmentation in Protected Mode, i.e. a process is subdivided into segment of variable size and each segment or few segments of the process can be placed anywhere in memory.
- 80386 and higher processors also support Paging, i.e. a Process may be divided into fixed size Pages and then only few pages may be loaded anywhere in memory for Process Execution.
- The key to such non-contiguous allocation systems is the addressing technique.

Address Translation

- In Protected Mode the direct method of seg * 10H + offset for Logical to Physical address translation is discarded and an indirect method is adopted.
Selectors
- In Protected Mode the Segment Registers are used as Selector.
- As the name suggest they are used to select a descriptor entry from some Descriptor Table.

Descriptor
- A Descriptor describes a Memory Segment by storing attributes related to a Memory Segment.
- Significant attributes of a Memory Segment can be its base (starting) address, its length or limit and its access rights.
The following slide shows the structure of descriptors of 80286 and 80386 processors

**80286 Descriptor**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Access Rights</td>
<td>Base (B15 - B0)</td>
<td>Limit (L15 - L0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**80386 Descriptor**

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base (B31 - B0)</td>
<td>Access Rights</td>
<td>Base (B23 - B16)</td>
<td>Limit (L15 - L0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Base (B31 - B0) contains the base address of Segment within the 4GB Physical space.
- Limit (L19 - L0) define the length of segment in units of bytes 
  - if G = 0 and in units of 4K (pages) if G = 1.
  - This allow the Segment to be of 1M if G = 0 and of 4G if G = 1.

**Descriptor**

- Access Right: contains the privilege level and other information depending upon the type of descriptor.
- G: the granularity bit selects the multiplier of 1 or 4K times the limit field. If G = 0 multiplier is 1; if G = 1 multiplier is 4K.
- D: selects the default registers size. If D = 0 the default register size is 16-bit, if D = 1 the size is 32-bit.
- AVL: the Operating System often uses this bit to indicate that the segment described by the Descriptor is available in memory.

**Segment Descriptor**

- If the Descriptor describes a memory segment then the Access Rights Byte will have the following meaning.

<table>
<thead>
<tr>
<th>P</th>
<th>DP</th>
<th>DPL</th>
<th>S</th>
<th>E</th>
<th>X</th>
<th>RW</th>
<th>A</th>
</tr>
</thead>
</table>

- P: Present bit, if P = 1 Segment is Present, if P = 0 Segment is not Present.
- DPL: Descriptor Privilege level 00 for highest and 11 for lowest. Low privilege level memory area cannot access a memory area with high privilege whereas vice versa is TRUE.
- S: Indicates data or code segment if S = 1 or a System Descriptor if S = 0.
- E: Executable, if E = 0 Then it’s a Data/Stack Segment, if E = 1 Then it’s a Code Segment.
- X: If E = 0 then X indicates the direction of expansion of the Segment.
  - If X = 0 Segment expands upwards (i.e. Data Segment)
  - If X = 1 Segment expands downwards (i.e. Stack Segment)
  - If E = 1 then X indicates if the privilege level of Code Segment is ignored (X = 0) or observed (X = 1).
- A: Accessed is set whenever the Segment is set.
System Descriptor & Access Byte

<table>
<thead>
<tr>
<th>Base (B24 – B31)</th>
<th>D</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Limit (B14 – B19)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>CPL</td>
<td>0</td>
<td>TYPE</td>
<td>Base (B15 – B0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lim (L15 – L0)</td>
<td></td>
</tr>
</tbody>
</table>

Descriptor Table

- GDT: Global Descriptor Table
- LDT: Local Descriptor Table
- IDT: Interrupt Descriptor Table
- GDT and LDT can have up to 8192 entries, each of 8-bytes width.
- IDT can have up to 256 entries.
Lecture # 44

Selector

- A Selector is called a Selector because it acts as an index into the Descriptor Table to select a GDT or LDT entry.

Segment Register

![Selector Diagram](image)

- 15: Selector Index
- 2 1 0: Selects a Descriptor Table entry from LDT or GDT

- TI = 0: Global
- TI = 1: Local

- Requested Privilege Level
  - 00: (Highest)
  - 11: (Lowest)

Address Translation in Protected Mode

- All the tables are maintained in Main Memory.
- Segment Registers are used as Selectors.
- The Descriptor Entry selected from the Descriptor Table is placed in a hidden cache to optimize address translation.
The hidden cache is illustrated in the slide below. The registers in dotted lines are hidden i.e. are not accessible to any application directly.

```
Address Translation in Protected Mode

* Whenever a Selector is assigned a new value, the hardware looks up into the Descriptor Table and loads the Base Address, Limit and Access Rights into the hidden cache.

* Whenever an instruction is issued the address referred is translating into Physical address using the effective Offset within the instruction and the Base Address in the corresponding Segment Cache, e.g.

  mov AX, [1234H]
  effective offset = 1234H
  base = base within the cache of DS
  abs. address = base +1234H

Or in instruction

  mov DL, [EBP]
  effective offset address = EBP
  base address = base address in cache of SS register
  abs. address = base address + EBP

* Hence the absolute address cannot be calculated directly from the Segment address value.
```
Control Register

- 80386 and above have 4 Control Registers CR0 ~ CR3.
- These Control Registers are used for conveying certain control information for Protected Mode Addressing and Co-Processors.
- Here we will illustrate only the least significant bit of CR0.

```
| CR0 | PE |
```

- The least significant bit of CR0 is PE-bit which can be set to enable Protected Mode Addressing and can be cleared to enter Real Mode.

Moving to Protected Mode

- Protected Mode can be entered by setting the PE bit of CR0, but before this some other initialization must be done. The following steps accomplish the switching from Real to Protected Mode correctly.
  1. Initialize the Interrupt Descriptor Table, so it contains valid Interrupt gates for at least the first 32 Interrupt type numbers. The IDT may contain up to 256, 8-byte interrupt gates defining all 256 interrupt types.
  2. Initialize the GDT, so it contains a NULL Descriptor, at Descriptor 0 and valid Descriptor for at least one Data and one Stack.
  3. Switch to Protected by setting the PE-bit in CR0.
  4. Perform a IntraSegment (near) JMP to flush the Internal Prefetch Queue.
  5. Load all the Data Selectors (Segment Registers) with their initial Selectors Values.
  6. The 80386 is now in Protected Mode.

Viruses

- Viruses are special program having ability to embed themselves in a system resources and there on propagate themselves.

**State of Viruses**

- Dormant State: A Virus in dormant state has embedded itself within and is observing system activities.
- Activation State: A Virus when activated would typically perform some unwanted tasks causing data loss. This state may triggered as result of some event.
- Infection State: A Virus is triggered into this state typically as a result of some disk operation. In this state, the Virus will infect some media or file in order to propagate itself.
Lecture # 45

Types of Viruses

- Partition Table Virus
- Boot Sector Virus
- File Viruses

How Partition Table Virus Works

- The Partition Table Code is executed at boot time to choose the Active Partition.
- Partition Table Viruses embed themselves in the Partition Table of the disk.
- If the Virus Code is large and cannot be accommodated in the Code Part of 512-bytes Partition Table block then it may also use other Physically Addressed Blocks to reside itself.
- Hence at Boot time when Partition Table is to be executed to select the Active Partition, the virus executes. The Virus when executed loads itself in the Memory, where it can not be reached by the OS and then executes the original Partition Table Code (stored in some other blocks after infection) so that the system may be booted properly.
- When the system boots the Virus will be resident in memory and will typically intercept 13H (the disk interrupt).
- Whenever a disk operation occurs int 13H occurs. The Virus on occurrence of 13H checks if removable media has been accessed through int 13H. If so then it will copy its code properly to the disk first Physical Block (and other blocks depending upon size of Virus Code). The removable disk is now infected.
- If the disk is now removed and is then used in some other system, the Hard Drive of this system will not be infected unless the system is booted from this disk. Because only on booting from this removable disk its first physical block will get the chance to be executed.

How Partition Table Virus Loads itself

- The transient part of Command.Com loads itself such that its last byte is loaded in the last byte of Conventional Memory. If somehow there is some Memory beyond Command.Com’s transient part it will not be accessible by DOS.
- At 40:13H a word contains the amount of KBs in Conventional Memory which is typically 640.
- If the value at 40:13H is somehow reduced to 638 the transient part of Command.Com will load itself such that its last byte is loaded at the last byte of 638KB mark in Conventional RAM.
- In this way last 2KB will be left unused by DOS. This amount of memory is used by the Virus according to its own size.
How Boot Sector Virus Works

- Boot Sector also works in almost the same pattern, the only difference is that it will embed itself within the Boot Block Code.

File Viruses

- Various Viruses embeds themselves in different executable files.
- Theoretically any file that can contain any executable code, a Virus can be embedded into it. i.e. .COM, .EXE are executable files so Viruses can be embedded into them, Plain Text Files, Plain Bitmap Files are pure data and cannot be executed so Viruses cannot be actively embedded into them, and even if they are somehow embedded they will never get a chance to execute itself.

COM File

- COM File is a mirror image of the program code. Its image on disk is as it is loaded into the memory.
- COM Files are single segment files in which both Code and Data resides.
- COM File will typically have a Three Bytes Near Jump Instruction as the first instruction in program which will transfer the execution to the Code Part of the Program.

```plaintext
jmp code
====
==== ;Data Part
====

code:
====
==== ;Code Part
```
How COM File Virus Infects Files

- A COM File Virus if resident may infect COM Files on execution.
- Typically COM File Virus will Interrupt 21H Service 4B. This Service is used to load a Program.
- Whenever a Program is to be Loaded int 21H Service # 4BH is used to Load a Program. The Virus if resident will check the parameters of this Service to get the file path. If the File is .COM File then the Virus appends itself to the file and tempers with the first 3-bytes of .COM File so that the execution branches to the Virus Code when the program is executed.

How COM Virus Loads Itself

- When a file is Loaded in Memory it will occupy a number of Paragraphs controlled by some MCB.
- If the file is infected the Virus is also loaded within the Memory Area allocated to the Program.
- In this case the Virus does not exist as an Independent Program as it does not have its own PSP. If the Program is terminated the Virus Code will also be unloaded with the program. The Virus will try to attain an Independent Status for which it needs to relocate itself and create its own PSP and MCB in Memory.
- When the program runs the Virus Code executes first. The Virus creates an MCB, defines a new PSP initializes the PSP and relocates itself, updates the last MCB, so that it can exist as an Individual Program, and then transfers the execution back to the Original Program Code.
- Now if the Original Program Terminates the Virus will still remain resident.
The following slide illustrates how a COM file virus relocates itself to make itself independent in memory.

**EXE File Viruses**

- The EXE File Viruses also works the same way in relocating themselves.
- The main difference in COM File and DOS EXE File is that the COM File starts its execution from the first instruction, whereas the entry point of execution in EXE File can be anywhere in the Program.
- The entry point in case of EXE File is tempered by the Virus which is stored in a 27-byte header in EXE File.
Detection

- Viruses can be detected by searching for their Signature in Memory or Executable Files.
- Signature is a binary subset of Virus Code. It is a part of Virus Code that is unique for that particular Virus only and hence can be used to identify the Virus.
- Signature for a Virus is selected by choosing a unique part of its Code. To find a Virus this Code should be searched in memory and in files. If a match is found then the system is infected.

Removal

Partition Table & Boot Sector Viruses

- Partition Table and Boot Sector Viruses can be removed by re-writing the Partition Table or Boot Sector Code.
- If the Virus is resident it may exhibit stealth i.e. prevent other programs from writing on Partition Table or Boot Sector by intercepting int 13H.
- In case it’s a stealth Virus the system should be booted from a clean disk will not give the Virus any chance to execute or load itself.

File Viruses

- If the Virus size is known Viruses can be removed easily from file.
- Firstly, the original value of first 3-bytes in case of COM File or the entry point in case of EXE should be restored.
- The appended portion of Virus can be removed by copying the contents of original file into a temporary file.

Original .Com temp

```
Program
Virus
```

- The Virus Code is not copied.
- The original file is then deleted and the temporary file is renamed as the original file.