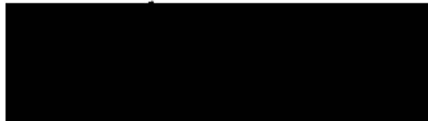
RT
344
c.1

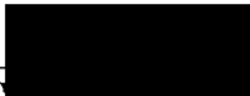
A DOS Client/Server Under TCP/IP

By
Tony Y. Fares



Dr. George E. Nasr, Chairperson
Assistant Professor of Electrical and Computer Engineering
Lebanese American University

Dr. Haidar Harmanani
Assistant Professor of Computer Science
Lebanese American University



Dr. [illegible]
Assistant Professor of Computer Science
Lebanese American University

Mr. Mounjed Mousallem
Assistant Professor of Computer Science
Lebanese American University

ABSTRACT

A DOS Client/Server Under TCP/IP

The client/server model has long been recognized, and new techniques were heavily applied during the recent years. However, considerable amount of problems, such as files representation, security, interworking, and drives mapping, still exist. The Thesis consists of creating a DOS client/server model, which uses TCP/IP as a communication protocol. Such a model can be simply installed in a UNIX network without the needs for protocol converter, and DOS is no more isolated from UNIX network. In addition, it will add to the client local drives, the server drives, thus, increasing the client capacity of files storing. The analysis includes client structure, server structure, client/server protocol, and hardware limitations. A source code is developed for both client and server to demonstrate the real work of this model. The experiment consists of loading the developed server program on a PC, and testing it in a UNIX LAN network, as well as, installing the developed client software, and establishing a simultaneous logging to both servers. As a result, the access to the server drives from several applications such as Microsoft Windows, MSD, and others was attained. The results of this study were successful proving general applicability on a wide range of hardware.

Tony Y. Fares

May 31, 1995

Acknowledgments

Many people deserve thanks for their assistance and help in the final production of my thesis. At the top of this list is Dr. George Nasr, my instructor and advisor, who provided me with his expert guidance, and all the needed support and assistance in solving the major problems I faced in my thesis.

I would like also to thank my teachers Dr. Keirouz,Dr. Harmanani, and Mr. Moussalem for their contributions and acceptance of being members of the judging committee.

Also, a special thanks to my friend Rachad Radi who provided me with all the TCP/IP tools.

I would like also to thank my parents Youssef, and Loris, and my sisters Mirna, Hala, Maha, and Rania who supported me to overcome all the obstacles I have faced in life, and I am greatful for what they gave me, because, without them I would't have been here.

Finally, thanks to my university LAU, and all its staff and faculty.

Table Of Contents

Acknowledgments ii
List of Figures v
List of Tables vii

Chapter 1 **Introduction** 1

Chapter 2 **TCP/IP and the New Library** 5
2.1 Internetworking 5
2.2 TCP/IP: the Internet protocols 5
2.2.1 TCP (Transmission Control Protocol) 6
2.2.2 IP Datagrams 6
2.3 OSI Model, Protocols, and Layering 7
2.4 Connections and Associations 9
2.5 New TCP APIs Developed Routines 9

Chapter 3 **Client/Server Concept** 14
3.1 Introduction 14
3.2 Technical Advantages 14
3.3 The Client/Server Model 15
3.4 Client/Server General Architecture 16
3.5 The Developed Client/Server Protocol 17
3.6 The Server Structure 18

Chapter 4 **DOS, BIOS, and Device Drivers** 20
4.1 Global View of DOS 20
4.2 Devices for DOS 21
4.3 The DOS Interrupts 22
4.4 DOS Device Management 22
4.5 Translating Service Calls to Device Driver Commands 23
4.6 The DOS Device Driver 23
4.6.1 Device Drivers for New Devices 23
4.6.2 New and Old Device Drivers 24
4.6.3 Overview of a Driver Program Structure 25
4.6.4 Communication of DOS with the Driver 26
4.7 Block and Character Devices 32
4.8 Device Driver Commands 33
4.9 Tracing a Request from Program to Device 35
4.10 An Overview of the Device Driver 37
4.10.1 The Device Header 37
4.10.2 The Device Attribute Field 37
4.10.3 The Device Header Name Field 40
4.10.4 DOS Command Processing 44
4.10.5 Exiting from the Device Driver 67

4.10.6 The Status Word for Unimplemented commands 68

Chapter 5 Client-Server Operations 70

5.1 Development Languages 70
5.1.1 Advantages of High-Level Languages 70
5.1.2 Disadvantages of High-Level Languages 71
5.2 A Closer Look at Tiny Model Program 73
5.3 Data Segment Preceding Code Segment 81
5.4 C Stack and Data 82
5.5 The C Run-Time Libraries 82
5.6 DOS Device Driver Header 83
5.7 DOS Device Driver Requests 84
5.8 DOS Device Driver Components 85
5.8.1 The Required Utilities 85
5.8.2 Segment Headers 85
5.8.3 Definitions 85
5.8.4 Global Data 86
5.8.5 C Environment 86
5.8.6 Commands 87
5.8.7 Ending Marker 87
5.8.8 Template Overview 88
5.9 Creating and Loading the Device Driver 88
5.10 Putting Client/Server into work 89
5.11 Layers Global View 90

Chapter 6 Conclusion 92

Appendix A Disk/Diskette Internals A-1
Appendix B Network Protocols & Communication B-1
Appendix C The OSI Layers C-1
Appendix D Glossary D-1

References R-1

List of Figures

Figure 1-1	Plugging DOS Server on a UNIX network	3
Figure 2-1	Layering in the Internet protocol suite	7
Figure 2-2	OSI model layers	8
Figure 2-3	Simplified 4-layer model connecting 2 systems	8
Figure 2-4	Socket system calls for connection-oriented protocol	12
Figure 3-1	Client/Server Model	16
Figure 4-1	The functional parts of DOS	21
Figure 4-2	Converting A simple service request to several possible disk device driver commands	24
Figure 4-3	Adding a new device driver to the list	25
Figure 4-4	The five basic parts of the device driver	26
Figure 4-5	DOS calling the device driver with a pointer to the request header	27
Figure 4-6	DOS calls the device driver twice.	29
Figure 4-7	The effect of three driver requests	30
Figure 4-8	DOS preparing to call the device driver for the first time	31
Figure 4-9	The STRATEGY procedure storing the address of the Request Header in local data storage	31
Figure 4-10	DOS processing the INTERRUPT routine	32
Figure 4-11	Block diagram of the paths taken to write a block of data to the disk	35
Figure 4-12	The five components of the Device Header	37
Figure A-1	The relative positions of the four components of a typical formatted disk	A-7
Figure A-2	The relationship between FAT entry and cluster	A-11

Figure A-3 The clusters used by myfile A-13

Figure A-4 The number of hidden sectors for the four partitions of a hard disk A-23

Figure A-5 DOS calculations of the start sectors for the FATs, the File Directory, and the User Data Area A-23

Figure A-6 The steps DOS takes to display the contents of the disk on a DIR command A-26

Figure A-7 The Initialization command requirement to return the address of the BIOS Parameter Block Table A-30

List of Tables

Table 4-1 The standard device names assigned by DOS 22

Table 4-2 The list of DOS interrupts (not BIOS) 22

Table 4-3 Definition of the Request Header that is passed to the device driver 28

Table 4-4 The list of commands for character-oriented devices 33
Table 4-5 The list of commands for block-oriented devices. 34

Table 4-6 The bit settings of the Attribute word 40

Table 4-7 Attribute bits setting that trigger device driver commands 41

Table 4-8 The various Attribute words found in various versions of DOS 42

Table 4-9 The DOS device driver commands, the DOS versions and the device types with which they work 44

Table 4-10 The Request Header for the Initialization command 45

Table 4-11 The DOS service that device drivers may use when processing the Initialization command 46

Table 4-12 The Request Header of The Media Check command 47

Table 4-13 The three values for the media change status word 48

Table 4-14 The Request Header for the Get BPB command 50

Table 4-15 The fields that comprise the BPB 50

Table 4-16 The Request Header for the IOCTL Input command 51
Table 4-17 The Request Header for the Input command 53

Table 4-18 The request Header for the Nondestructive Input command 54

Table 4-19 The Request Header for the Input Status command 54

Table 4-20 The Request Header for the Input flush command 55

Table 4-21 The Request Header for the Output command 56

Table 4-22 The request Header for the Output with Verify command 57

Table 4-23 The Request Header for the Output Status command 58

Table 4-24 The Request Header or the Output Flush command 59

Table 4-25 The Request Header for the IOCTL Output command 59

Table 4-26 The Request Header for the Device Open command 60

Table 4-27 The Request Header for the Device Close command 61

Table 4-28 The Request Header for the Removable Media command 61

Table 4-29 The Request Header for the Output Till Busy command 62

Table 4-30 The Request Header for commands 17 and 18 63

Table 4-31 The Request Header for the Generic IOCTL command 63

Table 4-32 The Major function codes for Generic I/O Control 64

Table 4-33 The Minor function codes for character devices 64

Table 4-34 The Minor function codes for the block devices 64

Table 4-35 The Request Header for the commands 20, 21, 22 65

Table 4-36 The Request Header for the Get Logical Device command 65

Table 4-37 The Request Header for the Set Logical Device command 66

Table 4-38 The Request Header for the IOCTL Query command 67

Table 4-39 The standard error codes for DOS device drivers 68

Table 4-40 The Request Header Status word for commands that are not implemented in device drivers 69

Table 5-1 The required tools for writing a DOS device driver 85

Table 5-2 Device driver components 88

Table 5-3 Commands exchange between client and server device drivers 90

Table 5-4	Client and server different layers	91
Table A-1	The amount of raw storage available for different types of disks	A-3
Table A-2	Some of the disk formats supported by DOS	A-5
Table A-3	The types of disks supported by the various versions of PC-DOS.	A-6
Table A-4	Disk sizes for other types of PCs using MS-DOS	A-6
Table A-5	The typical cluster sizes for different types of disks	A-9
Table A-6	The various FAT entries and what they mean	A-11
Table A-7	The number of File Directory entries and the number of directory sectors for each type of disk	A-14
Table A-8	The File Directory entry consists of eight fields	A-15
Table A-9	The various attribute bits that can exist for File Directory entries	A-15
Table A-10	How to decode the 2-byte time field	A-16
Table A-11	How to decode the 2-byte date field	A-16
Table A-12	How to interpret the start cluster number for the File	A-17
Table A-13	The 4-byte file-size field	A-17
Table A-14	The various calculations for determining the size of the FAT entries and the amount of overhead the disks can have	A-18
Table A-15	The fields that comprise the BIOS Parameter Block (BPB)	A-20
Table A-16	The various values for the media descriptor field	A-22
Table A-17	The typical values found in the vendor identification field and the BIOS Parameter Block for a 40Mb hard disk	A-25
Table A-18	The typical called DOS makes to the disk device driver in order to process the DIR command on a newly formatted disk	A-27

Table A-19 All of the applicable commands for block device drivers A-29

Table C-1 OSI Layers C-1

Chapter 1

Introduction

Software Engineers, developers, and designers have contributed to, and witnessed the growth and development of computers, from a simple punched card machine to a more complex and powerful machine with multiple processors. After a long period of interacting with these standalone machines, exploring every part of them, and developing different applications that range from the most complex (operating system, device drivers, compilers, etc.) to the simplest accounting program, computer users started to look for new concepts, wanting to go beyond the limitations of their machines. The new concepts were applied to wide range of networks starting from a single one connecting two PCs through a modem, and ending with a most complex network, such as Internet, connecting thousands of computers around the world.

A network means connecting computers together using various electronic techniques. A network can be as simple as connecting two computers, or more complex such as connecting thousands of computers in the world. The benefits of being connected to a network are:

1- Exchanging electronic mail with users on other computers. At this time one can communicate with others using special lines which supports voice, data, and image simultaneously.

2- Exchanging files between users, instead of copying files into a diskette and mailing it, which takes usually much more time than just simply forward it across the network.

3- Sharing peripheral devices, such as magnetic tape, hard disks, and printers, which are connected to a server, rather then buying such devices for each individual workstation.

4- Executing a program on a different machine. There are cases where some of the programs are to be run on other computers which are more powerful and more suited for these special applications. Also, a single network copy of a program will usually cost less and take less time to install it on several workstations.

5- The remote login features enabling users on different workstations to login into an account located on a remote machine, provided one has the right to access the account. It also allows the connection of a diskless machine to the network, and letting it boot from a remote server.

Designing a client-server model is not a new topic, several applications and studies are being done to create a perfect model. Some of these applications are:

(a) Novell Netware is the leader and the pioneer in DOS client/server architecture. Reliable and fast Novell versions, 2.2, 3.11, 3.12, 4.0, and 4.1 were developed. Novell Server provides several services such as, print services, file services, login services, and uses the IPX(Interwork Packet eXchange) as a communication protocol. To plug this Novell server on a network using TCP/IP as a communication protocol, a router is needed to convert IPX packets into TCP/IP packets and vice versa. This conversation feature was only implemented in Novell Netware versions 3.12 and higher[10].

(b) TFTP (Trivial File Transfer Protocol) is a simple method of transferring files between two systems using UDP. It does not provide many services such as directory listing, user authentication, etc. The only service provided by TFTP is the ability to send and receive files between a client process and a server process on a UNIX operating system machine. TFTP can be used to bootstrap a workstation on a LAN, since it is simple enough to implement in read-only memory. A protocol to do this is given in RFC 906 [24] and RFC 951 [23].

(c) NFS (Network File System) was developed by the computer vendor Sun Microsystems and introduced in late 1984. The NFS allows the file systems of remote computer systems to appear as if they were attached to the user's own local computer. It works on different operating systems such as DOS, UNIX, etc. It uses UDP (User Datagram Protocol) as a transport protocol [22].

Looking at the above proposed solutions, it is apparent that NFS succeded in mapping the server drives in multiple operating systems, but failed in enabling clients to access other server's resources, creating user accounts, and granting privileges. Novell Netware added most of the features that were lacking in NFS, but it did not take into consideration sharing new devices attached to the server such as optical drives. It is also, a fixed model that contains a pre-defined options list, and it uses a well amount of the local storage device. TFTP is just a simple file transfer that does not have the client server architecture.

The objective of this study is to create a DOS server which uses TCP/IP directly as a communication protocol instead of IPX, thus eliminating the usage of a router. This DOS server can be plugged in a UNIX network without any protocol conversion operation since both servers use TCP/IP protocol. Therefore, one can login to both UNIX and DOS servers and access files on both at the same time from any individual workstation on the network. In addition, the server drives are mapped into the clients drives, so a user can access these network drives as if they were local. It is model easy to be understood and modified by users who wish to add new devices to a server. The client and server source code are easy to understand and execute, in addition, they use a small space on the hard disk, and memory at loading time.

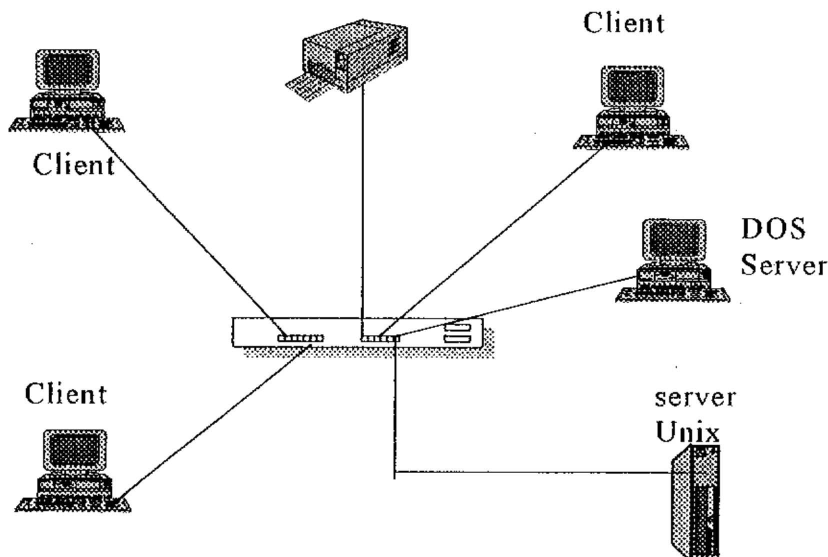


Figure 1-1: Plugging DOS Server on a UNIX network

In this thesis, a server program SERVER.EXE is developed to handle requests from multiple workstations (such as accessing the hard disk) and returns the result for each waiting workstation. It is a simple server model that is easily upgraded. The SERVER.EXE programs shows the different stages that a client should go through in order to access the destination device.

A client device driver DOS_DRVR.SYS is developed to access the server drives, and a login program LOGIN.EXE is also developed to establish connection to the server. The main function of DOS_DRVR.SYS is to add a logical drives to each workstation loading the designed client device driver, and to map the server physical drives to these logical drives. Whenever an access to these drives is needed, the developed driver sends the desired request across the network to the server, which handles this request, and returns the answer back to the device driver.

Since, the details of error handling resulting from packets sending or receiving, collisions, and traffic are managed by TCP/IP[14], only the application and presentation layers of the OSI model need to be investigated.

TCP/IP must be loaded on clients and DOS-Server, and no two TCP addresses must be the same on the same network, since a network error will occur. The DOS-Server machine runs the program SERVER.EXE which prepares the server for any expected connection. The DOS_DRVR.SYS is loaded in config.sys and LOGIN.EXE is run so, a complete access to the server's drives is obtained.

The first version of DOS_drvr.sys used 26k of lower memory. The improved version used 20k of lower memory. The TCP/IP development kit functions were not used since they can't be utilized from inside the device driver because they are calling interrupt 21h. As a substitute, a library called TCPCOM.LIB is developed to handle the required TCP/IP functions.

The thesis organization is as follows. Chapter 2 gives a brief introduction about TCP/IP, OSI representation, and describes the new developed TCP APIs library. In chapter 3, an outline is drawn about the developed client/server model, and an explanation of the used client-server protocol is discussed. Chapter 4, contains a detailed description about the client components such as, device driver structure,

commands , error control, and DOS device driver protocol. In chapter 5, we describe the needed tools, the language used, problems encountered and solutions provided. Also, the DOS client/server is designed using the various components discussed in previous chapters. Finally, a conclusion is presented in chapter 6.

Chapter 2

TCP/IP and The New Library

2.1 Internetworking

A computer network is a communication system for connecting end-systems. The end-systems are often referred to as hosts. The hosts can range in size from small microcomputers to the largest super computers. Some hosts on a computer network are dedicated systems, such as print servers or file servers, without any capabilities for interactive users. Other hosts might be single-user personal computers, while others might be general purpose time-sharing systems.

A local Area Network, or LAN, connects computer systems that are close together typically within a single building, but possibly up to few kilometers apart. Popular technologies today for LANs are Ethernet and Token Ring. LANs typically operates at high speeds, for instance, Ethernet operates at 10Mbps. Newer LAN technologies such as FDDI (Fiber Distributed Data Interface), use fiber optics and have a data rate of 100Mbps. Each computer on a LAN has an interface card of some forms that connects it to the actual network hardware. Additional information about different protocols are provided in Appendix B.

2.2 TCP/IP: the Internet protocols

During the late 1960s and the 1970s the Advanced Research Projects Agency (ARPA) of the Department of Defense (DoD) sponsored the development of the ARPANET [21]. The ARPANET included military, university, and research sites, and was used to support computer science and military research projects. ARPA is now called DARPA, with the first letter of the acronym standing for "Defense". In 1984 the DoD split the ARPANET into two networks, the ARPANET for experimental research, and the MILNET for military use. In the early 1980s a new family of protocols was specified as the standard for the ARPANET and associated DoD networks. Although the accurate name for this family of protocols is the "DARPA Internet protocol suite", it is commonly referred to as TCP/IP protocol suite, or just TCP/IP [20].

In 1987 the National Science Foundation (NSF) funded a network that connects the six national super computer centers together. This network is called the NFSNET. Physically this network connects 13 sites using high-speed leased phones lines and this is called the NFSNET backbone. About eight more backbone nodes are currently planned. Additionally the NSF has funded about a dozen regional networks

that span almost every state. These regional networks are connected to the NFSNET backbone, and the NFSNET backbone is also connected to the DARPA Internet. The NFSNET backbone and the regional networks all use TCP/IP protocol suite.

There are several interesting points about TCP/IP.

- It is not vendor-specific.
- It has been implemented on everything from personal computers to the largest supercomputers.
- It is used for both LANs and WANs.
- It is used by many different government agencies and commercial sites, not just DARPA-funded research projects.

The DARPA-funded research has led to the interconnection of many different individual networks into what appears as single large network or simply an internet. It is important to realize that while sites on the Internet use the TCP/IP protocols, many other nongovermental organizations have established their own internets using the same TCP/IP protocols. At one extreme an Internet using TCP/IP connects more than 150,000 computers throughout the United States, Europe and Asia, and at the other extreme a single network consisting of only two personal computers in the same room connected by an Ethernet, uses the same TCP/IP protocols[21].

2.2.1 TCP (Transmission Control Protocol)

It is a connection oriented protocol, that provides a reliable, full duplex, byte stream for a user process. It also provides sequencing, error control, and flow control. It is an end-to-end protocol, the TCP layer in the TCP/IP suite calculates and stores a checksum in the TCP message header which is then checked by the receiving TCP layer. TCP buffers store data internally and then pass it to next lower layer for transmission to the other end[8]. It also, allows flushing buffered data and sending it to the other end. This process is called the push process. Also, it allows unlimited amount of out-of-band data routing to be used, however, this is typically used only by network administrators for testing only. TCP/IP protocol suite uses a 32-bit integer to specify a combined network ID and host, and uses 16 bit-port numbers to identify specific user processes. The big endian representation for both the 16-bit integers and 32-bit integers is used[6].

Figure 2-1 shows the user process using TCP protocol, which in turn uses IP(Internet Protocol), to exchange data with other host process. The Internet protocol uses the hardware interface to connect to the network.

2.2.2 IP Datagrams

The IP layer provides a connectionless and reliable delivery system. It is connectionless because it considers each IP datagram independent of all others. Any association between datagrams must be provided by the upper layer. Every IP datagram contains the source address and the destination address so that each datagram can be delivered and routed independently.

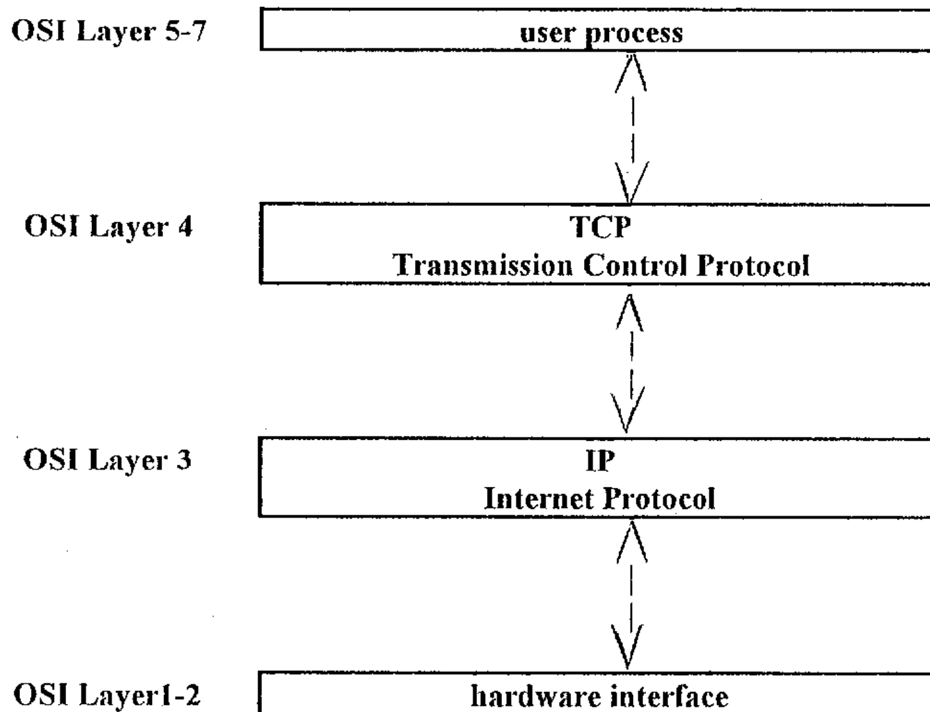


Figure 2-1: Layering in the Internet protocol suite

The IP layer computes and verifies a checksum that covers its own 20-byte header (that contains, for example, the source and destination addresses). This allows it to verify the fields that it needs to examine and process. But if an IP header is found in error, it is discarded, with the assumption that a higher layer protocol will retransmit the packet.

In Gateways, it is the IP layer that handles routing through the internet. The IP layer is also responsible for fragmentation. For example, if a gateway receives an IP datagram that it too large to transmit across the next network, the IP module breaksup the datagram into fragments and sends each fragment as an IP packet[15]. An IP packet can be fragmented into smaller IP packets. When fragmentation does occur, the IP layer duplicates the source address and the destination address into each IP packet, so the resulting IP packets can be delivered independently of each other. The fragments are reassembled into an IP datagram only when they reach their final destination. If any of the fragments are lost or discarded, the entire datagram is discarded by the destination host.

The IP provides an elementary form of flow control. When IP packets arrive at a host or gateway so fast they are discarded. In this case the IP module sends an ICMP source quench message to the original source informing the system that the data is arriving too fast.

2.3 OSI Model, protocols, and layering

The computer in a network uses well-defined protocols to communicate. A protocol is a set of rules and conventions between the communicating participants.

Since these protocols can be complex, they are designed in layers to make their implementation more manageable. Figure 2-2 shows a description of the 7 layers.
The model developed between 1977 and 1984, is a guide, not a specification[19]. It provides a framework in which standards can be developed for services and protocols at each layer. Indeed, the networks protocols (TCP/IP, XNS, and SNA) were developed before OSI model. Realize that no network is implemented exactly as the OSI model shows.

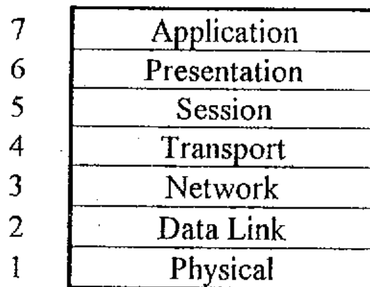


Figure 2-2: OSI model layers.

One advantage of layering is to provide well-defined interfaces between the layers, so that a change in one layer does not affect an adjacent layer. It is important to understand that protocols exist at each layer. A simplified 4-layer OSI model is shown in Fig. 2-3.

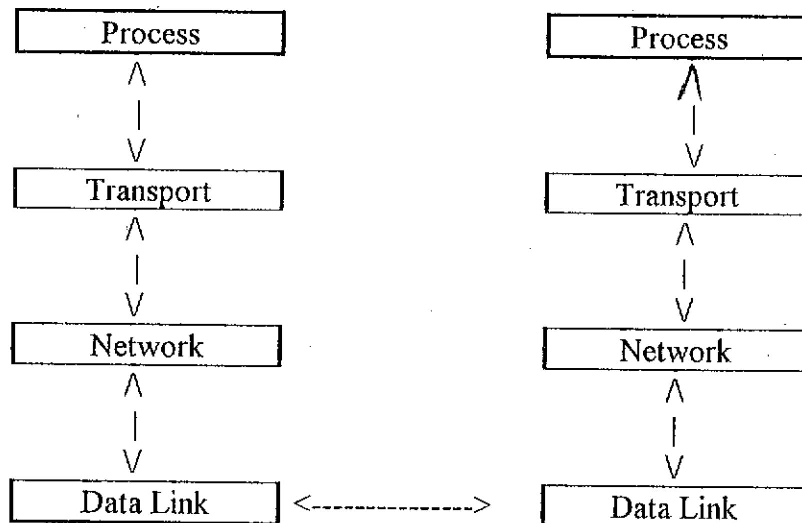


Figure 2-3: Simplified 4-layer model connecting 2 systems.

The layers that defines a protocol suite are the two boxes called the transport layer and the network layer. The multiple layers that define the network and hardware characteristics (Ethernet, Token Ring, etc.) are grouped together into the data-link layer. Application programs exist at the process layer where most of the user programs are run.

2.4 Connections and Associations

The term connection is used to define the communication link between two processes. The term association is used for 4-tuple that completely specifies the two processes that make up a connection:

{protocol, local_addr, base_port_addr, foreign_addr}

The local_addr and foreign_addr specify the network ID of the local host and foreign host. A typical format uses four integers from 0 to 255 separated by dots e.g. 255.255.66.1. The base_port_addr is the address of the logical port which is used to make the connection.

2.5 New TCP APIs Developed Routines

The two programs that are communicating are called the transport endpoints by TLI. TLI is a set of routines that provides the interface between the user process and the transport provider(TCP). Figure 2.4 contains a description of the TLI functions usage between the server and each workstation.

Calling the default TCP TLI routines(socket( ), connect( ), read( ), ...) from inside the device driver caused some serious problems. The device driver was debugged using some utilities developed with Borland C. It was found that whenever a request from DOS is issued to the device driver, instructions will be normally executed until a TCP/IP default function call is reached. These functions caused the PC to be rebooted, the program was traced, and after examining the assembly code of the socket( ) function, it was found that it is calling interrupt 21h, which is prohibited, because DOS is not reentrant. To get around this problem, and to make sure that interrupt 21h is never used in the device driver, under study, a new communication library containing all the previous listed functions is created, and this library is called **tcpcom.lib**. The first function in this library which retrieves TCP interrupt routine is GetNoIt( ),
where NoItMin, NoItMax represent the minimum and maximum interrupt number to search among for TCP/IP service routine. NoItMin and NoItMax were set equal to 60 and 80 respectively. TCP/IP signature, TCPTSR, was used for the signature field. The offset between the beginning of the ISR and the beginning of the signature is 3. Whenever TCP/IP TSR is loaded, its service routine address will be obtained, and modified communication functions will be used. Below is the code of GetNoIt( ):

```
int GetNoIt(int NoItMin,int NoItMax,char *Signature,long offset)
/*-----------------------------------------------------------*/
{
  char far *lpReel;
  char far *lpProt;
  char Found = 0;
  int  NoIt ;
```

```
  NoIt = NoItMin ;
  while ((!Found) && (NoIt<=NoItMax)) {
          lpReel = MK_FP(0, NoIt*4);
          lpProt= (char far *)(* ((long far *)lpReel));
          lpProt += offset;
          if (_fmemcmp(lpProt,Signature,strlen(Signature)) == 0)
                Found = 1 ;
          else
                NoIt++ ;
  } /* endwhile */
  if (!Found)
          NoIt = -1 ;

  return(NoIt);
}
```

After retrieving the TCP ISR number, a TCPCOM.LIB is created, and contains similar functions to the default TLI routines. An **s** character is added to the beginning of each function name to follow the same TLIs naming conventions. Below, is a description about the TLIs functions, and an example of the new developed TLIs functions is shown in
ssocket( ), for more information about others functions refer to TCPCOM.LIB found on the diskette.

•ssocket ( )
To do network I/O, the first thing a process must do is the socket system call. The TCPIP_INT argument is the integer value returned by GetNoIt( ) function:

```
int ssocket(int TCPIP_INT)
{
  struct REGPACK reg;

   reg.r.ax = NET_GETDESC;
   intr (TCPIP_INT, &reg);
   if ( reg.r_flags & 1);
        return -1;
   return reg.r_ax;
}
```

The ssocket returns a small integer value, similar to a file descriptor.

•sbind( )
The sbind call assigns a name to an unnamed socket

int sbind(int socket, struct addr *addr, int addrlen);

The second argument is a pointer to a protocol-specific address and the third argument is the size of this address structure. There are three uses of sbind:

1- Servers register their well-known address with the system. Both connection-oriented and connectionless servers need to do this before accepting client requests.
2- A client can register a specific address for itself.
3- A connectionless client needs to assure that the system assigns it some unique address, so that the other end (the server) has a valid return address to send its responses to.

•sconnect( )
A client process connects a socket descriptor following the ssocket system call to establish a connection with a server.

int sconnect(int socketfd, int type,struct addr *addr, int TCPIP_INT);

The socketfd is a socket descriptor that was returned by the ssocket system call. The second and the third arguments are a type of socket, and the third one is a pointer to a socket address. The fourth argument is TCP_IP interrupt number.
The sconnect( ) results in the actual establishment of a connection between the local system and the foreign system. Messages are typically exchanged between the two systems, and specific parameters relating to the conversation might be agreed on (buffer sizes, amount of data to exchange between acknowledgments, etc.). In this case the sconnect() does not return until the connection is established, or an error is returned to the process.

•slisten( )
This function is used by a connection-oriented server to indicate that it is willing to receive connections.

int slisten(int socketfd, int type, struct addr * addr, int TCPIP_INT);

slisten( ) takes the same arguments as sconnect( ), and is executed after both ssocket() and sbind( ).

•saccept( )
After a connection-oriented server executes the slisten( ), an actual connection from some client process is waited for by having the server executed saccept( ) function call.

int saccept(int socketfd, int type, struct addr * addr,int TCPIP_INT);

saccept( ) takes the first connection request on the queue and creates another socket with the same properties as socketfd. If there are no connection requests pending, this call blocks the caller until one arrives.

saccept( ) returns a new socket descriptor.

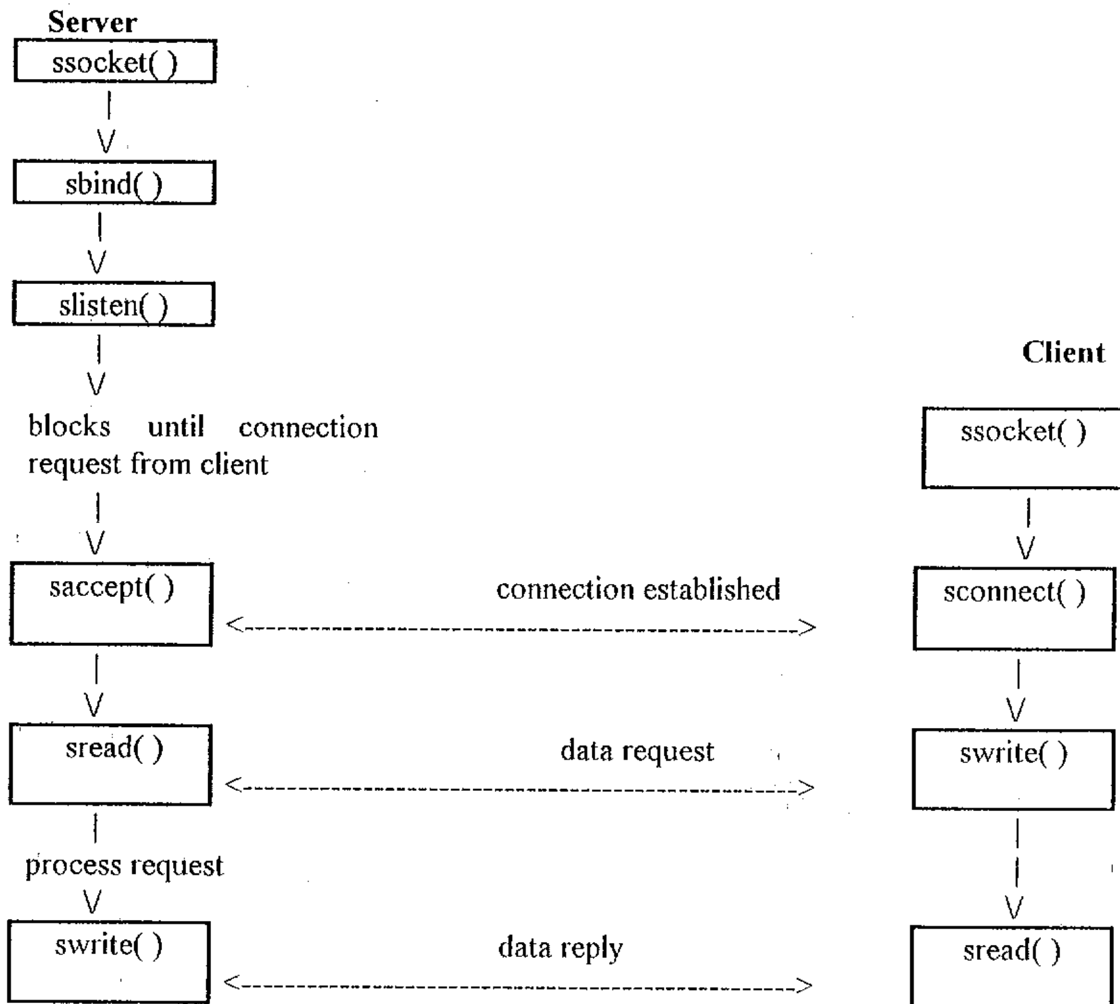


Figure 2-4: Socket system calls for connection-oriented protocol.

•sread( )
The sread( ) function is used to read a stream of bytes being sent by remote process to the local one.

int sread(int socketfd, char far * buffer, int size, int TCPIP_INT);

socketfd is the socket returned by ssocket() function, The two other arguments are pointer to the buffer allocated to receive the data, and size is the maximum number of bytes to read.
sread( ) returns the actual number of bytes being read.

•swrite( )
The swrite( ) function writes a stream of bytes to the network in order to be sent to the remote process.

int swrite(int socketfd, char far *buffer, int size, int TCPIP_INT);

swrite( ) takes the same arguments as sread( ), and returns the actual size of data sent to the remote process.

•sclose( )

int sclose(int socketfd, int TCPIP_INT);

sclose( ) is used to close a socket. If the socket being closed is associated with a protocol that promises reliable delivery (TCP,..) the system must assure that any data within the kernel that still has to be transmitted is sent. Normally the system returns from the sclose( ) immediately, but the kernel still tries to send any data already queued.

The modifications in the communication functions do not require the usage of interrupt 21h, and allow TCPCOM.LIB functions to be securely called from within the device driver.

Chapter 3

The Client/Server Concept

3.1 Introduction

Many companies today are beginning to implement client/server computing. It's not hard to understand why Client/server computing gives companies the means to make the most of all their resources: information, capital, technology - and, most of all, people.

Today, client/server solutions are at work in companies in all industries. They are helping people to work cross-functionality to improve customer service; to participate in international teams that enhance product quality; to apply information in new ways to create innovative and unique products; and to redesign and streamline business processes to reduce time-to-market and improve profitability.

Client/Server computing allows all of a company information to be almost immediately available on any authorized person's desktop system. Properly implemented, the old barriers between functions and different computer systems cease to exist. It is a completely new way of getting things done. It's also one of the most radical changes in information technology since the invention of the microcomputer back in the 70's. In fact, Client/Server computing is helping to revolutionize the way organizations do business.

3.2 Technical Advantages

Using open Client/Server computing offers advantages that are both visible and transparent to the professional and the end-user, including:

- Individuals can work with applications they prefer - but access and work with information held on other applications and databases across the enterprise.
- IT managers can maintain a cost-effective, reliable and secure infrastructure, but respond quickly to new challenges because client/server solutions are very flexible.
- Computing resources can be distributed across the organization - but managed centrally.
- New applications can be developed very quickly - but existing investments in hardware and software are protected.

3.3 The Client/Server Model

The standard model for network applications in the client-server model. A server is a process that is waiting to be contacted by a client process so that the server can do something for the client. A typical (but not mandatory) scenario is as follows:

A- The server process is started on some computer system. It initializes itself, then goes to sleep waiting for a client process to contact it requesting some service.

B- A client process is started, either on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by an interactive user entering a command to a time-sharing system. The client process sends a request across the network to the server requesting service of some form. Some of the type of service that a server can provide are:

- return the time-of-day to the client.
- print a file on a printer for the client.
- read or write a file on the server's system for the client.
- allow the client to login to the server's system.
- execute a command for the client on the server's system.

C- When the server process has finished providing its service to the client, the server goes back to sleep, waiting for the next client request to arrive.

We can further divide the server processes into two types.

1- When a client's request can be handled by the server in a known, short amount of time, the server process handles the request itself. We call these iterative servers[6]. A time-of-day services is typically handled in an iterative fashion by the server.

2- When the amount of time to service a request depends on the request itself (so that the server doesn't know ahead of time how much effort it takes to handle each request), the server typically handles it in concurrent fashion. These are concurrent servers. A concurrent server invokes another process to handle each client request, so that the original server process can go back to sleep, waiting for the next client request. Naturally, this type of server requires an operating system that allows multiple processes to run at the same time. Most client requests that deal with a file of information (print a file, read or write a file, for example) are handled in a concurrent fashion by the server, as the amount of processing required to handle each request depends on the size of the file.

3.4 Client/Server General Architecture

Client/server model is a natural model for many applications. The basic idea is that there is a *server*, which provides a useful service via some defined exported interface, and there are one or more clients which use the service by requesting the server to perform certain actions. The actions that may be requested are defined by the exported interface.

In essence, the server acts as an implementation of an abstract data type with a well defined interface (think in terms of object-oriented programming). We might think of parts of a program as clients and servers; for example, if you put a module in your program to access a database, you could think of that module as the server, and the parts of the program that use the module as clients. The location of the server may be on the same machine as the client, or it may be on another machine far away as in this model because, DOS is not multitasking operating system. (see Figure 3-1)

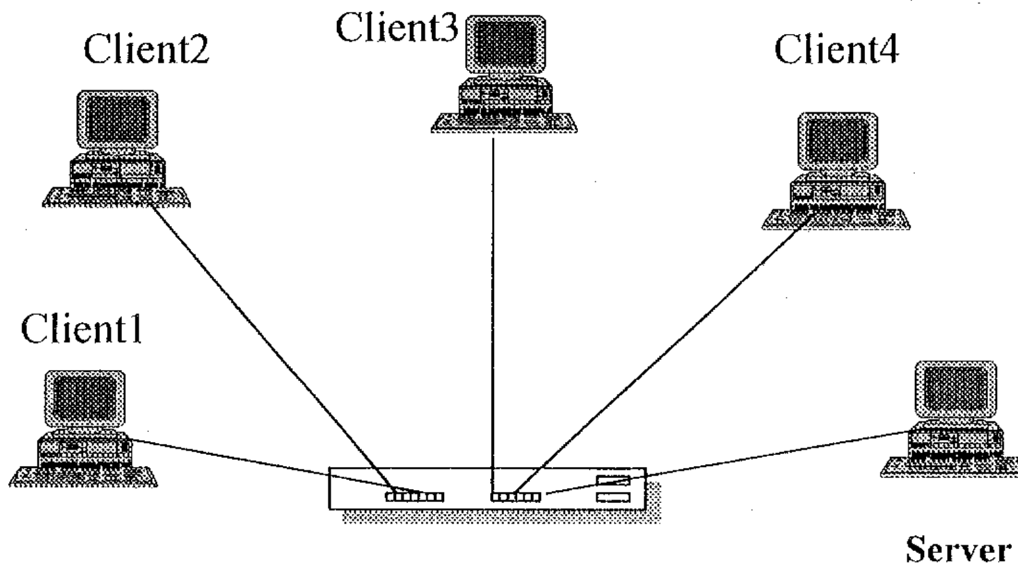


Figure 3-1: Client/Server Model.

Some important aspects of the client/server model are:

1- There is an asymmetric relationship between the client and the server; that is, each takes on well-defined role. For example, in the proposed model the client requests files from the file server, which provides the filesystem services.

2- Usually a machine or a process on a machine is dedicated to servicing requests. That is, a server machine or process doesn't generally have other work to be done.

3- Services may be cascaded: **A** may request a service of **B**, which may in turn request a service of **C** in order to complete **A**'s request.

4- The interface for the service must be agreed upon by the client and the server. It defines which services are exported (e.g., for a print server, the services may be queue, check status, dequeue, etc.) and how those services may be accessed. If a client and a server disagree on the interface protocol, it may lead to unpredictable results.

The reasons for the popularity of the client/server model are:

- It is convenient.
- It is the dominant model in use for today's distributed systems.
- It is simple to understand.

3.5 The Developed Client-Server Protocol

The Client-Server model is prevalent in computer networking. We define iterative servers, which means, that the server knows ahead of time about how long it takes to handle each request and the server process handle each request itself, because we know what the client requested from the server. Notice that the roles of the client an server processes are asymmetric. This means that both halves are coded differently. The server program SERVER.EXE is started first, and dedicates the computer for its usage. Typically it does the following:

1- Checks how many device drivers are loaded, and returns a pointer to the desired block device driver to access later on specified unit of that block device.
2- Displays a group of clients PC, and indicates later on which client is connected to the server, and tells the state of each one (request, error, frame type, etc.) .
3- Opens a communication channel and informs the local host of its willingness to accept client requests on some well-known address.
4-Waits for a client request to arrive at the well-known address.
5- Displays the client PC in connected mode, and shows all connections information.
6- Checks each request and sends it to a specific function. If it is a DOS request, it will forward the command to the specified unit in the block device driver, and waits for an answer from the device.
7- Forwards The answer returned by the device driver to a communication procedure, which returns the answer back to the client.
8- Loops back and goes to its wait state, checks for new clients desiring connections, and check if already connected clients have any read/write requests.

The client does the following:

1- Runs the login.exe program, which connects to a server through a well known address, and verifies if it is acknowledged by a server, and returns the server's total drives.
2- Displays a list of the server mapped drives.
3- Sends any command issued by DOS to access the server mapped drives, to the communication procedure, which forwards it across the network to the server, and blocks for an answer.

4- Retrieves the answer from the communication procedure, and forwards it back to the device driver, which returns it to DOS.
5- Repeats the same job until the user logged out of the system by issuing logout command that closes the communication channel, and terminates.

3.6 The Server Structure

The SERVER.EXE program, first, calls the list( ) routines which lists all the device drivers loaded on the server, and saves a pointer to the block device driver controlling its local drives. Then, it issues a call to Init_Connection( ) procedure that opens a socket and prepares the server process to accept incoming connections. To handle multiple connections at the same time, all clients requests must be served with short period of time. This problem can be managed by making the server non blocking at slisten( ) procedure, and simply doesn' t wait for clients connections request, it creates a table that contains the socket descriptors of all previous connections, in order to check if they have read/write requests.

Whenever the server receives a message from a client to execute a job for him, the server goes with it till it is accomplished before handling another request. Below, is simply a description of the main loop of the server program.

```
while (TRUE)
{
   if ( slisten( ))
   {
      fd = accept( );
      update_table( );
   }
   for ( i=0; i < max_connect; i++)
      if (read (client_info[i].table_fd, buffer, length))
         process_command ( client_info[i].table_fd);
} /* end while */
```

After the server collects the needed data, and information from the client, it calls call_dd( ) routines which sets ES, and BX equal to the address of the request header, and calls the server device driver, once through the device Strategy routine, and a second time through the Interrupt routine to perform the work.

```
void call_dd(void)
 {
   v_call = ( void ( far * ) () ) MK_FP(FP_SEG(disk),disk->dev_strat);
   _ES = _DS   ;
   _BX = FP_OFF (r_ptr);
   v_call ();

   v_call = ( void ( far * ) () ) MK_FP(FP_SEG(disk),disk->dev_int);
   _ES = _DS   ;
   _BX = FP_OFF (r_ptr);
   v_call ();

 }
```

So, after presenting a general description of the client-server architecture, and discussing the main issues concerning the server components, a detailed description of the client components is described in chapter 4.

Chapter 4

DOS BIOS and Device Drivers

4.1 Global view of DOS

Since its introduction with the IBM PC, DOS has become the most popular operating system in the world. From its humble beginnings, DOS has evolved into a powerful tool, with features such as hierarchical disk structures, the ability to control just about any device, and networking capabilities.

The conceptual model for DOS as the master supervisor of resources of a computer system is shown in Figure 4-1[1].

At the core of DOS is the kernel. The kernel provides control functions for administrating and managing the resources of the PC. Memory management routines provide space in which programs can execute. I/O requests from application programs are managed and processed by the kernel. File-management routines within the kernel organize the data for easy access by applications programs. In addition, the kernel is responsible for initializing itself when DOS is booted.

The DOS services interface provides a path for application programs to request services from DOS. It is a defined interface mechanism that processes requests by interacting with the kernel. DOS services include file I/O to devices and disk files, time and date functions, and program control.

Strictly speaking, device drivers are part of the DOS kernel. They provide a standard interface to the devices from within the DOS kernel. As a group, the device driver provide device management for DOS. Each device driver controls a device and uses the PC's BIOS routines, for example, the serial port device driver uses the serial port BIOS interrupt.

Programs generally use DOS services to access and control devices. However, DOS does not prevent a program from directly accessing the BIOS routines. The "back-door" approach is used by many programs to attain higher performance or to perform a task that DOS does not provide.

The most important utility program, and the one that users are familiar with, is COMMAND.COM. This program runs automatically when DOS is booted. COMMAND.COM provides the interface for users to communicate with DOS. The commands that are entered on the keyboard are translated to services requested of DOS. For example, COMMAND.COM is used to set the time and date, to run programs, and to control the devices attached to the PC.

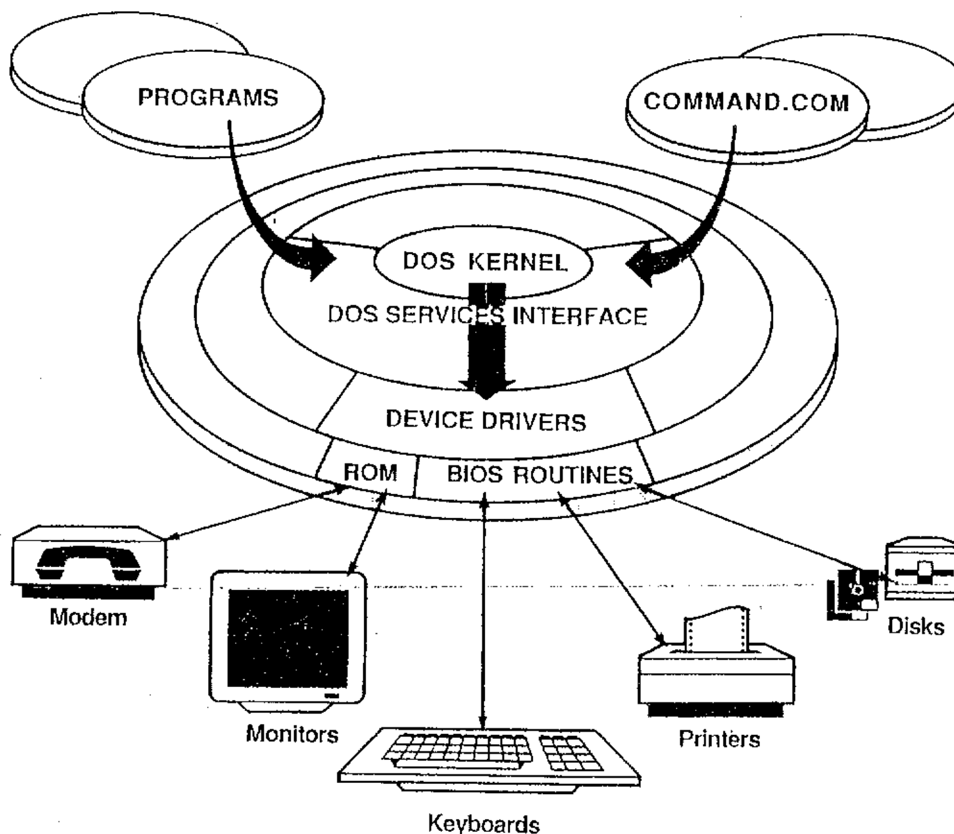


Figure 4-1: The functional parts of DOS.

Moreover, application programs request the PC's resources through the DOS services interface. Without DOS, these programs would have to incorporate all of the services provided by DOS and would, be incompatible with other application programs. DOS provides a common set of features. Application programs use the services provided by the DOS kernel by requesting services through programming calls to DOS.

4.2 Devices for DOS

As we have seen, DOS allows programs to control a set of standard PC devices: keyboard, screen, disks, and serial and parallel adapters. Each DOS device has a unique name assigned to it, and it is through these names that programs are able to access the devices. Table 4-1 lists the names of the standard DOS devices as they are defined in version 2.00 and higher.

4.3 The DOS Interrupts

DOS provide access to devices, files, and various services through the use of the 8086/8088 software interrupt mechanism and the int instruction. Programs call DOS through documented interrupt numbers which are in the range 20h to 3Fh[3]. These interrupt numbers are reserved for use by DOS; they should not be used by your programs. For more information and usage of these interrupts you can refer to any DOS book. These 32 interrupts are shown in Table 4-2.

DOS Device Name	Standard Device
con:	Keyboard/screen
com1:	Serial port #1
aux:	Auxiliary port(identical to com1)
com2:	Serial port #2
lpt1:	Printer port #1
lpt2:	Printer port #2
lpt3:	Printer port #3
prn:	Logical printer port
nul:	Null device
clock$	Software clock
A:	First diskette unit
B:	Second diskette unit
C:	Hard disk (normally)

Table 4-1: The standard device names assigned by DOS

20h	DOS	terminate program
21h	DOS	function call
22h	DOS	terminate address
23h	DOS	CTRL/break exit address
24h	DOS	vector for fatal error
25h	DOS	absolute disk read
26h	DOS	absolute disk write
27h	DOS	terminate but stay resident
28h-3fh	DOS	reserved

Table 4-2: The list of DOS interrupts (not BIOS).

4.4 DOS Device Management

To access a device using DOS, your programs need to indicate what file or device to use; this is called opening the file or device. DOS requires that the name of the file or device be specified through the DOS Open service(3Dh). After this interrupt is received, DOS sets up a file handle, which is used as a standard mechanism

to access the device. This file handle is also used to keep information regarding use of the file or device. A device such as the serial port must be opened using com1: as the device name. Then you can read or write to this device using DOS service calls.
When DOS services a request that requires device access, DOS will translate this request according to a standard set of rules imbedded in code. These rules are uniform across all devices, from simple output-only parallel devices to complex input and output devices, such as disks.

These request services, once converted to a specific command, are then passed to a certain set of routines that process the command. These routines are not common to all devices; rather, each device has a unique set of routines. These routines are the actual DOS device drivers.

DOS has a device drivers for each of the devices attached to the PC. Each service request, however complex, is eventually converted by DOS into a series of simple driver commands and passed to the appropriate device driver.

4.5 Translating Service calls to device Driver Commands

Device drivers are designed to handle simple commands from DOS. The two most common DOS services used to access devices are interrupt 21's read (ah = 3F) and write (ah = 40). These DOS services are relatively complex and may not be translatable to single device driver commands. DOS will issue as many commands to the appropriate device driver as necessary to satisfy the DOS service request.

For example, a program that writes to the disk may issues a write command - interrupt 21h (ah = 40) - that happens to append data at the end of the file. DOS may have to process this single service request by issuing several commands to the disk device driver[1]. The first of these driver commands will need to find more space on the disk for the new data. A driver command will be issued to read the File Allocation Table in which the information on disk space is kept. Then, if there is room on the disk, DOS will write the new data to the disk file by issuing a write command to the disk device driver. Lastly, DOS will update the disk to indicate the time of last access by issuing another driver command to write to the disk. Although this scenario has been simplified, the idea here is that DOS converts a single service request into one or more device driver commands. This is shown in Figure 4-2.

Now, after describing how DOS processes requests for device access by passing the request in the form of smaller, simpler commands to the device driver, let's explore device drivers themselves.

4.6 The DOS Device Driver

4.6.1 Device Drivers for New Devices

DOS device drivers are device-controlling software routines that actually become part of DOS. Because these programs are written to Microsoft-designed specifications, DOS can recognize these new devices and can integrate them with the rest of its standard devices.

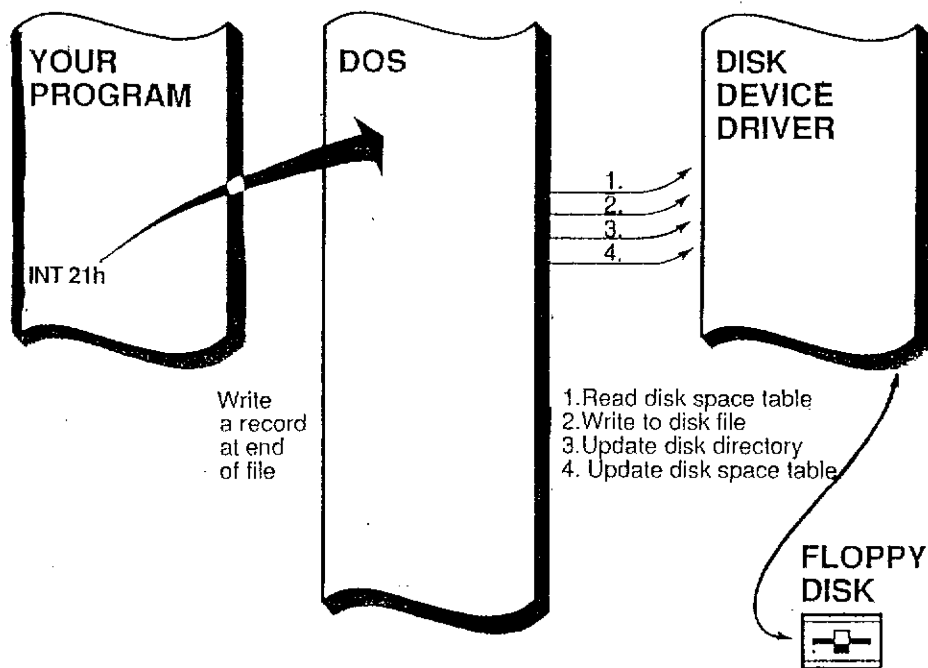


Figure 4-2: Converting A simple service request to several possible disk device driver commands.

Once DOS knows about these devices through their specific device driver routines, the devices can be accessed as easily as the standard disk and screen devices.

DOS needs to know only that the device driver is controlling a particular device, identified by a device name, and that it is capable of processing standard device driver commands.

Without installable device drivers that have a uniform interface to DOS, adding a new device to DOS would be difficult. The manufacturer of the device would have to supply a custom-modified version of DOS in order for you to use the new device. This would create a number of problems. First, you could not use a newer release of DOS unless the newer version also modified to control the new device. Second, because each device manufacturer uses different methods of modifying DOS, incompatibility problems would arise.

The DOS device driver is the most universal and meaningful method of software control for devices. New devices become standard devices in DOS, available for accessing at any time, from within programs and outside of programs, such as from the command level.

4.6.2 New and Old Device Drivers

As we discussed earlier, DOS manages requests for device access from programs by issuing commands to the appropriate device driver. Each device driver contains the name of the specific device it is controlling, and DOS locates the appropriate device driver by searching through the list of installed device drivers.

DOS maintains a linked list of the device drivers starting with the nul: device. The device driver for NUL: is the first in the list and contains a pointer to the next device driver. In turn, each device driver points to the next. The pointer for the last device driver will contain the value -1, thus signaling the end of the list.

DOS manages the standard, replacement, and new device drivers using a relatively simple mechanism. As shown in Figure 4-3, the list of DOS standard device drivers begins with NUL: and continues with CON:, AUX:, and so forth. These device driver programs reside in the area of the PC memory that DOS uses. Whenever a new device driver is installed, DOS inserts it in the list just after the NUL: device. This allows you to replace a standard device driver, because any device request will cause DOS to search this list starting from the first, which is nul:. If you replace a standard device with one of your own, DOS will find the new device first and will never reach the original device of that name, which is now second in the list. Thus, DOS will be able to access new, replacement, and standard device drivers simply by searching this list.

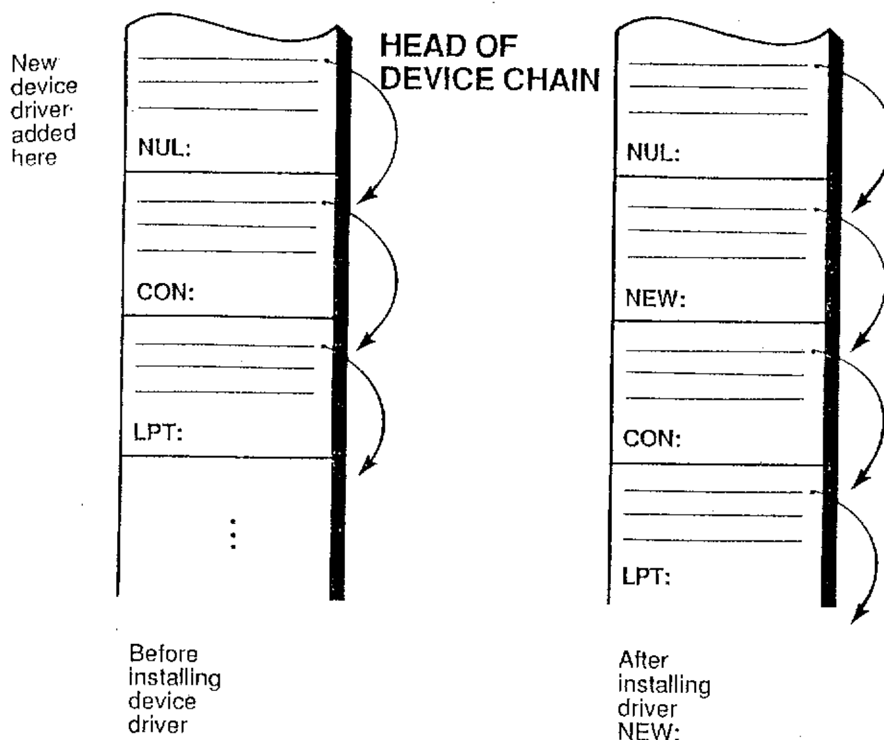


Figure 4-3: Adding a new device driver to the list.

This list of DOS device drivers is called the device chain and is a linked list of the actual device driver programs. To access drivers all DOS needs is a pointer to the first item, the device nul:. DOS can then find the rest of the device drivers.

4.6.3 Overview of a Driver Program structure

A device driver program consists of five parts: the Device Header, data storage and local procedures, the STRATEGY procedure, the INTERRUPT procedure and the command-processing routines (see Figure 4-4).

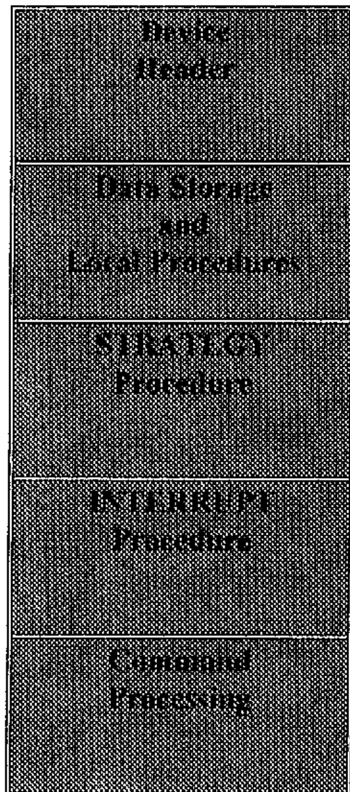


Figure 4-4: The five basic parts of the device driver

Let's look briefly at these five sections. The beginning of a device driver program does not contain code the way normal programs do. Rather, the Device Header contains information about the device driver itself. This information is used by DOS and includes the device name for the driver and the pointer to the next driver.

The second part of the driver is used to store local data variables and local routines and procedures.

The third and fourth parts of the device driver contain what Microsoft calls the STRATEGY and INTERRUPT procedures. These two procedures are integral to proceeding each command that is passed
from DOS to the device driver. They allow DOS to pass control to the driver.

The last part of the driver contains the actual code routines that process each of the commands that DOS passes to the device driver.

4.6.4 Communication of DOS with the Driver

Let's see how DOS and the driver work together. Figure 4-5 shows that when DOS calls the driver it passes a packet of data to the device driver. This call might be to write to a RAM disk or send some special character to a graphics board. This packet of data is called a Request Header and contains information for the device driver such as the data to be written to the device. DOS sets up the registers ES and BX to contain the address of the Request Header when DOS calls device driver.

The Request Header The request Header is a packet of data that is passed from DOS to the driver; this data tells the driver what to do and the location the data involved in the work to be performed. For example, if DOS wants write a character to the serial port, it needs to specify the write command and the character to write. Therefore, DOS needs to pass to the driver both command and some data. Both of these are contained in the Request Header (Note: Do not confuse the Request Header with the Device Header. The Device Header tells DOS about the driver program, and the Request Header contains the data on which the device driver works.) The Request Header is described in Table 4-3.

As shown in Table 4-3, the Request Header is a variable-length packet of data. Within this packet, the length of the Request Header is contained in the first entry. The second entry contains the unit code of the
device. This is normally used when more than one device is attached to the controller. An example of this is the floppy disk controller, which often controls two drives.

The A: drive would be unit 0, the B: drive would be unit 1, and so forth. The third entry is the command code, which tells the device driver what action to take. The fourth entry is used as a status indicator.

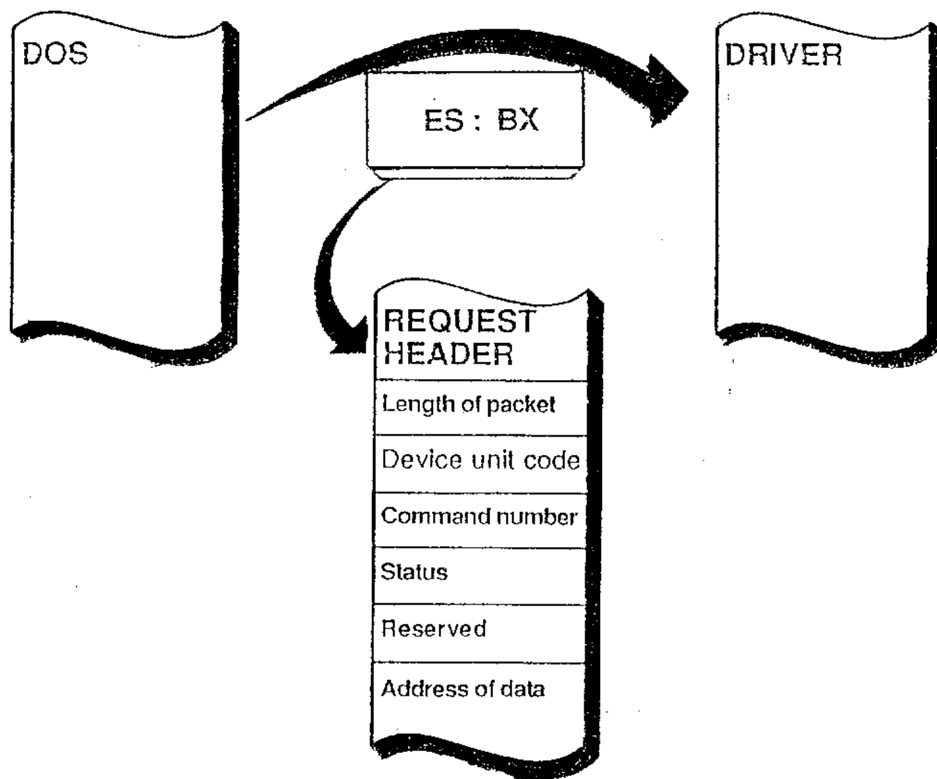


Figure 4-5: DOS calling the device driver with a pointer to the request header.

Entry #	Length (bytes)	Description
1	1	Length in bytes of this Request Header (varies with the amount of data in the Request
2	1	Unit code of the device
3	1	Command code
4	2	16-bit word for the status upon completion
5	8	Reserved for DOS
6	Varies	Data specific for a command

Table 4-3: Definition of the Request Header that is passed to the device driver.

The fifth entry is reserved for use by DOS (its use is undocumented). Finally, the last entry is the data field. This field varies in length depending on the command in the third field.

DOS automatically sets up a Request Header whenever a program makes a request to DOS that involves device driver. This data packet resides in DOS's reserved memory space and is built with information provided from the calling program. The address of the Request Header is passed to the device driver when DOS passes control to the driver. This address is stored in the driver's local storage area. You need to specify both the segment address and the offset address of this Request Header, because the Request Header can be anywhere in the 640K memory. specifying only an offset address assumes that the packet will be in the current segment of memory in which the program is executing. DOS passes this segment and offset address in the ES and BX registers of the 8088/8086, respectively[2].

Drivers Calls from DOS You might assume that each command DOS passes to the driver involves a single call to the driver. Alas, this is not the case. Recall that DOS expects the device driver to have two procedures defined - the STRATEGY and the INTERRUPT procedures. Let's explore the two-step call that DOS makes to the device driver for each command request.

The Two-step Call to the Device Driver Each time DOS asks the device driver to process a command, for example a read or write command, DOS will call the device driver twice. The first time, DOS will pass control to the STRATEGY procedure defined for the device driver. The second time, the device driver will be called at the address specified for the INTERRUPT procedure.

Think of the STRATEGY procedure as instructions that perform the set-up and initialization for the driver. The INTERRUPT procedure the uses the information from the STRATEGY procedure to process the command request from DOS. This process is shown in Figure 4-6.

Although it is not apparent from DOS manuals, this two step approach allows DOS to distinguish between the request for the driver (the set-up) and the actual work to be done by the driver. You can think of this two-step process as analogous to writing a check and cashing it at a bank. You may write the check on Monday(the set-

up) and not cash it (the work) until Friday. In the same way, DOS notifies the driver that here is a work to be done with a call to STRATEGY and then calls the driver again through INTERRUPT to allow it to work.

Let's develop a scenario to see why the STRATEGY and INTERRUPT are necessary. Assume that your PC, through DOS, can multitask, which means that it can perform several tasks at one time. This permits you to do more work in given period of time[1]. Although DOS does not provide third capability currently, it is an important feature that future versions of DOS will have.

It is likely that the various multiple tasks in order of importance, the calls they make to device drivers also need to be prioritized. For example, a task that is downloading a file using a modem might be higher priority than a task that is updating a collection of addresses. The two-entry point approach allow DOS to do this. DOS can process the device driver calls in the priority order of the calling task. This is accomplished by linking into a chain all driver request calls(all the calls to STRATEGY and putting all the actual work calls (calls to INTERRUPT) into another chain in priority order. After DOS calls all device drivers through the STRATEGY routine, it then inspects the INTERRUPT chain to see which one has the higher priority. The closer a device driver is to the beginning of the chain, the higher its priority.

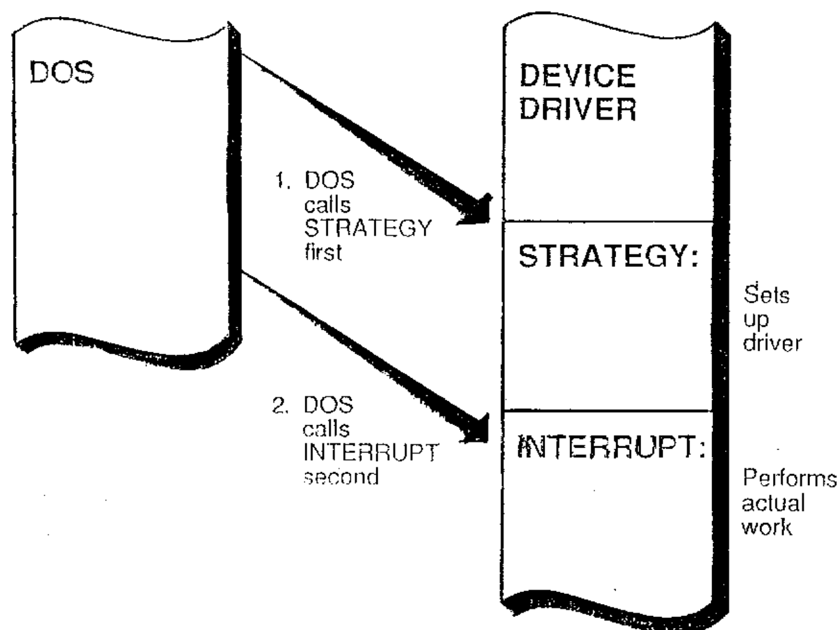


Figure 4-6: DOS calls the device driver twice.

Without this two-step mechanism to set up and perform the actual work DOS would call the device drivers on a first-come, first-served basis. To make this scenario a little easier to understand, let's use an example. Assume that there are three outstanding driver requests:

- Request A has a low priority
- Request B has a medium priority
- Request C has a higher priority

The STRATEGY and INTERRUPT chains are illustrated in Figure 4-7. As this figure shows, each program request for device driver service causes DOS to place the first (set-up) call in the STRATEGY chain and the second (work) call in the INTERRUPT chain.

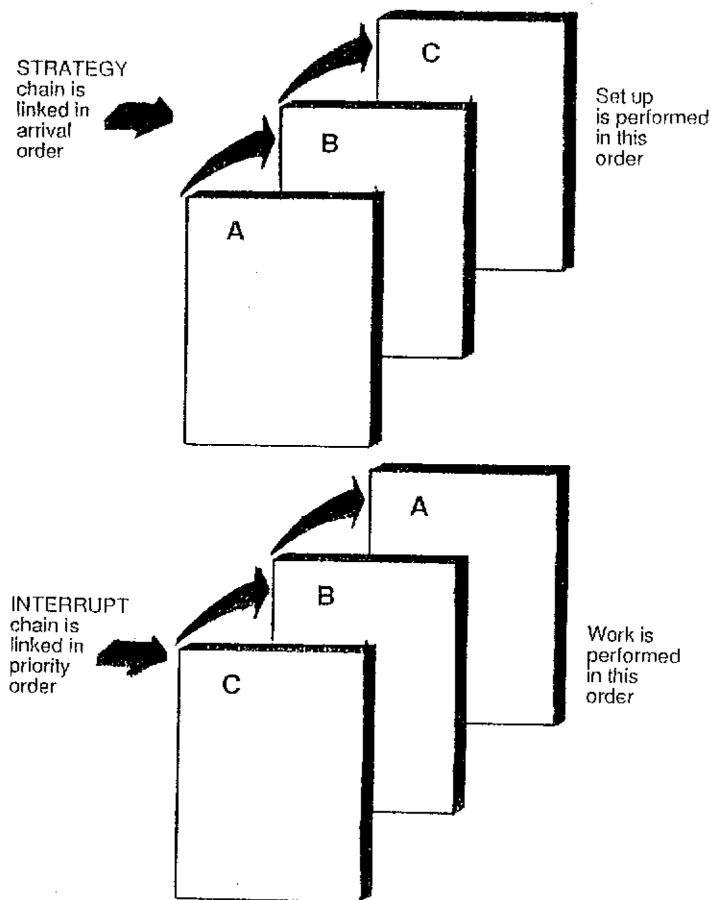


Figure 4-7: The effect of three driver requests

When three programs make device driver requests, the set-up calls are linked into the STRATEGY chain in order of arrival, and the work calls are placed in the INTERRUPT chain in priority order. Think of this as writing checks in order during the week and then sending out the most important checks first on Saturday. In effect, you are handling all the incoming items as they arrive but storing the most important items into a work list for processing[1].

What The STRATEGY Procedure Does When the driver is first called, the STRATEGY routines saves the address of the Request Header, which is contained in the ES and BX registers. This is done to prepare the driver for the second call to its INTERRUPT procedure.

The sequence of events is shown in Figure 4-8, in which DOS prepares to call the device driver by building a Request Header, and in Figure 4-9, DOS calls the device driver at the STRATEGY procedure.

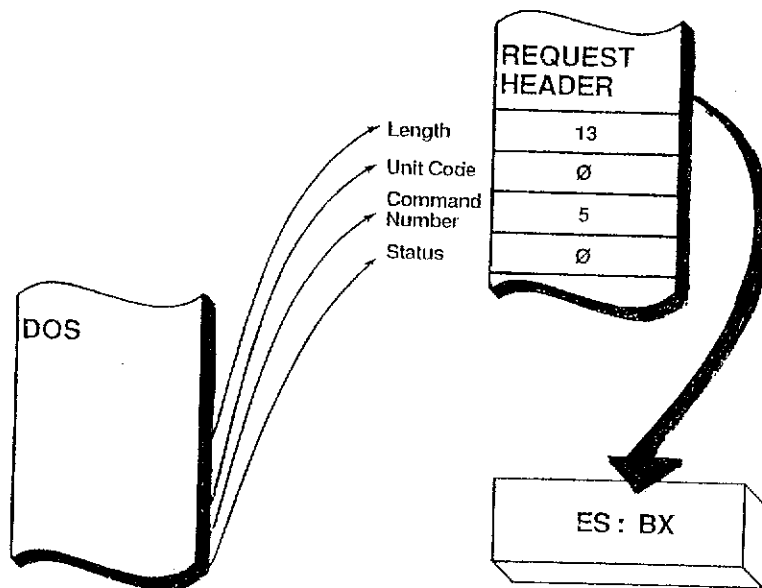


Figure 4-8: DOS preparing to call the device driver for the first time.

When DOS calls the device driver the second time, it does so through the INTERRUPT procedure. Here the real work of the device driver begins. The Request Header that contains information for the device driver to process is handled by the code located in the INTERRUPT procedure. Control is then passed to the command-processing routines. This is shown in Figure 4-10.

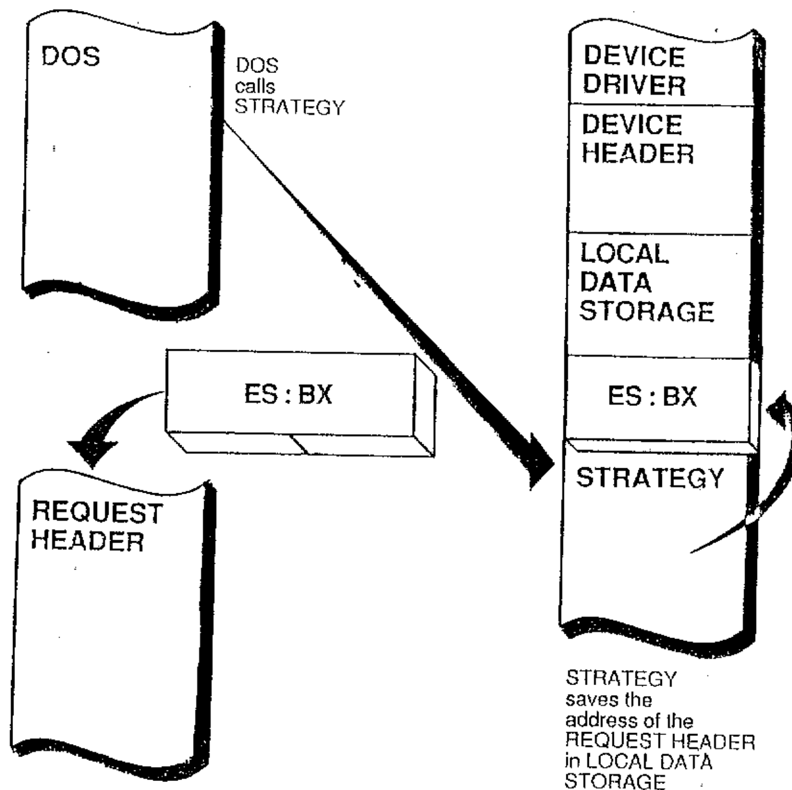


Figure 4-9: The STRATEGY procedure storing the address of the Request Header in local data storage.

4.7 Block and Character Devices

DOS drivers need to distinguish between character and block devices. Recall that a block device transfers data in groups of characters, and character devices transfer data one character at a time. Of the control commands that the device driver issues to the device, some are appropriate to character devices and some to block devices. The Media Check command is one example of a block device command. Because diskettes can be formatted for single-sided or double sided use, the DOS disk device driver needs to know which format has been used. To find out, DOS issues a Media Check command to the disk device driver, which in turn reads a block of data from the disk. From the information returned in this block of data, DOS can determine if the diskette is single or double-sided.
The Media Check command is unique to disk block devices and is not applicable to character devices.

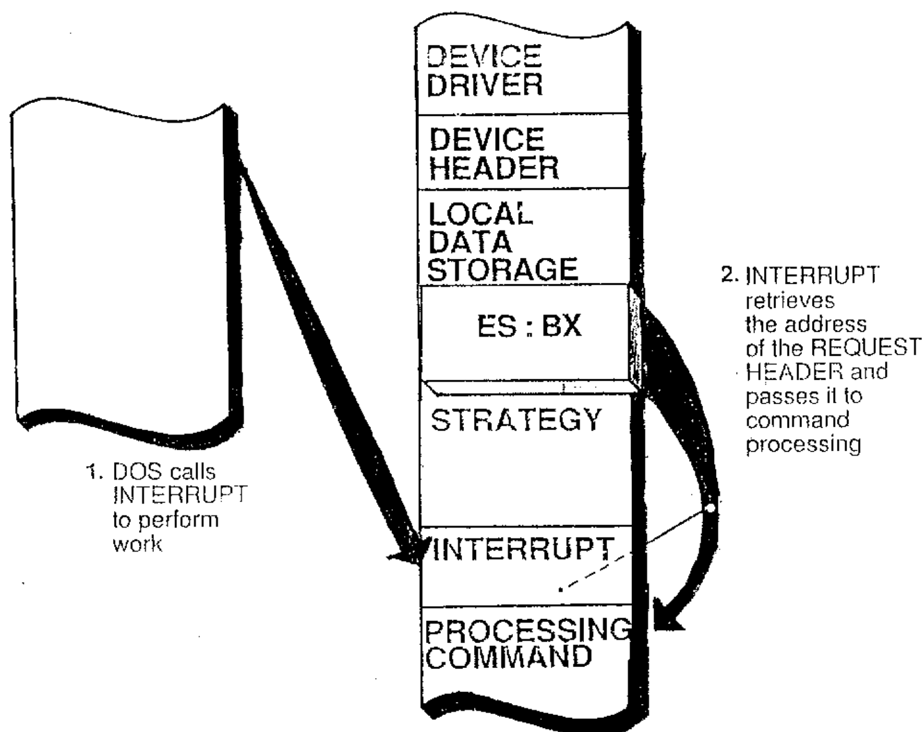


Figure 4-10: DOS Processing the INTERRUPT routine.

DOS also needs to know which type of device its driver is controlling in order to determine the appropriate commands the device driver can perform.

4.8 Device Driver Commands

Recall that programs make service requests of DOS. Each of these requests translates to a specific set of commands that the driver understands[2]. These commands are common to all device drivers.

Number	Command Description
0	Initialization
1-2	Not applicable
3	IOCTL Input
4	Input
5	Nondestructive Input
6	Input Status
7	Input Flush
8	Output
9	Output With Verify
10	Output Status
11	Output Flush
12	IOCTL Output
13*	Device Open
14*	Device Close
15*	Not applicable
16*	Output Till Busy
17-18**	Undefined
19*	Not applicable
20-22**	Undefined
23**	Get Logical Device
24**	Set Logical Device
25***	IOCTL Query

* = DOS version 3+ only
** = DOS version 3.2 only
*** = DOS version 5.0 only

Table 4-4: The list of commands for character-oriented devices.

Commands defined by Microsoft for device drivers are listed by device type in Table 4-4 for character devices and in Table 4-5 for block devices. Note that not all of the commands are available for all versions of DOS.

Number	Command Description
0	Initialization
1	Media Check
2	Get BIOS Parameter Block
3	IOCTL Input
4	Input
5	Not applicable
6	Not applicable
7	Not applicable
8	Output
9	Output With Verify
10	Not applicable
11	Not applicable
12	IOCTL Output
13*	Device Open
14*	Device Close
15*	Removable Media
16*	Not applicable
17-18**	Undefined
19*	Generic IOCTL
20-22**	Undefined
23**	Get Logical Device
24**	Set Logical Device
25***	IOCTL Query

* = DOS version 3+ only
** = DOS version 3.2 only
*** = DOS version 5.0 only

Table 4-5: The list of commands for block-oriented devices.

4.9 Tracing a Request from Program to Device

To finish this section, we will look at an example of what happens along the way as a program calls a device driver. Let's assume that a program has asked you to type some data from the keyboard into a file called MYFILE. Let's say the program will write the data into a record in a disk file. Figure 4-11 shows the various steps performed by your program, DOS, the disk device driver, the BIOS, and the device itself.

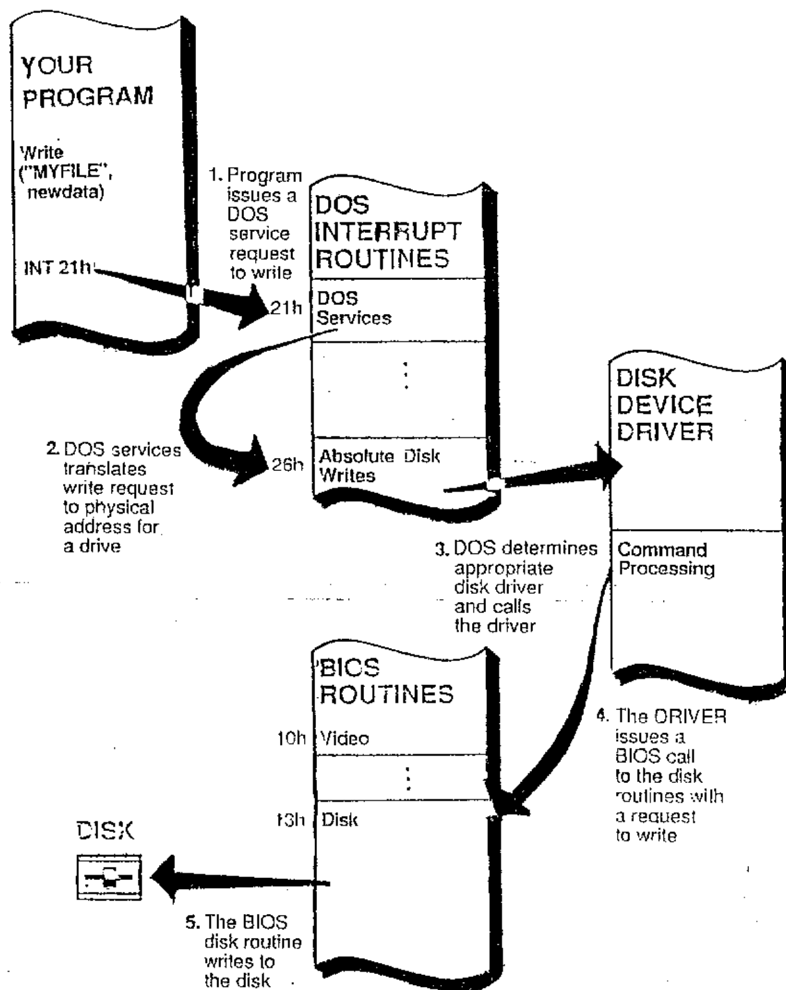


Figure 4-11: Block diagram of the paths taken to write a block of data to the disk.

When you have typed in all your data, your application program will issue a Write to a previously opened disk file named "myfile". The data to be written is contained in a record or variable block of data named "newdata". The Write is a call to a library function in the programming language used in your program. This function will take your Write command and convert it to a DOS function call. There are many DOS calls that write data to a file; for this example, we will assume that it is simply a Write Sequential File Record call. The library function is generally written in assembly language. It will set up the data for a Write Sequential File Record as DOS needs it and then call DOS by issuing interrupt 21h.

The first part of DOS that is used is the call handler, which is where control goes when the interrupt 21h is executed. It is here that DOS inspects the type of function that the caller has set up (as found in the AH register). In this case the function is hex 15, which means Write Sequential File Record to DOS.

DOS then internally locates the relative position of the disk file to which you record is to be written. Next, DOS finds the starting address, relative to the beginning of the disk, of the file "myfile." This is done by searching through the disk directory for information on where "myfile" resides. The relative position of the record to be written to is added to the position of the start of the file; this yields the absolute position on the disk at which the "newdata" record should be stored. This part of the DOS call handler is responsible for determining all the information for a given disk and all the information for the files on this particular disk.

The next step performed is that this data is sent to the general disk handler, which is also the DOS Absolute Disk Write routine (also known as interrupt 26h). This is called from the DOS kernel.

Interrupt 26h or the DOS Absolute Disk Write routine requires two basic pieces of information. The first piece of information identifies the drive to which DOS needs to write the data. The second piece of defines the location of the write relative to the beginning of the disk(that is the starting sector). The reason for this routine is that DOS treats all disks alike: all of the sectors of each disk are numbered from 0, starting at the beginning of the disk. Thus, the file, and the general disk handler calculates the relative position of the record within the disk. What the DOS Absolute Disk Write routine does is to determine the actual physical address to which the data should be written, using the relative information calculated by the original int 21h service Write Sequential. The physical address referred to here is the relative physical sector on the disk to which data should be written. Finally, this information is passed to the disk device driver.

In turn, the disk device driver is responsible for converting the physical address to a track, a sector, and a surface; it also performs the actual write.

A point should be made here about the BIOS routines. The disk device driver uses the disk BIOS routines to perform the actual reads and writes to the disk. This is accomplished by executing an interrupt 13h after specifying the appropriate subfunctions for read or write.

Once the disk device driver has finished the write operation, it will return a status to the Absolute Disk Write routine, which, through the DOS call handler, will return the status to the original calling program. Just as the original write request passed through the DOS call handler, the Absolute Disk Write routine, the disk device driver, and the disk BIOS routine, the status "percolates" through the layers back to the original program.

So the device driver plays a vital role in ensuring that your data is written to the disk. This illustration of the complicated process of writing a record to the disk has involved many steps. You have seen the relative roles of the device driver, DOS, and the BIOS. the interactions for all device drivers are similar to those in the example.

4.10 An Overview of the Device Driver

We will describe in the next section what you will need to know in order to write code for these sections: "Device Header Required by DOS", " The STRATEGY Procedure", " The INTERRUPT Procedure", " DOS Command Processing", " Error Exit", and " Common Exit".

4.10.1. The Device Header

The Device Header is the first piece of data that DOS sees, it defines to DOS how to deal with the device. Figure 4-12 shows the five basic parts of the Device Header.

Three of the five basic components of the Device Header deal with address pointers. The first is a double-word pointer (offset and segment address) to the next device driver in the file. When DOS loads the device driver into memory from a file, other device drivers can be added to the same file. In fact the PC-DOS standard device drivers for the console, floppy disk, printer, communications port, and clock are contained in a single file named IBMBIO.COM. DOS uses the pointer to index past the current device driver for the next device driver, if there is one. To signal to DOS that there is not another device driver, place -1 in both words of this first field.

Device Header
Pointer Next Device Driver
Device Attributes
Pointer Strategy Procedure
Pointer Interrupt Procedure
Device Name

Figure 4-12: The five components of the Device Header.

The second and third pointers of the Device Header are used by DOS to locate the driver's STRATEGY and INTERRUPT procedures. These fields contain the offset addresses of these procedures; they are simply the labels that locate the procedures.

4.10.2 The Device Attribute Field

The second field of the Device Header is important for DOS. This field describes to DOS the type of device your device driver is controlling, and, more

importantly, it defines the types of commands that must be implemented in the device driver. In earlier versions of DOS, this field used bits to define the type of device. In later versions, some of the bits were used to indicate for what types of commands the device driver provided processing. Table 4-6 completely describes the Attribute word of the Device Header[2].

Let's look at the purpose of each bit in detail.

Bit 15 and 14 Bit 15 defines to DOS whether the device driver controls a block-oriented device (0) or a character-oriented device (1). This bit is crucial because several of the following bits (13 and 0) have different meanings depending on whether the device is a block or a character device. Also, the name field of the Device Header (described later) will have different meanings depending on the type of device.

Bit 14 is used to tell DOS whether the device driver supports the I/O control commands (IOCTL Input and IOCTL Output). Recall that I/O control is used to pass control information to and from the driver. If this bit is set, you need to implement the two IOCTL commands.

The Evolving Bit 13 Bit 13 has several meanings, depending on the device type. If the device is block-oriented, setting this bit will indicate to DOS that the device is a disk that contains a non-IBM-compatible format; leaving this bit off will tell DOS that the device contains an IBM-compatible format. If the device is character-oriented and the DOS version 3.0 or greater, setting this bit indicates that the device driver can handle Output Till Busy commands.

The issue of whether a disk uses an IBM-compatible format has evolved from a simple concept to a complex one. Recall that the FAT follows the Boot Record (also known as the reserved area). On all IBM PC-DOS formatted diskettes, the FAT is always the second sector of the diskette. This was the initial definition for bit 13 set to 0. This also meant that DOS used the Media Descriptor to identify diskettes. Instead of using the Media Descriptor byte from the BIOS parameter Block, however, DOS used the Media Descriptor byte from the first FAT entry. Thus, to identify the type of diskette in use, DOS would have to read he FAT into memory and pick off the first FAT entry. DOS could not do this unless it could presume that the FAT was always in the same place on all diskettes. The inner workings of DOS to accomplish this task are even more complicated. As we shall show you later, in the section on the GetBPB command, the contents of the data-transfer area will depend on whether or not bit 13 is set.

To make matters worse, the definition of bit 13 in later versions of DOS has changed subtly. You may recall that if bit 13 is set to 1, the format of the disk need not be IBM-compatible. This means that the FAT need not start at the second sector. What DOS will do at this point is to use the BPB to locate the FAT, the File Directory, and the user data area. This is the current definition of bit 13 as found in the manuals. If bit 13 is not set, the device driver uses the Media Descriptor from the FAT to determine the media type. If bit 13 is set, the device driver uses the BPB to determine the media type.

To try to make some sense of all this, keep in mind that, as we showed in the chapter on disk fundamentals, the media descriptor is not a good mechanism to determine the media type. Disks come in all different sizes and have different physical characteristics, such as the number of tracks, cylinders, and heads. With different sizes

for the FAT, the number of FATs, and File Directory, it is impossible to fit all these different combinations into a single media descriptor, particularly one that is limited to eight combinations(F8h to FFh). This is made worse by the fact that disks can have almost any media descriptor; there is nothing sacred about a given media descriptor value.

In order to allow for all these possibilities, you can set bit 1 on, allowing DOS to use the BPB to determine where things are.

Bits 12 to 0

Bit 12 is undefined and should contain a value of 0.

Bit 11 is used to indicate whether the device driver supports the Device Open, the Device Close, and the Removable Media commands. Note that all three commands are applicable to block-oriented devices, such as disk drives, and only the first two are applicable to character-oriented devices such as screens.

Bit 10 Through 8 are undefined and should be set to 0.

Bit 7 is used by DOS 5.0 device drivers to allow user programs to query whether certain IOCTL functions are available for use.

Bit 6 is used only with device driver written for DOS version 3.2 or greater and indicates whether the device driver supports the Get Logical Device command (23) and the Set Logical Device command (24). For DOS versions 3.3 or greater this bit, if set, indicates that the device driver supports Generic IOCTL commands for both character and block devices.

Bit 5 is undefined and should be set to 0.

Bit 4 is the Special bit that is set if the device driver supports fast console I/O by implementing interrupt 29h code.

Bit 3 is set if the device driver implements a clock device. If this bit is set, DOS replaces the standard clock device driver with the current lock device driver.

Bit 2 is set if the device driver is the NUL: device. You cannot replace the NUL: device driver, so this bit is not available for use. This bit is set for the standard NUL:device driver and allows DOS to identify when it is being used.

Bit 1 is set if the current device driver is to be the standard output device (also known as the screen or video output device). Set this bit to indicate that you are replacing the standard console output device. If this is the case, then bit 0 should also be set. For DOS version 4.0 or greater, setting this bit means that block device drivers have the capability of using 32-bit sector addresses, thus supporting disks larger than 32Mb.

Bit 0 has several meanings. For character-oriented devices, setting this bit indicates that the DOS standard console input device is being replaced by the current device driver. For DOS version 3.2 through 4.01, if the device is a block-oriented device, setting this bit indicates to DOS that the device driver supports Generic I/O Control through command 19.

Bit	Value	Description	DOS Version
15	0	Device is block-oriented	2+
	1	Device is character-oriented	
14	0	I/O control is not supported	2+
	1	I/O control is supported	
13	0	IBM format block device	2+
	1	Non-IBM format block device	
	1	Output Till Busy command Available for character devices	3+
12	0	Undefined value should be 0)	
11	0	Open/Close/Removable Media not supported	3+
	1	Open/Close/Removable Media supported	
10	0	Undefined (value should be 0)	
9	0	Undefined (value should be 0)	
8	0	Undefined (value should be 0)	
7	1	IOCTL Query	5.0
6	1	Get/Set Logical Device(block device)	3.2+
	1	Generic OCTL	3.3-5.0
5	0	Undefined (value should be 0)	
4	1	Special bit for fast console I/O	2
3	1	Current clock device	2+
2	1	Current NUL device	2+
1	1	Current standard output device	2+
	1	32-bit sector addresses (block device)	4.0-5.0
0	1	Current standard input device (character device)	2+
	1	Supports Generic I/O control (block device)	3.2-4.xx

Table 4-6: The bit settings of the Attribute word.

Bottom-line Necessary Settings As we mentioned earlier, setting some of the Attribute bits will trigger the possibility of DOS sending certain types of commands to the device driver for processing. This is because some of the bits are used not just for device definition but for command definition. Table 4-7 shows across index of Attribute bits and commands that the device driver may encounter. Not shown in this table are the commands that the device driver normally processes that are not triggered by an Attribute bit being set.

In summary, the Attribute word is a powerful feature that allows each driver to identify itself to DOS. You can control the commands that DOS is allowed to send to the device driver as well as replace the DOS standard devices. Table 4-8 summarizes the Attribute words for various versions of DOS for the DOS standard devices.

4.10.3 The Device Header Name Field

The Device Name field is 8 bytes in length and has two meanings. For character-oriented devices, this field contains the actual text name of the device. If you replace any of the DOS standard devices, you must supply the name of the device

you replace: CON:,PRN:,etc. If you are not replacing a standard device, supply the name you wish to use to identify the device. Be sure to choose a name that does not normally interfere with file names that are in use. For example, if you use the name BASIC for your driver, you can no longer refer to files named BASIC. Indeed, the name that you supply for a driver's name becomes a reserved name and is no longer available for use as a file name. The device name must be in upper-case characters. If the device name is less than 8 bytes in length, you have to fill the rest of the field with blanks.

Commands	**Bits Set**															
Triggered	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Initialize																
Media Check																
Get BPB																
IOCTL Input	R															
Input																
ND Input																
Input Status																
Input Flush																
Output																
Output Verify																
Output Status																
Output Flush																
IOCTL Output			R													
Device Open				R												
Device Close				R												
Removable						B										
Output Til Busy			C													
Undefined																
Undefined																
Generic IOCTL										R					B	
Undefined																
Undefined																
Undefined																
Get Logical Device										B						
Set Logical Device										B						
IOCTL Query									R							

R = Required for both character and block devices
C = Character devices only
B = Block devices only

Table 4-7: Attribute bits setting that trigger device driver commands.

Device Name	DOS Version	Vendor	Attribute Word	Bits Set and Description
NUL:	All	All	8004h	15 character device 2 NUL:device
CON:	All	Most	8013h	15 character device 4 Fast I/O 1 Standard Output 0 Standard Output
	2.11	Victor	C013h	15 character device 14 IOCTL support 4 Fast I/O 1 Standard Output 0 Standard Output
AUX:	All	Most	8000h	15 character device
	2.11	Victor	C000h	15 character device 14 IOCTL support
PRN:	2	IBM	8000h	15 character device
LPTx:	2.11	Others	8000h	15 character device
	2.11	Victor	C000h	15 character device 14 IOCTL support
	3.0	IBM	8800h	15 character device 11 Open/Close
	3.1	IBM	A000h	15 character device 13 Output Til Busy
	3.2-4.01	IBM	A040h	15 character device 13 Output Til Busy 6 Get/Set Logical device
	5.0	Most	A0C0h	15 character device 13 Output Til Busy 7 IOCTL Query 6 Generic IOCTL
COMx:	All	Most	8000h	15 character device
	2.11	Victor	C000h	14 IOCTL support

Table 4-8: The various Attribute words found in various versions of DOS.

Device Name	DOS Version	Vendor	Attribute Word	Bits Set and Description
CLOCK$:	All	Most	8008h	15 character device 3 Clock device
	2.11	Victor	C008h	15 character device 14 IOCTL support 3 Clock device
Disk	2	IBM	0000h	- block device
	All	Victor	6000h	- block device 14 IOCTL support 13 Non-IBM format
	3.0, 3.1	IBM	0800h	- block device 11 Open/Close/Removable
	3.2, 3.3	IBM	0840h	- block device 11 Open/Close/Removable 6 Get/Set Logical device
	4.XX	Most	0842h	- block device 11 Open/Close/Removable 6 Get/Set Logical Device 1 32-bit sector addresses
	5.0	Most	08C2h	- block device 11 Open/Close/Removable 7 IOCTL Query 6 Get/Set Logical device 1 32-bit sector addresses

Table 4-8: (continued) The various Attribute words found in various versions of DOS.

For block-oriented devices, this field does not specify the device name; instead, the first byte of the field is used to specify the number of devices the device driver controls. Because block devices are assumed to be disks, the number of disks already installed by DOS will determine the drive letters with which a particular device driver will start. If another disk-type device driver follows the current one, the sum of the disks already installed by DOS and the current number of units will determine the drive letter for the following disk device driver.

4.10.4 DOS Command Processing

When DOS makes a request of the device driver, a command is sent to the device driver in the form of a Request Header. DOS expects the device driver to perform a function based on the command. There are 26 different commands available to device drivers for processing.

No single device driver will have to process all 26 of these commands. Some of the commands are not defined and are reserved for use by future versions of DOS; some commands are only applicable for certain types of devices. The version of DOS for which you write a device driver will determine the number of commands that are applicable. Finally, you can simplify choose not to implement some commands.

Table 4-9 shows the list of DOS device driver commands with device-type and DOS-version applicability.

Command Number	DOS version	Device Type	Description
0	2+	Both	Initialization
1	2+	Block	Media Check
2	2+	Block	Get BIOS Parameter Block
3	2+	Both	I/O Control Input
4	2+	Both	Input (from device)
5	2+	Character	Non-Destructive Input
6	2+	Character	Input Status
7	2+	Character	Input Flush
8	2+	Both	Output(to device)
9	2+	Both	Output(with verify)
10	2+	Character	Output Status
11	2+	Character	Output Flush
12	2+	Both	I/O Control Output
13	3+	Both	Device Open
14	3+	Both	Device Close
15	3+	Block	Removable Media
16	3+	Character	Output Till Busy
17-18	3.2+	-	Undefined
19	3.2+/3.3+	Block/Both	Generic I/O Control
20-22	3.2+	-	Undefined
23	3.2+	Block	Get Logical device
24	3.2+	Block	Set Logical device
25	5.0	Both	IOCTL Query

Table 4-9: The DOS device driver commands, the DOS versions and the device types with which they work.

The number of commands that a device driver needs to process will depend on four factors: the operations permitted by a device the type of device being controlled, the Attribute bits set, and the DOS version for which it is intended.

Drivers for output-only devices, such as printers, need only implement the Output commands (Output, Output Verify, Output Status, Output Flush, Output Till Busy). Character-oriented device will have a maximum of 14 applicable commands. In addition, by not setting certain bits in the Attribute word, we can avoid having to implement associated commands. For example, if bits 14 (I/O Control) and 11 (Device Open/ Device Close/Removable Media) are not set, up to five of the commands need not be implemented. Lastly, if we write device drivers for DOS version 2.0, we will be dealing with only 13 commands.

In the following sections we will describe each of the commands and what they do. We will use the corresponding Request Header structures as an aid to developing the required responses for each command.

Command 0-INIT Command

This is the first command issued by a device driver at startup time. DOS issued this command directly after loading the device driver into memory. This command is issued once at loading time and never used after that.

The purpose of this command is to allow the device driver to prepare the device for use by setting up values in various registers, data buffers, pointers, and counters. Once the device driver has been initialized, DOS assumes that it is ready to process other commands.

Table 4-10 shows the structure of the Initialization command.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
02H	Command(00H)	1	In
03H	Status word	2	Out
0DH	Number of units	1	Out
0EH	End of available memory*	4	In
0EH	End address	4	Out
12H	Pointer to command line	4	In
12H	Pointer to table of BPB pointers	4	Out
16H	Device number**	1	In
17H	Error message flag***	1	Out

Table 4-10: The Request Header for the Initialization command.

The steps required to process the Initialization command are listed below:

1. Initialize the device, data buffers, and counters.
2. Set the number of units for block devices.
3. Set the Break address.
4. Set the pointer to the table of BPB addresses for block devices.
5. Set the status word.

It is important to know that only in this command we can use some of the DOS service routines which are prohibited in other device driver commands. Even this feature, being allowed in driver initialization, is limited to DOS services 01h through 0Ch, 30h, 25h, and 35h. Other services are not permitted, for DOS is still in the

process of initializing itself. We can use these services to determine the DOS version and to display messages on the screen only during initialization. Table 4-11 lists the allowed DOS services.

Service	Description
01h	Keyboard Input
02h	Display Output
03h	Auxiliary Input
04h	Auxiliary Output
05h	Printer Output
06h	Direct Console I/O
07h	Direct Console Input Without Echo
08h	Console Input Without Echo
09h	Print String
0Ah	Buffered Keyboard Input
0Bh	Check Standard Input Status
0Ch	Clear keyboard buffer
25h*	Set Interrupt Vector
30h	Get DOS Version Number
35h*	Get Interrupt Vector

* = DOS 5.0

Table 4-11: The DOS service that device drivers may use when processing the Initialization command.

Let's look at the structure used to define the dynamic part of the Request Header (refer to Table 4-10). The byte variable at offset 0Dh, is set by the block device driver, and indicates the number of units controlled. The device driver must return the number of units. This number overrides the first byte of the Device Name field of the Device Header.

The variable at offset 0Eh contains the break address, which signals the end location in memory of the device driver. This address tells DOS where the next available memory location is for loading other device drivers. We can use this feature to our benefits because, the initialization code is used only once, we can place this code at the end of our device driver and specify the beginning of this code as the Break Address. This address is required for all device drivers.

The variables at offset 12h are the addresses (offset and segment, respectively) of the BPB table that must be returned to DOS by block device drivers that control a disk. DOS needs to know the types of disks the device driver can handle. We can satisfy this requirement by building BPB for each type of disk the device driver can handle. A table is created that contains the addresses of each of these BPBs, and it is the address of this table that is returned to DOS. With this information, DOS and the device driver can determine if disks have been removed or changed and where the information is on each disk.

The address used by the BPB table pointer is also used by DOS to pass to the device driver a pointer to the command line in the CONFIG.SYS file. Recall that a DEVICE= command specifies to DOS that a device driver is to be loaded. We can use this pointer in both character and block device drivers to access the entire string beyond the "=" character. Note that we cannot change the command line but we can

use this feature to specify run-time parameters that the device can use for special configuration.

Note that the DEVICE= command string is terminated by an 0Ah when there are no arguments. When there are arguments, the string is terminated with the following sequence: 00h, 0Dh, 0Ah.

The variable at offset 016h contains the next available driver letter. This variable is available for device drivers running under DOS versions 3.0 and greater.

Block device drivers can use this information to display the drive letters that are controlled by the device driver. The drive letter is actually a number that corresponds to the drive letter (0 means A:, 1 means B:, etc...). Finally, the Status word, at offset 03h must be set before exiting from the device driver.

Command 1- Media Check

The Media Check command is valid only for block devices. This command is sent by DOS to determine whether the disk has changed. Among the three types of disks: floppy disk, hard disk, and RAM disk. Only the floppy disk is capable of being change. However, DOS plays it safe by always issuing a Media Check command before performing any reads or writes to any disk.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command(01H)	1	In
03H	Status word	2	Out
0DH	Media description	1	In
0EH	Media status	2	Out
0FH	Pointer to volume name*	4	Out

Table 4-12: The Request Header of The Media Check command.

* **Only returned when open/close/removable media bit is on and media status = FFH(-1).**

The structure for the Media Check command is shown in Table 4-12. The sequence of events for determining whether the media has changed is shown below:

1. Retrieve the Media Descriptor byte.
2. Determine whether the disk has changed by checking the amount of time elapsed since the last access, using hardware detection methods, or comparing disk information.
3. Set Media status.
4. Set the Status word of the Request Header.

Hard disk and RAM disk do not change, so we can simply indicate this. However, for floppy disks, determining whether the media has changed is a difficult task. As shown above, three basic methods can be used to determine whether the

media has changed: a check for elapsed time, a check for hardware detected disk change, and a check of disk information.

The first method involves keeping track of the time of last disk access compared with the current time. From a practical point of view, changing floppy drives takes a certain amount of time, at least two seconds. If we calculate that less than two seconds have elapsed since the last access, we can assume that the media has not changed. If the last access was more than two seconds ago, however, we cannot be sure whether the disk has changed or not.

The second method is the best of the three. High-capacity (1.2Mb) diskette drives send a signal when the drive door is opened, we can detect this and set the media status accordingly. This signal is often called the changeline signal and is active if the door has been opened. Unfortunately, this signal is not available from most other disk drives.

The last method is most complex, requiring the disk device driver to save information on the disk with each access. The information saved includes the media descriptor byte, the volume ID, and the BPB. If any of these parameters changes between the last disk access and the current one, we can assume that the disk has changed. However, this method is not always reliable. For example, comparing the media descriptor byte from the Request Header with the media descriptor of the current disk does not reliably indicates a disk change. If they are different, the disk has changed. If we changed disks using two similarly formatted diskettes whose media descriptor bytes would be identical, this method could erroneously assume that the disk has not changed. This would also be the case if we compared the BPBs or the volume IDs.

However there s a way around the problem of determining disk changes. As shown in Table 4-13, the media change status allows or three conditions:"media has changed", "media has not changed ," and "don't know whether media has changed". If we cannot determine whether the disk has changed, then we set the media status word, at offset 0Eh, to 0, which indicates "do not know if the media has changed."

Value	Description
-1	Media has changed
0	Don't know if media has changed
+1	Media has not changed

Table 4-13: The three values for the media change status word.

The media status word should be set to -1(media has changed) for all disk types on the first Media Check command. This is true for the very first access of RAM disks and hard disks as well as floppy disks, because DOS does not have accurate information on the disk. Subsequent Media Check commands for hard disks and RAM disk should have the media status word set to 1(media has not been changed).

If the disk device driver has set bit 11 (Open/Close/Removable Media) of the Attribute word in the Device Header, there is an additional programming consideration. If the disk device driver sets the variable at offset 0Eh to -1 (media has changed), then the variable at offset 0Fh must be set to the offset and segment address

of the previous volume ID. This presumes that the device driver has saved the volume ID of the previous disk. If the device driver has not been programmed to save the volume ID, these variables should point to a field containing a volume ID of NO NAME, followed by four spaces and 00h. This is the signal that tells DOS that there should be no checking of the volume ID.

DOS uses the volume ID information on a disk change if the previous disk needs to be reinserted. This allows DOS to update the disk that was prematurely removed.

Lastly, the Status word of the Request Header must be set before exiting the device driver. If there is an error in reading the disk for media information, the error bit and error code should be set with the number of the error that was encountered.

Command 2 - Get BPB

The Get BIOS Parameter Block (BPB) command is valid for block device drivers only. DOS sends this command to the device driver when it needs to know more about the current disk. This occurs under two conditions: if the Media Check command returns a status of -1(media has changed) or if the Media Check command returns a status of 0 (don't know) and there are no dirty buffers for the disk.

Before any reads or writes to the disk, DOS need to check if the disk has been changed or not. As long as there is data to be written to the disk, DOS assumes that the disk has not been changed, and this buffered data is called dirty buffers. If there is no dirty buffers, and the media check returns 0 (don't know); then DOS assumes that the disk has been changed. This solves the problem of disk change that was discussed in the previous section. The reason of this works is simple: If there are any buffers to be written out, DOS will do so at the earliest possible time. This ensures that disks can be changed at any time without having to perform an action to write out data. Thus, if a time period has been exceeded or if the device driver cannot determine a disk change, DOS assumes that the disk has been changed. This causes DOS to assume that the disk is new and that new disk information will be received.

The GET BPB command accesses the disk and returns to DOS the BPB. This information allows DOS to locate the File Allocation Table (FAT), the File Directory, and the user data area for the new disk. The steps needed to process the Get BPB command are shown below:

1- Determine where the Boot Record is on the new disk.
2- Read the Boot record into memory.
3- Retrieve the BPB from the Boot Record.
4- Return a pointer to the new BPB.
5- If bit 11 of the Attribute word is set, determine where the File Directory begins, search the File Directory for the volume ID, save the old volume ID, and save the new volume ID.
6- Set the Status word of the Request Header.

The device driver is responsible for reading the BPB from the disk. A pointer to the new BPB is then returned to DOS through the Request Header variable at offset 012h. The Get BPB structure is shown in Table 4-14.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (02h)	1	In
03H	Status word	2	Out
0DH	Media description **a**	1	In
0EH	Pointer to FAT **b**	4	In
12H	Pointer to BPB	4	Out

Table 4-14: The Request Header for the Get BPB command.

a This value is of little use. The new BPB's media description value supersedes this one.

b If non-IBM bit is set, this field points to a temporary buffer. If the bit is clear, the field points to the old disk's FAT - do not modify the old disk's FAT.

The BPB is located in the Boot Record (also known as the reserved area). For floppy disks, this is the first sector of the disk; for hard disks, this is the first sector of the logical disk drive. Recall that a hard disk may be partitioned into several logical drives. It is up to the device driver to determine the start of the logical drive (partition) relative to the first physical sector of the hard disk. Obviously, many calculations are necessary to find the hard disk BPB. Table 4-15 describes the BPB.

The buffer address specified by the variable at offset 0Eh has different meanings depending on the DOS version and the setting of bit 13 of the Attribute word of the Device Header. Bit 13 is set to indicate that the disk format is not IBM-compatible. This specifies to the device driver that the buffer can be used for anything. Otherwise, the buffer contains the initial FAT sector (with the first entry being the media descriptor byte) and must be not alerted for all versions of DOS. For DOS version 3.2, we can use this buffer even if bit 13 is not set. We need not concern ourselves with this, for the BPB contains all the information that DOS needs about the new disk.

Name	Starting Location	Length	Description
SS	0	2	Sector Size in bytes
AU	2	1	Allocation Unit size (sectors per cluster)
RS	3	2	Number of reserved Sectors
NF	5	1	Number of FATs on this disk
DS	6	2	Directory Size (number of files)
TS	8	2	Number of Total Sectors
MD	10	1	Media Descriptor
FS	11	2	FAT Sectors (each FAT)
ST	13	2	Number of Sectors per Track
NH	15	2	Number of Heads
HS	17	2/4*	Number of Hidden Sectors
LS	21	4*	Number of Large Sectors

(* = DOS .0 +)

Table 4-15: The fields that comprise the BPB.

Finally, because DOS assumes that there is a new disk, the device driver can read the new volume ID off the new disk and save the old volume ID. This involves determining where the File Directory is on the new disk and searching through it for the volume ID entry. Once the volume ID is found and stored in a variable, the other command processing sections can return the old volume ID in the event of an illegal disk change. For example, the Media Check command returns this old volume ID if the disk has changed.

Command 3 - I/O Control Input

Command 3, I/O Control Input, is valid for block and character device drivers if the I/O Control Support bit (14) of the Attribute word is set. Recall that the attribute word of the Device Header allows DOS to pass I/O control strings to and from the device driver. I/O control strings are data passed between a program and the device driver. The data is not intended to be sent to the device; these strings are merely a means of communicating with the device driver.

We can use I/O control strings in two ways. The DOS service IOCTL Output is used to send control information to the device driver. When control information from the device driver is required, the IOCTL Input DOS service is used. The DOS 044h services call provides IOCTL functions. Table 4-16 shows the IOCTL Input structure.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
02H	Command (03H)	1	In
03H	Status word	2	Out
0EH	Pointer to buffer	4	In
12H	Transfer count (sectors for block, bytes for character)	2	I/O
14H	Start sector number (block devices only)	2	I/O

Table 4-16: The Request Header for the IOCTL Input command

The steps required to process the IOCTL Input command are listed below:

1. Retrieve the address of the data-transfer area.
2. Retrieve the transfer count from the Request Header.
3. Store the I/O control string in the data-transfer area.
4. Return the transfer count.
5. Set the Status word of the Request Header.

The I/O control string data that is passed to the device driver in the data transfer area need not to be moved into a buffer inside the device driver. The device driver simply use a pointer to access the data.

The format for the I/O control string information must be agreed upon between the program and the device driver. Otherwise, the program sends data that the device driver does not understand. This data can be binary, ASCII, or a combination of both. A command code should be set up for each function desired.

Then, within the application program using IOCTL functions, we should decide how to interact with the user to determine which of the command codes to send to the device driver. Within the device driver, we must add code to recognize these command codes and process them accordingly.

The transfer-count variable at offset 12h is an important part of the common I/O control string format. This transfer count determines if the data transferred is correct. Because both sides must agree on the format, the number of bytes (or sectors) to be transferred can also be confirmed.

Using the variable at offset 0Eh as a pointer, the device driver can read or write an I/O control string in the data-transfer area. For the IOCTL Input command, the device driver is instructed to return I/O control string to DOS. DOS, in turn returns it to the program requesting I/O control information.

Once an I/O control string is stored in the data-transfer area, the device driver sets the variable 012h to indicate the number of bytes in the data transfer area. Next, the Status word of the Request Header is set to indicate the appropriate status; the device driver then exits back to DOS.

Command 4 - Input

The Input command is used by all device drivers to send data from the device back to DOS. The steps for processing this command are shown below:

1. Retrieve the address of the data-transfer area.
2. Retrieve the transfer count from the Request Header.
3. Read the requested amount of information from the device.
4. Return the transfer count.
5. Set Status word of the Request Header.

The Input command reads data from the device into the data-transfer address specified by the variable at offset 0Eh. The count is contained in the variable at offset 12h. For character devices the count is the number of bytes to be transferred. For block devices the count is the number of sectors to be transferred. In addition, the variable at offset 14h indicates the start sector number for the block device if it is less than 65,535. For disks larger than 32Mb the sector number may be larger. If so, offset 14h will have 0FFFFh and the 32-bit starting sector number will be found in variable at offset 1Ah.

Once the transfer is complete, the device driver specifies the number of bytes or sectors transferred in the same variable, at offset 12h. This variable does not have to be updated if the transfer was successful, this variable must be changed to indicate the number of bytes or sectors transferred. This tells DOS that the data was only partially transferred.

For block device drivers that implement the Open/Close/Removable Media bit (11) of the Device Header Attribute word, there is an additional programming consideration. In fact we have seen in the Get_BPB command section that disks can be changed even though DOS still has data for the disk. If the device driver receives an Input command and determines that the wrong disk in the unit, the device driver aborts the Input command and return an error to DOS. This type of error is detected by timing the last disk access or by monitoring a disk-changed signal from the

hardware. If it is determined that the Input command is for the wrong disk, the device drivers returns an error (0Fh - illegal disk change) and the old volume ID. This allows DOS to ask the user to reinsert the disk that has the old volume ID. This allows DOS to ask the user to reinsert the disk that has the old volume ID. Note that this feature is for DOS versions 3.0 or greater.

The Status word in the Request Header is set to indicate DONE and any errors before the device driver exits back to DOS. This is partially important if we have encountered an error. Table 4-17 shows the Input command structure.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (04h)	1	In
03H	Status word	2	Out
0DH	Media description	1	In
0EH	Pointer to buffer	4	In
12H	Transfer count	2	I/O
14H	Starting sector **a**	2	In
16H	Pointer to volume name **b**	4	Out
1AH	Long starting sector **a**	4	In

Table 4-17 : The Request Header for the Input command.

a With DOS 4.0 and above, if the field at offset 14H contains FFFFH, the starting sector is in the field at offset 16H.

b Returned only when invalid disk change error occurs (DOS 3.0 and above)

Command 5 - Nondestructive Input

The Nondestructive Input command is valid for character devices only. The applications program using the DOS service Get Input Status (0Bh) causes DOS to send this command to the device driver, asking to look ahead one character. DOS assumes that character devices have an input buffer in which characters are stored. The device driver requests the next character in this buffer. Some devices have the ability to retrieve a character from the buffer without removing the character. Other devices require the character to be removed from the buffer. The term nondestructive means that the character will still be available for the next Input command.

Not all devices have a data buffer. For devices that do not, the device driver must actually do a read of one character. This character is saved for the next Input command as well as being passed back to DOS to satisfy the Nondestructive Input command. Device drivers also store characters for keyboard devices. Keyboard input using the ROM BIOS interrupt 16h returns two bytes. The device driver returns one byte and saves the other. The Nondestructive Input command would simply retrieve the stored character. If the device driver did not have a character saved, the device driver would request the next character.

The steps required to process the Nondestructive Input command are listed below:

1. Retrieve a byte from the device.
2. Set the Status word of the Request Header.

The device driver retrieves a byte from the device and stores it in the variable at offset 0Dh. If there is no character in the device buffer, the device driver sets the BUSY it of the Status word to indicate that the device buffer is empty. The status word of the Request Header is set before exiting from the device driver. The Scheme for the Nondestructive Input structure is shown in Table 4-18.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
03H	Command(05H)	1	In
03H	Status word	2	Out
0DH	Character returned	1	Out

Table 4-18: The request Header for the Nondestructive Input command.

Command 6 - Input Status

The Input Status command is valid for character devices only. This command returns the status of the character-device input buffer, telling DOS whether there are any characters in the device buffer ready to be input. Table 4-19 shows the structure for the Input Status command. The steps involved in processing the Input Status command are shown below:

1. Retrieve the status from the device.
2. Set the BUSY bit of the Status word:
   0 If there are characters in the device buffer or if the device doesn't have a buffer
   1 If there are no characters in the buffer
3. Set Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (06H)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-19: The Request Header for the Input Status command.

The device driver processes this command by retrieving the status from the device. If the device has characters in the buffer, the BUSY bit is not set. If the device does not have characters in the buffer, the BUSY bit is set.

For devices that do not have a data buffer, the BUSY bit is not set. This is contrary to what we might expect based on the preceding descriptions. The logic

behind this is that DOS will wait for the device buffer to fill if the BUSY bit is set. On the other hand, if the BUSY bit is not set, DOS will issue an Input command immediately. This will result in an actual read, and DOS will not have to wait for nonexistent buffer to fill.

Command 7- Input Flush

The Input Flush command is valid for character devices only. This command empties the character device buffer. Table 4-20 shows the scheme for the Input Flush structure. The steps required to process the Input Flush command are listed below:

1. Flush the character device buffer.
2. Set Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (07H)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-20: The Request Header for the Input flush command.

To process this command, execute instructions that cause the device buffer to be empty. Most devices do not accept control information that causes the buffer to drain. Instead, the device driver simply reads characters from the device until the device status indicates that there are no more characters in the buffer. The device driver sets the Status word in the Request Header before exiting.

Command 8 - Output

The Output command is used by all device drivers to send data to the device. Table 4-21 shows the structure to be used to process the Output command.

The steps taken to process the Output command are listed below:

1. Retrieve the address of the data transfer area.
2. Retrieve the transfer count from the Request Header.
3. Write the requested amount of information in the data transfer area to the device.
4. Return the transfer count.
5. Set the Status word of the Request Header.

old volume ID. When DOS receives the 0Fh error, DOS will prompt the user with the old volume ID, requesting a reinsertion of the old disk.

Command 9- Output With Verify

The Output With Verify command is valid for both character and block devices. This command is used much as the Output command except that, we should build our driver to read back the data after it is written to the device. We should use this command to ensure that the data has been written to the device correctly. The structure for the Output With Verify command is shown in Table 4-22.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (09H)	1	In
03H	Status word	2	Out
0DH	Media description	1	In
0EH	Pointer to buffer	4	In
12H	Transfer count	2	I/O
14H	Starting sector **a**	2	In
16H	Pointer to volume name **b**	4	Out
1AH	Long starting sector **a**	4	In

Table 4-22: The request Header for the Output with Verify command.

a With DOS 4.0 and above, if the field at offset 14H contains 0FFFFH, the starting sector is in the field at offset 016H.

b Returned only when invalid disk change error occurs (DOS 3.0 and above)

The steps required to process this command are shown below:

1. For devices that cannot read data just written, jump to the Output routine.
2. For devices that can read data just written, set a flag to indicate a read. Next, jump to the Output routine and modify it to read the data back in if the flag is set.

The VERIFY command is used to set the verify flag within DOS. If this flag is set, all writes to the device will appear in the device driver as Output With Verify commands instead of Output commands.

For devices that cannot read data just written, this command should be processed by including a jump instruction to the Output routine. If the device can read data just written(as disks can), a flag should be set to indicate that we want to validate the data by reading it back in; then jump to a modified Output routine. The Output routine will write the data to the device and, if the flag is set, will read the data back in. This method uses both the Output and the Input routines to process the Output With Verify command.

Command 10 - Output Status

The Output Status command is valid for character devices only. This command is used to return the status of the device output to DOS. Devices that are output one only, such as printers, have buffers that contain characters waiting to be output. This command is used to Check the status of this buffer. The structure for the Output Status command is shown in Table 4-23.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0AH)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-23: The Request Header for the Output Status command.

The steps required for processing this command are shown below:

1. Retrieve the status from the device.
2. Set the BUSY bit of the status word:
   0 If the device is idle or the buffer is not full
   1 If the device is busy or the buffer is full
3. Set the Status word of the Request Header.

When DOS needs to write to a device, an Output Status command is first issued to the device driver. This tells DOS whether to send the Output command immediately or to wait and issue another Output Status command.

To process this command, the BUSY bit of the Request Header Status word is set. If the device is ready for output, the device driver does not set the BUSY bit. If the device is not ready, the driver sets the BUSY bit.

Command 11 - Output Flush

The Output Flush command is valid for character devices only. This command is used to empty the output device's buffer. The structure for the Output Flush command is shown in Table 4-24.

The steps for processing this command are listed below:

1. For devices that have an output buffer, execute instructions to empty the buffer.
2. Set the Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0BH)	1	In
03H	Status word	2	Output
05H	Reserved	4	
09H	Reserved	4	

Table 4-24: The Request Header or the Output Flush command.

To process the Output Flush command, the device driver executes instructions that empty the output device's data buffer. If the output device does not have a buffer, the device driver simply does nothing. Before the device driver exits, the Status word in the Request Header should be set.

Command 12 - I/O Control Output

The I/O Control Output command is valid for character and block devices if the Device Header Attribute bit 14 is set, indicating that I/O Control is supported. This command is used to send control information from a program directly to the device driver. Data that is passed to the device driver is not meant for the device but for controlling the device. The device driver may use this information in any fashion. The format of the control information must be agreed upon by both the program issuing IOCTL service calls and the device driver. The structure for the IOCTL command is shown in Table 4-25.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
02H	Command(0CH)	1	In
03H	Status word	2	Out
0EH	Pointer to buffer	4	In
12H	Transfer count(sectors for block, bytes for character)	2	I/O
14H	Start sector number(block devices only)	2	I/O

Table 4-25: The Request Header for the IOCTL Output command.

The steps required to process the IOCTL Output command are listed below:

1. Retrieve the address of the data-transfer area.
2. Retrieve the transfer count from the Request Header.
3. Decode the I/O control string contained in the data-transfer area.
4. Set the Status word of the Request Header.

The device driver processes this command by retrieving the address of the data-transfer area in the variable at offset 0Eh. The length of the I/O control string to be processed is contained at offset 12h. This count allows the device driver to determine if the I/O control string has been properly constructed. As we discussed in

the IOCTL Input section, the length of the transfer is important in ensuring that the format if the I/O control string is correct.

The device driver then processes the I/O control string by performing the functions requested. These functions will vary depending on the type of device being controlled and the actions desired.

If there are any errors, the device driver sets the Request Header Status word accordingly.

Command 13 - Device Open

The Device Open command is available to both character and block devices under DOS version 3.0 or greater if the Device Header Attribute bit 11(Open/Close/Removable Media) is set. This command is sent by DOS each time the device as been opened. Used in conjunction with the Device Close command, this command can enable us to determine if devices are being accessed properly. For example, if we want the device to be accessed only by one user at a time, we can reject new opens for our device if we have not received a close command for the previous open. The structure for the Device Open command is shown in Table 4-26.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0DH)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-26: The Request Header for the Device Open command.

For character devices, the Device Open command is used to initialize the device. For example, we can initialize printers by sending a command that sets the top of form or loads a standard font.

For block devices, we can use the device open counter in a different manner. Recall that setting the Attribute bit 11 requires the block device driver to determine whether there is an illegal disk change. We can use the device open counter for this purpose. Disks can be changed when the device open counter is 0 (which means that there are no open files for the disk). As long as the counter is not 0, disks cannot be changed, because there are files opened for the disk.

Command 14 - Device Close

The Device Close command is available to character and block devices running under DOS version 3.0 or greater if the Device Header Attribute bit 11 (Open/Close/Removable Media) is set. This command is sent by DOS each time the device is closed by a program. Use this command to track the number of times a device has been opened. Used with the Device Open command just described, this

command can enable us to determine if devices are being accessed properly. The structure for the Device Close command is shown in Table 4-27.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0EH)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-27: The Request Header for the Device Close command.

The steps required to process this command are listed below:
1. Decrement a (Device Open) counter.
2. Set the Status word of the Request Header.

To process this command, the counter within the device driver that was incremented by a Device Open command is decremented. When the count is 0, we will know that there are no outstanding opens for this device: the device is free.

For character devices, the Device Close command is used to send an optional string to the device. For example, we can send a form feed command to finish a print job. Note that the CON:, AUX:, and PRN: devices are never closed.

As we have just seen for the Device Open command, we can use this device open counter differently for block devices. If the device open counter is 0, the disk may be changed. Therefore, if a GET BPB command is received by the device driver, the disk change is legal. However, if the device open counter is not 0 and the device driver receives a GET BPB command, the disk change is in error.

Command 15 - Removable Media

The Removable Media command is valid for block devices running under DOS version 3.0 or greater that have the Device Header Attribute bit 11 (Open/Close/Removable Media) set. This command is sent by DOS when a program issues an IOCTL service call (44h) asking whether the media is removable (08h). Programs use this command to determine whether the disk is changeable. The structure for Removable Media command is shown in Table 4-28.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0FH)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-28: The Request Header for the Removable Media command.

The steps required to process this command are shown below:

1. Set the BUSY bit of the Status word:
   0 Media is removable
   1 Media is not removable
2. Set the Status word of the Request Header.

To Process this command, the BUSY bit is returned in the Request Header Status word, indicating the media status. The BUSY bit should be set if the media is not removable; it should not be set if the media is removable.

Programs that request this information through the IOCTL service call can decide whether to prompt the user to change disks. For example, the FORMAT program uses this information to prompt the user for floppy disks but not for hard disks.

Command 16 - Output Till Busy

The Output Till Busy command is valid for character devices running under DOS version 3.0 or greater that have the Device Header Attribute bit 13 set (Output Till Busy supported). This command is used by print spoolers to output data to a character device until the device signals busy.

The structure for the Output Till Busy command is shown in Table 4-29. The steps required to process this command are listed below:

1. Retrieve the address of the data-transfer area.
2. Retrieve the transfer count from the Request Header.
3. Write the requested amount of information in the data-transfer area to the device until the device signals busy.
4. Return the transfer count.
5. Set the Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (0DH)	1	In
03H	Status word	2	Out
0DH	Media Description	1	In
0EH	Pointer to Data transfer area	4	In
12H	Byte count	2	I/O

Table 4-29: The Request Header for the Output Till Busy command.

To process this command, the pointer is first retrieved to data-transfer area. The variable at offset 0Eh contains the offset and segment address at which the data resides. The transfer count is then retrieved at offset 12h, which is the number of bytes to write.

The device driver writes characters from the data-transfer area to the device until all the characters are written or until the device signals busy. If all the characters were not written, the number actually written is returned at offset 12h. The device driver sets the Request Header Status word upon exit.

Commands 17 and 18

Commands 17 and 18 are undefined; they are reserved for use by future versions of DOS. For the sake of completeness, the Request Header structures for both commands are shown in Table 4-30.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (11H or 12H)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-30: The Request Header for commands 17 and 18.

Command 19- Generic I/O Control

The Generic I/O Control command is valid for block devices running under DOS version 3.2 or greater that have the Device Header Attribute bit 0 set (Generic I/O Control supported). DOS 3.3 or greater allows Generic I/O Control commands for character devices. This command is used by programs that issue an IOCTL service call (44h) specifying Generic I/O Control functions (0Dh). The structure for the Generic I/O Control command is shown in Table 4-31.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (13H)	1	In
03H	Status word	2	Out
0DH	Major	1	In
0EH	Minor	1	In
0FH	SI Register	2	In
11H	DI Register	2	In
13H	Pointer to IOCTL request	4	Out

Table 4-31: The Request Header for the Generic IOCTL command.

The steps required to process this command are listed below:

1. Retrieve the Major and Minor function codes.
2. Process the Minor function request.
3. Return the transfer count.
4. Set the Status word of the Request Header.

The purpose of this command is to provide a standard I/O control service for block-oriented devices. Beginning with version 3.2, DOS defines a more standard approach to controlling block devices. The Minor function codes define operations that were not truly a part of DOS. For example, formatting a disk was an operation performed by utility programs.

To process this command, first the Major and Minor function codes that are contained in the variables at offset 0Dh and 0Eh are retrieved. Next, it should be verified that the Major function code is correct. The Major codes are shown in Table 4-32. The Minor codes and their meanings are shown in Tables 4-33 and 4-34.

Value	Description
01H	Serial device
03H	Console
05H	Parallel printer
08H	Disk

Table 4-32: The Major function codes for Generic I/O Control.

Value	Description
45H	Set iteration Count
4AH	Select Code page
4CH	Start Code -page prepare
4DH	End Code-page prepare
65H	Get Iteration Count
6AH	Query Selected Code Page
6BH	Query Code-Page Prepare List

Table 4-33: The Minor function codes for character devices.

Value	Description
40H	Set Device Parameters
60H	Get Device Parameters
41H	Write logical drive track
61H	Read logical drive track
42H	Format and verify logical track
62H	Verify logical drive track
46H	Set Media ID
66H	Get Media Id
68H	Sense Media type

Table 4-34: The Minor function codes for the block devices.

The Request Header contains additional information that assists the device driver in processing the Generic I/O Control command.

Commands 20, 21, and 22

Commands 20, 21, and 22 are undefined; they are reserved for future DOS versions. The Request Header structure for these commands is shown in Table 4-35

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (14H or 15H or 16H)	1	In
03H	Status word	2	Out
05H	Reserved	4	
09H	Reserved	4	

Table 4-35: The Request Header for the commands 20, 21, 22.

Command 23 - Get Logical Device

The Get Logical Device command is available for block device running under DOS version 3.2 or greater that have the Device Header Attribute bit 6 set (Get/Set Logical Device supported). DOS 3.2 or, greater allows the user to specify multiple drive letters for a device unit.

For Example, the second disk unit, normally accessed as logical drive letter B:, can also be accessed with the logical drive letter E:. Table 4-36 shows the structure for the Get Logical Device command. The steps required to process this command are listed below:

1. Retrieve the input unit code.
2. Return the last device referenced.
3. Set the Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command(17H)	1	In
03H	Status word	2	Out
0DH	Unit Code	1	In
0DH	Last Device	1	Out
0EH	Command Code	1	
0FH	Status	2	
11H	Reserved	4	

Table 4-36: The Request Header for the Get Logical Device command.

This command is processed by retrieving the logical unit specified in the variable at offset 0Dh. The device driver will determine if there is another logical drive assigned to the same logical unit. If there is no other logical drive assigned, the device driver returns a 0 at offset 0Dh. Otherwise, the device driver returns the logical drive that was last referenced. The values contained at offset 0Dh are 1 for drive A:, 2 for drive B:, etc. Confusing as this sounds, this command is asking the device driver what other drive letter was used to access the same physical device unit.

Command 24 - Set Logical Device

The Set Logical Device command is available for block devices running under DOS version 3.2 or greater that have the Device Header Attribute bit 6 set (Get/Set Logical Device supported). This command allows DOS 3.2 (or greater) users to specify multiple drive letters for a logical drive. Table 4-37 shows the structure for the Set Logical Device command. The steps required to process this command are listed below:

1. Retrieve the input unit code.
2. Save the unit code.
3. Set the Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command(18H)	1	In
03H	Status word	2	Out
0DH	Unit Code	1	In
0DH	Last Device	1	Out
0EH	Command Code	1	
0FH	Status	2	
11H	Reserved	4	

Table 4-37: The Request Header for the Set Logical Device command.

This command is processed by retrieving and saving the logical unit specified in the variable at offset 0Dh. If the device driver does not recognize this drive letter as an alternate drive letter for the units controlled, a 0 is returned at offset 0Dh. The drive letters are numbered starting with 1, where 1 represents a:, 2 represents B:, etc.

Assigning alternate drive letters is accomplished through the use of the DRIVER.SYS device driver supplied with DOS versions 3.2 through 5.0. Arguments on the DEVICE command for this device driver specify additional drive letters for the unit specified.

Programs make use of this feature by using the DOS IOCTL service call (44h) to get and set logical drives.

Command 25 - IOCTL Query

The IOCTL Query command is valid for both character and block devices running under DOS version 5.0 that have the Device Header Attribute bit 7 set (IOCTL Query supported). This command is used by programs to query device drivers to determine whether the device driver supports a specific generic IOCTL function. These are the Minor codes as shown in Table 4-33 for character devices and Table 4-34 for block devices.

The structure for the IOCTL Query command is shown in Table 4-38. The steps required to process this command are listed below:

1. Retrieve the Minor function code.
2. Set the DONE bit of the Status word if the Minor function code is supported by the device driver.
3. Set the Status word of the Request Header.

Offset	Contents	Length(bytes)	Type
00H	Length of packet	1	In
01H	Unit number	1	In
02H	Command (13H)	1	In
03H	Status word	2	Out
0DH	Major	1	In
0EH	Minor	1	In
0FH	SI Register	2	In
11H	DI Register	2	In
13H	Pointer to IOCTL request	4	Out

Table 4-38: The Request Header for the IOCTL Query command.

This command is processed by retrieving the minor code passed to the device driver in variables at offsets 0Dh and 0Eh. If the device driver supports the Minor code, then the DONE bit of the Status word is to be set. Otherwise, the ERROR bit of the Status word is set and the ERROR_CODE field is set to 3 for Unknown command.

Programs can make IOCTL Queries of device drivers through the use of the DOS Query IOCTL Handle (4410h) or Query IOCTL Device (4411h) service calls.

4.10.5 Exiting from the Device Driver

When device drivers exit to DOS, the Status word in the Request Header must be set. There are four items about which we need to be concerned. The DONE bit is always set upon exit from the device driver. This indicates to DOS that the command was properly processed. Next, certain commands (Input Status, Output Status, Removable Media, and Output Till Busy) will set the BUSY bit. The ERROR bit is set if the device driver determines that an error has occurred; in addition, the

ERROR_CODE field must contain a code indicating the error. Table 4-39 lists the appropriate error codes for use by the device driver.

Hex Code	Description of ERROR_CODE
0	Write protect violation
1	Unknown unit
2	Drive not ready
3	Unknown command
4	CRC error
5	Bad drive request structure length
6	Seek error
7	Unknown media
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure
D	Reserved (DOS 3+)
E	Reserved (DOS 3+)
F	Invalid disk change (DOS 3+)

Table 4-39: The standard error codes for DOS device drivers.

The code that executes when exiting from a device driver sets the Request Header Status word and restores the registers that were saved on entry.

4.10.6 The Status Word for Unimplemented commands

When we write device drivers for new devices, we may often be puzzled by what bits in the Request Header Status word to set. We have found that there is no easy formula. Table 4-40 shows bits that should be set for each command upon exit.

Command	Status Word
Initialization	DONE
Media Check	DONE
Get BPB	DONE
IOCTL Input	DONE, ERROR, ERROR_CODE = 3
Input Non-Destructive	DONE
Input	DONE, BUSY
Input Status	DONE
Input Flush	DONE
Output	DONE
Output With Verify	DONE
Output Status	DONE
Output Flush	DONE
IOCTL Output	DONE, ERROR, ERROR_CODE = 3
Device Open	DONE
Device Close	DONE
Removable Media	DONE, ERROR, ERROR_CODE = 3
Output Till Busy	DONE, ERROR, ERROR_CODE = 3
Generic IOCTL	DONE, ERROR, ERROR_CODE = 3
Get Logical Device	DONE, ERROR, ERROR_CODE = 3
Set Logical Device	DONE, ERROR, ERROR_CODE = 3
IOCTL Query	DONE, ERROR, ERROR_CODE = 3

Table 4-40: The Request Header Status word for commands that are not implemented in device drivers.

After understanding the device driver structure, DOS-device driver protocol, and device driver interaction with BIOS an disks, it is time for creating a client that interacts with the already presented server.

Chapter 5

Client/Server Operations

5.1 Development Languages

Selecting the development language is one of the most important tasks for the developer. You can find a variety of programming languages which varies from the low level one, such as Assembly, to the high level language, such as C. Each of these languages have its own advantages and drawbacks. So, you need to evaluate each language before writing the code if it fits your needs, therefore, you must take into consideration the code length, time, speed, skills, parameters passing, etc. So, as you see, it is not a matter of taste rather, it is a matter of logic, and design.
There is no existence for the word good or bad for a programming language, rather there is good or bad selection for the language which will affect our way of implementation.

Device drivers are not an accounting or stocks programs which can be written in 4th generation language, rather they are part of the DOS operating system which needs to be written in a technical language that control every register, stack, segments, and devices. So, we had to choose between two languages, Assembly or C, and I selected C for several reasons, and I will show you in the following sections the advantages and disadvantages of both concerning device drivers development.

5.1.1 Advantages of High-Level Languages

In Assembly, you spend half of your time explaining the code, in order not to forget the exact job being done, and in order to create a simple function that performs a read or a write, you need to write several lines of code, while you can just do it simply in C with just one function call.
Also, we must not forget the time spent to make sure that the registers contain the correct information. We also, expend a lot of effort reinventing the wheel every time we need a routine to convert numbers from one base to another, or to add numbers larger than can be contained in a single register. In addition, we must pay attention to the order of registers pushed in the stack in order to pop them up in the same order.

So, writing device driver in a high language seems very appealing because, it will free us from the tedium task of managing all small, but critical details, and let us concentrate more on the algorithms needed for the task at hand. Besides being easier to write in C, the compactness of the code makes it considerably easier to comprehend and trace than several pages of assembly language.

However writing a device driver in high level language has also its drawbacks.

5.1.2 Disadvantages of High-Level Languages

In general, all things being equal, assembly language programs tend to be faster and more controlled, and due to the fact that device driver are very hardware oriented, they require skills that is not normally associated with writing in C.

Let's take a look to the requirements needed for writing device driver in a high level language.

Driver Loading and Code Organizing

The foremost requirement of a device driver is that the Device Header must be at the very beginning of the file and be first loaded into memory. This is unlike the case with most programs, where the program code is usually found at the beginning of the executable file. In addition, because device drivers are not normal DOS programs, they do not need to allocate space for the Program Segment Prefix (PSP) that precedes the program code. This forces the program code to load at an offset of 100 hex relative to the start of the executable file.

There are two problems here. First, programs written in a high level language such as C do not have any control over where the code is loaded. During the linking phase, the linker decides where and how to load the program code and data as well as the stack and heap. Therefore, we must find a way to load the Device Header at the beginning of our device driver file.

The second problem is that most of C compilers by default generate several segments of program code and data. There will be different segments for code, data, and stack. We will need to order these into the same single segment. By combining all segments into a single segment group, we can minimize the amount of memory used and eliminate the need to use far references, which are slower in execution.

C Expertise Required

By their very nature, device drivers require more programming expertise than other, more "normal" types of programs. Pointers are used to access data that resides outside the device driver program-typically to pass data back and forth to DOS. Structures are used to define the Request Headers specific to each device driver command. As well as, pointers are also used to reference structures and structure members.

The segmented Intel processor architecture forces the use of memory models in C compilers for the PC. Both Microsoft and Borland C support six memory models: *tiny* (one 64K segment for both code and data), *small* (one 64K code segment, and one 64K data segment), *compact* (one 64K code segment and one or more 64K data segments), *medium* (one 64K data segment and one or more 64K code segments), *large* (one or more 64K segments for either code or data), *huge* (same as large but a data item in the data segment may be as large as a 64K segment). Device drivers are usually built as tiny or small model programs. This type of memory segmentation use near references for both code and data since they reside in the same segment.

However not only there is a need to understand memory models in terms of the code generated, but we also need to master the techniques used in referring to data

objects outside of the immediate memory segment: the *far* keyword is often required to complete the memory reference. Thus, each time we read or write data they may require inspecting the data item to see if it is *near* or *far* references.

Finally, we need also to mention that writing device drivers in C requires a good understanding of the requirement of the assembly language for writing device drivers. Since, in C language we can use the assembly directive to write assembly code in our C programs. So, we must understand the use of registers to pass data back and forth between assembly and C, also we need to know how to access C variables using assembly language.

C programming Barriers

The first barrier for writing device drivers in C is that we must be extremely careful when using C library calls in our program. Many library calls translate to DOS calls which, when executed from device driver, would crash DOS because DOS is not reentrant. One useful trick is to avoid the use of either stdio.h or dos.h header files.

The second barrier that C needs to overcome is the problem of the stack. There's not a lot of space on the default stack that is active when control is passed to a device driver; there is only enough room for about 20 pushes. This may not be enough stack space for passing large amounts of data from one routine to another, or when there is frequent nesting of routine calls. Another use of the already small stack by C programs is for local variables used by each routine.

There are two solutions for the small DOS stack problem. The first solution is to live with the existing stack by minimizing the use of the stack. Variables are made global, which eliminates the need to pass data between routines. This also means that no local variables are used unless absolutely necessary. However, large numbers of global variables is generally regarded as poor programming practice.

The second solution , which is more recommended, is to switch to a larger stack upon entry to the device driver and before calling any C routines. This solution gives us the ability to control stack usage; we can declare a large enough stack for the worst-case usage of local variables as well as nested routine calls.

Compiler Complications

There are number of compiler-dependent complications that arise when developing device drivers in C. First, the compiler should be able to compile the C device driver using the tiny memory model, which forces the code and data into a single 64K segment. Most C compilers today provide this capability.

Another complication is that some C compilers add code at the beginning of each routine that checks for potential stack overflow. This added code is a call to routine that calculates whether there is enough space to allocate the routine's local variables on the stack. Unfortunately, the side effects of all this is that a stack segment is also declared. With a stack segment, which is not needed for the C device driver, EXE2BIN will not convert the .EXE file format to a .COM file format. There's an added benefit of removing stack checking: programs are tighter and have faster running code.

The last complication we may find is that most C programs expect a **main** function to which control is passed when the program is executed. The code for using **main** may be generated or included directly by the C compiler or indirectly by the use of certain runtime library functions. This will result in errors during the link phase because the reference will remain unresolved. Device drivers do not have normal entry points because access to the device driver is through the Strategy and Interrupt address definitions of the device header. To solve this problem, we can simply add a definition for the unresolved item in assembly file and declare it **public**.

Linker Madness

The link phase is not without its share of problems. One or more of the problems that may be encountered are described below.

First, most assemblers generate external references in uppercase. This conflicts with C's case-sensitive nature when the linker tries to match the uppercase **Externs** generated for the assembly file with the lower case C routine names. Both Microsoft's MASM and Borland's Turbo Assembler provide a switch to keep public and external symbols case sensitive.

The Linker may be able to produce a COM file directly, thus eliminating the need to use EXE2BIN utility. This requires specifying the **tiny** memory model to the Linker through a switch plus compiling the C modules using the **tiny** memory model. In addition, there cannot be a stack segment defined.

Lastly, several warnings from the Linker could be seen. The first warning will indicate no stack segment which is not needed for the C device driver. Second, there could be a warning that indicates that our program has no start address. Again, it is not needed for device drivers.

5.2 A Closer Look at Tiny Model Program

The best way to understand what a tiny model program means is to create one, then inspect the output of the compiler. The following program, first.c, uses a number of features of the C programming language that we will use later in developing our DOS device driver.

```
/* PROGRAM :    F i r s t                                    */
/*                                                           */
/* REMARKS .    First is a program that is designed to be    */
/*      compiled in TINY model by the TURBO C Version 2.0    */
/*      compiler. Once compiled the assembler output is      */
/*      reviewed to identify the structure and problems that */
/*      will be encountered when developing a DOS device     */
/*      driver in this language.                             */
```

```
#include    <stdio.h>

/*                                                        */
/*      Global Data Required For This Program             */
/*                                                        */

unsigned int   global_int;
unsigned char   global_byte;

/* Function:     F U N C T I O N                          */
/*                                                        */
/* REMARKS:  Function is  a function responsible for      */
/*           accessing the supplied parameters and assigning     */
/*           this global data variables to the current values of the */
/*           parameters.                                  */

 Function (int Param_int, char Param_byte)

{
            global_int  = param_int;
            global_byte = Param_byte;
}

/*  REMARKS :  main is the main program function that is              */
/*             responsible for initializing  its local data variables  */
/*             and then calling Function with them as parameters.  */
/*                                                                    */

void main (void)
{
            int   local_func_int;
            char   local_func_byte;

            local_func_int = 0;
            local_func_byte = 0;
            Function (local_func_int, local_func_byte);
}
```

The program first . c declares two global variables that are visible to the entire program. It contains function main and another function Function, which has two formal parameters.

You will notice that even though first.c is a very small and simple C program, it performs the following types of operations:

* Global variable access

* Local (stack) variable access

* Parameter passing to a function

* Function parameter access

* Function invocation.

Each of the above operations is critical to the operation of a C program. Therefore, an understanding of these items is important in the development of DOS device drivers written in the C programming language.

first . c was compiled with TURBO.C version 2.0. We used the following command to compile the program:

```
tcc -mt -y -M first.c
```

This command requests the TURBO C compiler to generate a tiny model program (-mt) that includes line number information (-y) and a link/load map (-M). The following is the link/load map created from this compilation:

Start	Stop	Length	Name	Class
00000H	00659H	0065AH	_TEXT	CODE
00660H	007E7H	00188H	_DATA	DATA
007E8H	007E8H	00004H	_EMUSEG	DATA
007ECH	007EDH	00002H	_CRTSEG	DATA
007EEH	007EEH	00000H	_CVTSEG	DATA
007EEH	007EEH	00000H	_SCNSEG	DATA
007EEH	00837H	0004AH	_BSS	BSS
00838H	00838H	00000H	_BSSEND	STACK

Address	Publics by Name
0000:02D6	DGROUP@
0000:07CF	emws_adjust
0000:07D3	emws_BPsafe
0000:07C8	emws_control
0000:07D1	emws_fixSeg
0000:0785	emws_initialSP
0000:06F5	emws_limitSP
0000:07C5	emws_nmiVector
0000:07C1	emws_saveVector
0000:07D5	emws_stamp
0000:07C9	emws_status
0000:07CD	emws_T0S
0000:07D9	emws_version
0000:02C0	_abort
0000:046F	_atexit
0000:063A	_brk
0000:06CF	_environ
0000:06D8	_errno
0000:0305	_exit
0000:02D8	_Function
0000:07EE	_global_byte
0000:07EF	_global_int
0000:02E9	_main
0000:0574	_malloc
0000:0648	_sbrk
0000:06DD	__8087
0000:06C8	__argc
0000:06CD	__argv
0000:07E6	__atexitcnt
0000:07F2	__atexittbl
0000:06ED	__brklvl
0000:06D1	__envLng
0000:06D3	__envseg
0000:06D5	__envSize
0000:0220	__exit
0000:07DC	__exitbuf
0000:07DE	__exitfopen
0000:Q7E0	__exitopen
0000:06E9	__heapbase
0000:07E2	__heaplen
0000:06F1	__heaptop
0000:0688	__Int0Vector
0000:068F	__Int4Vector
0000:06C3	__Int5Vector
0000:06C7	__Int6Vector
0000:06D9	__osmajor
0000:06DA	__osminor

0000:06D7 __psp
0000:07EE __RealCvtVector
0000:0283 __restorezero
0000:07EE __ScanTodVector
0000:033A __setargv
0000:0425 __setenvp
0000:06DF __StartTime
0000:07E4 __stklen
0000:06D9 __version
0000:05E2 ___brk
0000:06E5 ___brklvl
0000:0836 ___first
0000:06E3 ___heapbase
0000:06E7 ___heaptop
0000:0832 ___last
0000:0495 ___Pull_free_block
0000:0834 ___rover
0000:0606 ___sbrk

Address	Publics by Value
0000:0220	__exit
0000:0283	__restorezero
0000:02C0	_abort
0000:02D6	DGRDUOP@
0000:02D8	_Function
0000:02E9	_main
0000:0305	_exit
0000:033A	__setargv
0000:0425	__setenvp
0000:046F	_atexit
0000:0495	___pull_free_block
0000:0574	_malloc
0000:05E2	___brk
0000:0606	___sbrk
0000:063A	_brk
0000:0648	_sbrk
0000:0688	__IntOVector
0000:068F	__Int4Vector
0000:06C3	__Int5Vector
0000:06C7	__Int6Vector
0000:06C8	__argc
0000:06CD	__argv
0000:06CF	_environ
0000:06D1	__envLng
0000:06D3	__envseg
0000:06D5	__envSize
0000:06D7	__psp
0000:06D9	__version
0000:06D9	__osmajor
0000:06DA	__osminor
0000:06D8	_errno
0000:06DD	__8087
0000:06DF	__StartTime
0000:06E3	___heapbase
0000:06E5	___brklvl
0000:06E7	___heaptop
0000:06E9	__heapbase
0000:06ED	__brklvl
0000:06F1	__heaptop
0000:06F5	emws_limitSP

```
0000:0785   emws_initialSP
0000:07C1   emws_saveVector
0000:07C5   emws_nmiVector
0000:07C9   emws_status
0000:07C8   emws_control
0000:07CD   emws_TOS
0000:07CF   emws_adjust
0000:07D1   emws_fixSeg
0000:07D3   emws_BPsafe
0000:07D5   emws_stamp
0000:07D9   emws_version
0000:07DC   __exitbuf
0000:07DE   __exitfopen
0000:07E0   __exitopen
0000:07E2   __heaplen
0000:07E4   __stklen
000:07E6    __atexitcnt
0000:07EE   __ScanTodVector
0000:07EE   __RealCvtVector
0000:07EE   _global_byte
0000:07EF   _global_int
0000:07F2   __atexittbl
0000:0832   ___last
0000:0834   ___rover
0000:0836   ___first
```

Line numbers for first.obj(first.c) segment _TEXT

41 0000:02D8	44 0000:02D8	45 0000:02Ei	46 0000:02E7
59 0000:02E9	64 0000:02F0	65 0000:02F2	67 0000:02F6
68 0000:02FF			

Program entry point at 0000:0100
Warning: no stack

You can see from this link/load map that the compiler includes a number of functions and variables that are not present in the original source code. The majority of these inclusions come directly from the start-up module that the compiler links to your C programs.

The linker produces the message *Warning : no stack*. This is a normal message for a tiny model program.

Now, let's analyze the structure of this simple C program. The following lines from the link_load map indicate that the output from the compiler begins with the code segment, _TEXT, which starts at hex location C000 and continues until hex location 0659 with a length of hex 065A.

Start	Stop	Length	Name	Class
00000H	00659H	0065AH	_TEXT	CODE
00660H	007E7H	00188H	_DATA	DATA
007E8H	007E8H	00004H	_EMUSEG	DATA
007ECH	007EDH	00002H	_CRTSEG	DATA
007EEH	007EEH	00000H	_CVTSEG	DATA
007EEH	007EEH	00000H	_SCNSEG	DATA
007EEH	00837H	0004AH	_BSS	BSS
00838H	00838H	00000H	_BSSEND	STACK

The code segment is followed by the data segment, _DATA. _DATA contains the initialized data values for the program. first.c does not contain any initialized global variables, but you will find that the C start-up module does.

The segment named _BSS is the segment containing the uninitialized global variables. This is where the two global variables from first.c can be found. Following the _BSS segment, you will find the _BSSEND segment, which identifies where the stack can be placed without declaring a specific STACK segment.

From this brief analysis, you can see that a problem exists in the ordering of the segments produced by the compiler. Specifically, the data must precede the code in a DOS device driver.

A structural problem with the output from the compiler indicates the likelihood of a number of problems with the more detailed aspects of the code and data generated by the compiler as well. The only way to determine whether this statement is true is to have the compiler produce assembler language output, then inspect that output for instances that might conflict with the guidelines specified for DOS device drivers.

first.c must be recompiled with the options required to produce assembler language output. The following command is sufficient to accomplish this task:

```
tcc -mt -y -c -S first.c
```

This command requests TURBO C to generate a tiny model program (-mt) that includes line number information (-y), compile it only (-c), and produce an assembler language listing (-S). The assembler language listing will be in first.asm.

We edited first.asm to remove various debugging statements produced by the compiler, and we have reformatted portions of the assembler language program for readability.

A group named DGROUP is defined as being the data segment, _DATA, followed by the uninitialized data segment _BSS. However, the code segment _TEXT is not contained in this group even though the program can grow only to a maximum size of one segment (64K). Furthermore, all named variable references are preceded with the name of the group (DGROUP) to calculate correctly the offset of the variable within the program. However, this is not the case when references are made to locations within the code segment (_TEXT) as indicated by the call to _Function.

Although this might seem like a lot of double-talk, it is really important that you understand some basic concepts concerning just how the compiler is generating code from your C program. The main reason for doing this exercise is to demonstrate that the data segment must be relocated to the beginning of the object file, as we will explain in the next section. Once this has been accomplished, the references to the code segment will be incorrect because the compiler assumes the code segment will always begin at location zero. Therefore we must take corrective action to resolve this problem as well.

5.3 Data Segment Preceding Code Segment

If we are conform to the specification of a DOS device driver, we must change the order of the segments generated by the compiler. In other words, the data segment must be relocated to the beginning of the output file, and the code must be moved to the end of the output file.

The following changes to the assembler output are sufficient to accomplish the desired results:

```
DGROUP  group   _DATA,_BSS,_TEXT
        assume  ds:DGR0UP,ss:DGROUP,cs:DGROUP
```

The changes are simple. First, the code segment (_TEXT) was included in the data group (DGROUP). Second, the assembler is instructed that the code segment register (CS) is assumed to be relative to the data group (DGROUP).

You can make these changes with your favorite editor. However, this type of operation is prone to errors. That's why we developed a utility called **arrange**, which performs these modifications.

arrange also modifies the code references to include that the DGROUP prefix rather than maintaining the compiler's assumption that the code segment always begins at location zero.

5.4 C Stack and Data

The stack a C program uses during execution is established by the start-up module that is linked with the program's object module. Because the DOS device driver is not linked with the C start-up module, some provision must be made to support a stack during program execution. One of the best ways to address this problem is to view it as Interrupt Service Routine (ISR) that must save all registers on entry and establish its own operating environment every time it is executed, and this routine is called DOS_Setup( ).

We will establish a stack for program execution each time the DOS device driver is executed. The stack size will be determined by the setup code contained within the DOS device driver. Typically, a DOS device driver attempts to minimize the usage of resident memory. Therefore the amount of stack space allocated from within a DOS device driver varies from a few hundred bytes as much as one kilobyte.

The stack size can be thought of as a very limited resource. It would be nice to understand just what type of operations require space on this stack. In C any time we make a function call, we use a stack space to record the return address and the parameters being passed to the function. The stack is also used when a function declares any local variables. These variables are allocated on the stack unless we use the static keyword to promote them to a global allocation level. Remember, if you run out of stack space within a DOS device driver anything can happen!

5.5 The C Run-Time Libraries

DOS device drivers are not allowed to use DOS functions or services. The reason for this is that when a DOS device driver is executing, an application has already request a specific DOS function or service to be performed. DOS is attempting to accomplish the specified DOS operation by invoking the appropriate DOS device driver. If that device driver were to request a DOS function or service to be performed, then DOS would have to be reentrant, which is not. Therefore, DOS device drivers are not allowed to use any DOS functions or services.

The problem encountered in DOS device drivers written in C that use the supplied C run-time routines is that those routines might attempt to issue a DOS request. As mentioned above, this is not allowed because DOS is not a reentrant operating system. Therefore the use of C run-time routines must be limited to those functions that do not require DOS intervention. Here is a short list of some of these routines:

* String functions (strcpy, ...)
* Memory movement (memmove, ...)
* Direct console I/O (cprintf, cput, ...)

As a general rule, the list of C run time routines that can be used safely is provided with the documentation for the C compiler. This list will change from compiler to compiler[2].

5.6 DOS Device Driver Header

A DOS device driver is a memory-image file, .COM, that contains all the logic required to realize the device attachment or implementation. Although the device driver file is a standard type of file, it does have one main difference. Typically, .COM files are required to start at hexadecimal location 100. This requirement allows DOS to create a 256-byte Program Segment Prefix (PSP) in memory prior to loading the .COM file itself. If the .COM file were to start at location zero, then when DOS loaded the file it would write over the PSP and the program would not be able to operate.
DOS device drivers do not start at location 0x100. Instead, DOS device drivers start at location zero. Because DOS device drivers represents an extension to the DOS kernel they start at location zero. Therefore DOS has allocated memory and specific internal data structures to manage the location and operation of each device driver in the system. Furthermore, once DOS device drivers are being loaded into memory, their memory addresses do not change. However, .COM files are constantly being loaded into memory, executed, and then removed from memory.

All DOS device drivers must have a DOS device driver header located at location zero. The DOS device header is analogous to the PSP for .COM files. DOS uses the device driver header to link all device drivers into a singly-linked list of device drivers. Therefore, if we were to find the head of the list of DOS device drivers we should be able to see all of the devices in our system.

The DOS device driver header has a specific format. The following C structure describes the format.

```
struct DDH_struct
{
        struct DDH_struct far *next_DDH;
        unsigned int            ddh_attribute;
        unsigned int            ddh_strategy;
        unsigned int            ddh_interrupt;
        unsigned char  ddh_name[8];
};
```

As you can see, the DOS device driver header contains five fields, which were explained in details in chapter 4.

Getting a pointer to the first device in the linked list is shown in the following code:

```
struct DOS_struct far *dos_ptr;
_AX = 0x5200;
geninterrupt (0x21);
dos_ptr = MK_FP (_ES, _BX);
```

DOS_struct has the following format:

```
struct DOS_struct
{
        unsigned char       reserved [34];
        struct DDH_struct       far *ddh_ptr;
};
```

The DOS_struct defined in the above structure contains a far pointer to the beginning of the list of device driver headers, ddh_ptr. Therefore, it easy to obtain a pointer to the DOS_struct from DOS, then traverse the linked list and visit each device driver header. This is done by loading 0X0052 in AX, and performing interrupt 21h. DOS returns the address of the first device driver in the linked list in ES, and BX respectively. The macro MK_FP takes the contents of these registers and builds a far pointer that is usable by C programs.

5.7 DOS Device Driver Requests

In the latest version of DOS, the interface between DOS and the device driver supports twenty six commands. Each of these DOS device driver requests or commands has a specific request format associated with it. However, a portion of the DOS device driver request is common to all twenty six. This common portion is referred to as the DOS device driver request header. The following C structure describes the contents of the common portion of the DOS device request header.

```
struct REQ_struct
{
        unsigned char length;
        unsigned char unit;
        unsigned char command;
        unsigned int status;
        unsigned char reserved[8];
}
```

The length is always 13 (3 bytes unsigned char, 2 bytes unsigned int, 8 bytes unsigned char). The second field in REQ_struct is used for accessing specific units of a block device driver. The third field is the command number shown above. The fourth field indicates the result of the request DOS command. The high-order bit of the status word indicates whether an error occurred. If an error occurred, then the low order byte of the status word contains one of the following error codes.

Whenever there is a return from the device driver command after a DOS call, the status word in the request header is set (for more information on the return code see chapter 4).

5.8 DOS Device Driver Components

5.8.1 The Required Utilities

The tools that is needed to write, compile, and link a device driver is shown in Table 5-1.

Tool	Description
Editor	A word processor or text editor program which allows entering source code into a file and save it. It is also used to modify the text file.
Assembler	A utility used to convert the source assembly language program into relocatable object modules.
Compiler	A utility used to convert source code programs into relocatable object modules
Linker	A linker is a utility which combines one or more relocatable object modules into an executable file.
EXE2BIN	A utility program that converts normal executable files into memory image files. Memory image files are known as .COM files and are required for device drivers.
Arrange	A utility program found on the supplied media, used to arrange the code and the data segments of the .asm files.

Table 5-1: The required tools for writing a DOS device driver.

5.8.2 Segment Headers

DOS device drivers require that the data segment be at location zero in the file, and if it is not at this location we used to move it around. For completeness a header file is included, **drvr_hdr.asm** the file rearrange the segments in the link file, and this file contains assembler language pseudo-operations and no assembler language instructions. Also, it groups all the segments in one group.

5.8.3 Definitions

The file **drvr_dd.h** contains the C structure and definitions that describe the entire DOS device driver environment.

```
struct DEVICE_HEADER_struct
{
 struct DEVICE_HEADER_struct far *next_hdr;
 unsigned int attribute;        /*device driver attribute     */
 unsigned int dev_strat;        /*pointer to strategy code    */
 unsigned int dev_int;          /*pointer to interrupt routine */
 unsigned char name_unit[8]     /*Name/Unit field             */
}
```

it contains also the REQ_struct which represents the DOS device driver command interface to pass commands around in a device driver. Each command has a unique structure in the variable portion of the DOS request structure :

```
struct REQ_struct
{
   unsigned char length;
   unsigned char unit ;
   unsigned char command;
   unsigned char status;
   unsigned char reserved[8];
   union
   {
        struct INIT_struct                  init _req;
        struct MEDIA_CHECK_struct           media_check_req;
        struct BUILD_BPB_struct             build_bpb_req;
        struct I_O_struct                   i_o_req;
        struct INPUT_NO_WAIT_struct         input_no_wait_req;
        struct IOCTL_struct                 ioctl_req;
        struct L_D_MAP_struct               l_d_map_req;
   }req_type;
};
```

Also, it includes the file **mydef.h** which contains definitions for TCP/IP functions value.

5.8.4 Global Data

One of the most important aspects of a DOS device driver is the DOS device driver header. This header must be at location zero in the .sys file and it must be initialized for the device driver to operate correctly. The file **drvr_da.c** allocates and initializes the device driver header, it contains also all globally allocated data. Note, that the template based DOS device driver is initialized as a block device driver.

5.8.5 C Environment

The most critical portion of code is found in **drvr_ev.c** file, this file contains the routines that is called directly by DOS whenever a DOS request must be processed by a device driver.
The code in the **drvr_ev.c** file is critical because it receives the DOS requests and transforms the current DOS environment, typically assembly languages into a usable C environment, **DOS_Setup( )** accomplishes this task. The Strategy and Interrupt functions call **DOS_Setup( )** as soon as they receive a DOS request. **DOS_Setup( )** then saves the current operating environment and creates a new C environment, complete with its own local stack. **DOS_Setup( )** function is the reason why the tiny model is used. **DOS_Setup( )** will not work properly if another compiler model is used.

5.8.6 Commands

The **drvr_dr.c** file contains all the functions for the DOS device driver commands. All functions have the same input parameter: a far pointer to the DOS request structure. Each function in the file performs the specified operations, then sets the appropriate return code in the status word of the DOS request structure, the **Init_cmd( )** function corresponds to the DOS INIT request command which, checks only the number of local drives, and does not any communication procedure because it is executed when DOS is loading config.sys drivers, that is, before TCP/IP batch file is being executed by autoexec.bat. It also points to the end of the device driver code, so that DOS can load another driver starting from this location. The **media_check ( )** which corresponds to the MEDIA CHECK command, the **Build_BPB( )** which corresponds to BUILD BPB command, etc. But, with these functions we can use the TCP communication functions to send and receive data from the server. All DOS device driver commands are implemented as functions into the file **drvr_dr.c**.
N.B. All DOS device drivers must respond to the DOS INIT commands to be functional.

5.8.7 Ending Marker

When DOS issues the INIT request the device driver must respond with its ending address. The code segment follows the data segment because they are rearranged by the arrange utility. Therefore, the ending address of the device driver is located somewhere within the code segment. **drvr_en.c** contains C function named **End_code( )**. This file is linked last and truly becomes the end of the code segment. It is nothing more than a place holder that **Init_cmd( )** uses to determine the end of the template-based DOS device driver, and the function code is described below:

```
unsigned char end_data;
void End_code(void)
{
}
```

5.8.8 Template Overview

Table 5-2 shows the main components of the client device driver.

File	Description
drvr_hdr.asm	Header file in the device driver that rearranges the segments in the link file.
drvr_da.c	next_hdr attribute dev_strat(Strategy function offset) dev_int(Interrupt function offset) name_unit remaining drvr_da.c data
drvr_ev.c	drvr_ev.c data dos_Setup function Strategy function Interrupt function
drvr_dr.c	functions to support DOS requests
drvr_en.c	function to mark end of device driver code

Table 5-2: Device driver components.

5.9 Creating and Loading the Device Driver

The Make Utility

the make utility uses a makefile, and the makefile contains a list of commands and dependencies to build an executable program. The makefile is named **drvr.mak.**

Usually, a directory is being created and all files that are used by the makefile are copied into it. The make command syntax is:

```
make -f drvr.mak
```

More information about the makefile can be found in different books. The output of the drvr.mak is a .sys file, which is the extension of the device driver.

The Arrange Utility

The **arrange** utility expects three arguments:

- the file name that contains the substitute commands
- the input file
- the output file

Below, is an example of the arrange utility which is used in drvr.mak:

```
arrange drvr.arr  drvr_da.asm  m2.asm
```

to arrange the segments in drvr_da.asm. The file drvr.arr contains the data to be modified in the input file drvr_da.asm. m2.asm contains the final output modified file.

Loading the Device Driver

The DOS_DRVR.SYS is like any other DOS device driver. To install a device driver you must have an entry defined in the config.sys file in your system.
The command that is of interest for the device driver in the config.sys file is the DEVICE= command; this command informs DOS at initialization time to load and initialize this new device driver. The following entry is added into config.sys using any DOS editor:

```
DEVICE = DOS_DRVR.SYS /E
```

the /E means to start mapping drive from E and above. This option is in order not to get confused on the clients side which drives are the server's drives but, in this case there will be a lost drives, for example, drive D: if there is only one hard disk on the client machine.

5.10 Putting Client/Server into Work

The program that should be run on a client in order to open a communication line with the server is login.exe. The source code is found in h_login.c. The job of this program is first, to check the DOS version if it is greater or equal to 3.1. Then it checks if the DOS_drvr.sys is loaded in config.sys, otherwise, it gives an error message. The second part is to check if TCP/IP is loaded and determines the TCPIP_INT for the TCP TSR. A point to mention here, that ssocket ( ) and sconnect ( ) were used first in h_login.c to open communication with the server, then set the fd in the device driver data by getting a pointer to the header of the device driver, and add to it the exact value of bytes to reach the variable fd in the data segment. But, later on this code was implemented in drvr_en.c where it is executed by the first command issued by DOS other than INIT command, then a flag is set in order not to execute it again.

Now, the question is how to map every command issued by DOS to the server device driver? The answer to this question is explained in
Table 5-3 which represents exactly how DOS gets the answer from the server driver through the client driver.

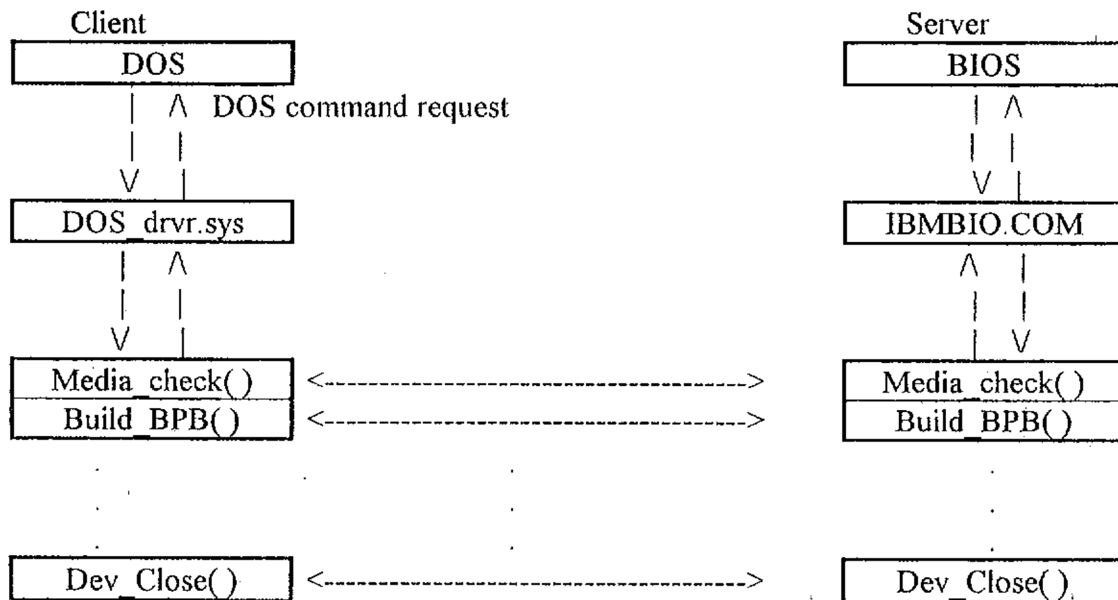


Table 5-3: Commands exchange between client and server device drivers.

After running the login program successfully, the server drives can be accessed if a communication link is established with the server. Once, it is done, every command issued by DOS to DOS_drvr.sys will go first, into Strategy( ) routine than through the Interrupt ( ) routine which call DOS_Setup( ) (function explained before) which in its turn depending on the number of command issued by DOS, call the equivalent routine
(Media_check( ), Build_BPB( ), etc.). Each of these routines or commands maps to the same command on the server. So, all the data given by DOS through the request header is sent to one of the client commands procedure, which sends it to the same command procedure on the server, and on its turn inquires information from the server device driver IBMBIO.COM, and returns it back to the client command procedure which returns it to DOS.

It seems a bit confusing, isn't it? and you are wondering how the server is implemented and how it works? After the server initializes itself, it listens for incoming connection from the clients. Once a connection is established with a client, the server reads the incoming data coming on the form of packet, that contains the command code to be executed on the server, the data length, and finally the data. The server directs this packet to the correct command procedure which retrieves this data, and issues a call to the device driver through the call_dd( ) routine. Then it gets the answer back from the device driver, and returns it to the client and loop back again.

5.11 Layers Global View

If we take a look at the client, and server architecture we found that they are made of different layers. Each layer adds some modifications to the packet received from the upper layer, and sends it to the lower layer until it reaches the physical layer which, and sent across the line to the server. The server receives the packet through the physical layer and starts taking out the data specific for each layer and sends the

packet to the upper layer. Table 5-4 shows the different layers used by the client and the server.

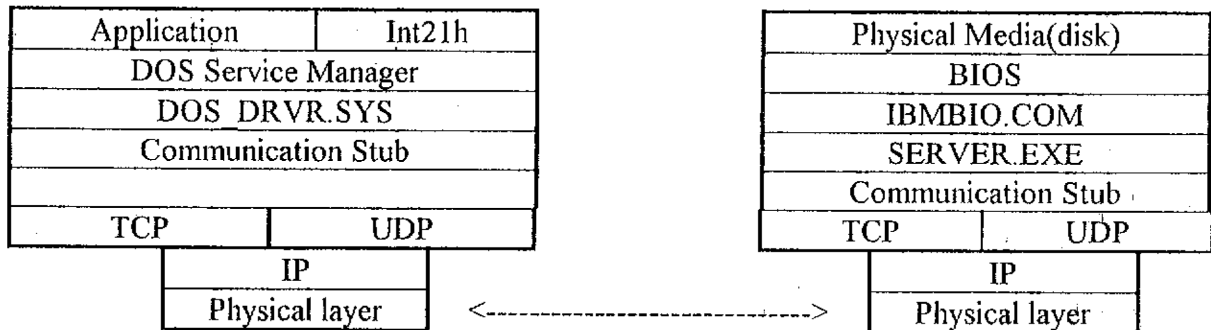


Table 5-4: Client and server different layers.

what happens when a command is issued by DOS, till this later received the answer is explained below:

1- User type any DOS command at the DOS prompt, or use int 21h to access the server's drives.
2- DOS dissects this command into several requests, and calls the device driver DOS_DRVR.SYS for each of these requests.
3- DOS_DRVR.SYS gets each request from DOS, and transforms it to the appropriate DOS command function, which passes it to the communication stub.
4- The communication stub adds some data to the received packet, and makes the necessary modifications before calling the TCP APIs.
5- TCP APIs adds the necessary header to the packet, and passes it to IP.
6- The IP on its turn adds some data to the packet, such as destination, and takes care of the routing, and pass it to the physical layer to be sent across the network line.
7- The server receives this packet through the physical layer which passes it to the IP layer, which checks it and takes out the necessary data and gives it to the upper layer.
8- The TCP layer does what ever is required with the packet such as error checking, synchronizing, etc. and passes it to the communication stub.
9- The Communication stub reads the necessary data, and puts it in a form understandable by the device driver.
10- The server.exe program calls the device driver after setting the BX register, and ES segment to contain the ordered data.
11- The IBMBIO.COM calls the BIOS to perform the real work, which contacts the physical media.

After the IBMBIO.COM gets the required information, it passes it all the way down through the layers, and send it back to the client who extracts the necessary data from the packet and passes it from layer to layer until it returns to DOS.

All this work can be described as if the DOS issues command on a client that is really executed on the server, and the answer is returned back to the client.

Chapter 6

Conclusion

The DOS client/server, DOS-client protocol, and client-server protocol were analyzed. Different server hard disks, multiple clients connections and applicability regarding network topologies were considered.

The client program was tested carefully regarding DOS commands handling, packet transferring, and errors detection. On the other hand, the server program was playing the role of DOS in issuing commands to the device driver and receiving answers. TCP/IP have provided a logical link between clients and server. In addition, TCP/IP proved to be reliable in controlling multiple clients connections, packets generation, and errors detection.

Furthermore, different applications and designs can be applied to the presented model such as new security establishment, client-client communication, files sharing, hard disks and external devices accessing, and printing queues generation..., and much more applications that are classified under the client-server model.

Finally, the presented client-server model acts as a guideline for students who wants to work on client/server and develop different applications for this model. It is a simple model that is easy to understand and modify to satisfy different needs.

Appendix A

Disk/Diskette Internals

In order to learn how to write a disk device driver, you should review the topic of disk internals. DOS supports a variety of disks , with storage capacities ranging from a hundred thousand bytes to hundred of megabytes . In this chapter, we will describe how DOS manages different types of disk storage.

Starting with basic definitions, we will show how data is written to a disk (*disk* here means both hard [fixed] disk and floppies), how DOS organizes the data on the disks, and how DOS determines the disk type. We will distinguish between floppy and hard disk drives and look at some of the special features of hard disk drives. Lastly, we will describe the internal information that is contained on each disk drive and how disk device drivers interact with DOS to access disks.

The Physical Side of Disks

Disks are storage devices that are based on a rotating disk with magnetically alterable surfaces. The surfaces store digital information. Read/Write heads are built into the disk drive to retrieve and store data to the disk drive. Disk drives are also known as *random access devices*, because you can independently position the read / write head to any spot on the disk.

Disk drives come in two different forms. Floppy disks are those types of disks that can be removed from the drive unit. Hard disk drives are fixed and cannot be removed.

Disk Types

Floppy disks are built using flexible materials and are usually made in three sizes: 3 1/2 inches, 5 1/4 inches, and 8 inches in diameter. Information is both recorded on both surfaces; most floppy disks use both surfaces.

Hard disks are built with one or more platters mounted on a spindle driven by a small motor. Each of the platters is magnetically coated on both sides for storing information . A read / write head is assigned to each surface of a platter. These disk heads are mounted on arms that move together and are controlled by another motor.

Connected to every disk drive through a cable is a disk controller; a PC add-on circuit board that provides electrical signals to control the disk and read/write head. The disk - controller board is inserted in a slot on the PC's motherboard, which connects it to the main bus and allows the board to receive instructions from the CPU. The controller is responsible for transferring data to and from the PC and for positioning the read/write head to a desired position on the disk.

Organizing Data on Disk Drives

In this section, we will examine organizing data on disks, storage capacities, sector sizing and numbering , and formatting.

Tracks on a Disk

Each surface of a disk is divided into tracks on which information is recorded. The read/write head assigned to a disk surface is positioned to one of these tracks before a read or write is performed.

Most 5 1/4 - inch floppy disks have either 40 or 80 tracks. There's also another standard format based on 3 1/2 -inch disks. Disks that contain 40 tracks are commonly called *double density disks*. Historically, the original disks for the PC could record at half this density and were called single density disks. With improving technology, the density has increased to 80 tracks; such disks are known as *high density disks*.

Because the surfaces are rigid and easier to control to tighter tolerance, hard disks can have many times the number of tracks on a floppy disk. A 10-MB fixed disk for the IBM PC typically has 305 tracks. When there are two or more platters in a disk drive (the spinning surface is called a platter), the term *cylinder* issued to refer to all tracks that are identically numbered.

Tracks are numbered from 0 to the highest track number for the disk. Each recording surface of the disk is also numbered in this manner.

Raw Storage Calculations

Often you need to calculate just how much capacity there in on a disk. Several specifications can be used to determine the amount of storage. First, you will need the amount of data that can be recorded in one track, which is usually specified in bytes per track. Next, you will need the number of tracks, which is determined by the track density (usually specified as tracks per inch, or tpi) multiplied by the circumference of the recording surface that contains tracks. Finally, you will need the number of recording surfaces. This is usually 1 for a single- sided disk and two for a double-sided one. For hard disks, this number is usually twice the number of platters.

The total amount of "raw" storage on a disk is calculated with the following formula:

Total storage = storage / track * tracks / surface * surfaces

Not all of this storage area is available for your use, because the overhead needed to manage the data stored on the disk is not taken into consideration in this calculation. *Overhead* is a term used to describe the additional information recorded onto the individual tracks that is required for the disk controller to find each track.

Disk tracks are further subdivided into sectors for ease of management; we will describe why this is done shortly. Table A-1 summarizes the various types of disks and the amount of data that can be stored.

Size	3 1/2	3 1/2	5 1/4	5 1/4	5 1/4	5 1/4
Type	floppy	floppy	floppy	floppy	floppy	hard
Density Type	___	___	double	double	quad	___
Density (Tracks/Inch)	135	135	48	48	96	720
Tracks/Surface	80	80	40	40	80	305
Surfaces	2	2	1	2	2	4
Bytes/Track	5,120	10,240	5,120	5,120	5,120	10,416
Total Storage Size	800K	1.6Mb	200K	400K	800K	12Mb

Table A-1: The amount of raw storage available for different types of disks.

Organizing Data into Sectors

Sectors are subdivisions of a circular track; they form the basic unit of storage for disk drives. Using sectors allows you to use a common method for storing data for disk drives of varying sizes.

Whenever a disk is called upon to pass data back to the CPU, the read/write head of the disk is first positioned to a particular track. Then, as the track rotates under the head, the disk controller will scan the sectors that pass by, searching for the desired sector. Once the desired sector is found, the disk controller reads the contents of the sector and returns the data.

The number of sectors in a track sometimes varies. This number depends on the amount of data that can be stored on a track. Version 1.0 of PC-DOS supported only floppy disks, and these were formatted for 8 sectors per track. PC-DOS version 3.0 allows 8, 9, and 15 sectors per track for floppy disks. Some machines , such as the Victor 9000, have formats that put more sectors per outer track than per inner tracks. This is because the larger outer tracks can contain more data than the smaller inner tracks.

For hard disks, the standard number of sectors per track is 17. However, as you will see in the section on the BIOS Parameter Block, DOS can handle just about any number of sectors per track.

Sector Numbering and Sizing

In general, for both PC-DOS, the physical-sector numbering starts at 1. Therefore, for a disk with 9 sectors per track, the sectors are numbered from 1 through 9; for a hard disk with 17 sectors per track, the sectors are numbered from 1 to 17.

Caution: Physical sectors are numbered starting at 1. This scheme is used when the disk BIOS routines are used to format, read, or write sectors. DOS uses a different scheme that numbers sectors beginning with 0. You will see this later, in the section on the BIOS Parameter Block.

As you saw above, the amount of data stored in each sector depends on the amount of storage per track and the number of sectors per track (assuming a fixed density for all the tracks). Because the amount of storage per track is fixed, the sector size can be varied by varying the number of sectors per track. Usually, the sector size is fixed at so many bytes per sector, and the number of sectors per track is calculated by dividing the amount of storage per track by the desired sector size (plus some overhead). When a track is divided into sectors, some storage is lost in defining management and location overhead for each of the sectors. Defining sectors on a track is performed by the formatting process. The formatting information, or *overhead*, reduces the amount of storage available for your use.

The DOS Standard for Sector Sizing

The DOS standard sector size is 512 bytes; however, DOS disk support allows sector sizes of 128, 256, 512, and 1,024 bytes per sector. Sector sizes other than 512 bytes are rare. Because many parts of DOS have been written to assume a sector size of 512 bytes, other sector sizes may not be used under all conditions without modifying DOS. Table A-2 shows the number of sectors for the typical disk types that are supported by DOS.

Formatting Disks

A special program is used to create tracks and sectors within tacks on a disk. This program is known as FORMAT. COM, and it performs a number of additional tasks. The first task is to create a number of sectors on a track. This is repeated for all the tracks of a disk. The second task is to test each sector to ensure that data can be written to and read from the sector. The FORMAT. COM program will create a table for DOS that identifies which sectors are good or bad, so that bad sectors can be ignored. You will see more of this later in the section on File Allocation Tables.

Size	3 1/2	3 1/2	5 1/4	5 1/4	5 1/4	5 1/4
Type	floppy	floppy	floppy	floppy	floppy	hard
Density Type	----	----	double	double	quad	----
Raw Storage	800K	1.6Mb	200K	400K	800K	----
Bytes/Sector	512	512	512	512	512	512
Sectors/Track	9	9	9	9	8	17
Tracks/Surface	80	80	40	40	80	820
Surfaces	2	2	1	2	2	6
Total Sectors	1,440	2,880	360	720	1,440	83,640
Formatted Storage	720K	1.44 Mb	180K	360K	720K	40.84 Mb

Table A-2 : Some of the disk formats supported by DOS.

When the data is organized by sectors, the overhead of identifying each sector results in a small loss of total storage. Typically, this is about 10 percent.

Technical Details of DOS Disk Support

In this section, we will discuss how DOS accesses the various parts of a disk, the File Allocation Tables and File Directory, and the parameters in the Boot Record that describe the disk to DOS.

Disks Supported by DOS

The earliest versions of DOS (1.0) supported only single-sided disks. The next version (1.1) supported double-sided floppy disks. Hard-disk support began with MS-DOS version 1.25 and PC-DOS version 2.00. Prior to these versions, hard-disk support was largely a matter of the disk manufacturer providing custom software routines to access the hard disk. Today, hard disks of all sizes may be added to IBM and IBM-compatible PCs without requiring special software. The use of drivers facilitates the task of adding support for a large number of disks. Table A-3 summarizes the types of disks supported by PC-DOS for the IBM PC.
Special mention should be made of disk types supported by other vendors for non-IBM PCs. MS-DOS can be tailored to just about any machine that uses an 8086/8088 microprocessor, so the number of disk types for non-IBM compatible

DOS Version	Single Side 5 1/4	Double Side 5 1/4	3 1/2	1.2Mb Floppy	Hard Disk 10 Mb	Hard Disk Size
1.0	x					
1.1	x	x				
2.0	x	x			x	
2.1	x	x			x	
3.0	x	x		x	x	10Mb+
3.1	x	x		x	x	10Mb+
3.2	x	x	x	x	x	10Mb+
3.3-5.0	x	x	x	x	x	10Mb+

Table A-3: The types of disks supported by the various versions of PC-DOS.

PC	Disk Type	Size	Description
HP 150	3 1/2 floppy	270K	Single-sided disks
Tandy 2000	5 1/4 floppy	720K	Double-sided 96 tpi disks
DEC Rainbow	5 1/4 floppy	720K	2 single-sided 96 tpi disks
Victor 9000	5 1/4 floppy	1.2 Mb	Double-sided 96 tpi disks

Table A-4 : Disk sizes for other types of PCs using MS-DOS.

machines is large. Table A-4 shows other types of PCs and the disk types supported by MS-DOS.

How Disks Are Organized

DOS is capable of supporting more than one type of disk. This is made possible by requiring that information regarding a disk's specific area defined by DOS.

Each disk must also have additional information stored on it indicating the amount of storage currently used, names of existing files, and other information required for managing the files and disk space. This information is invisible to the user but is a necessary component of all disks.

DOS expects the information on the disk to defined in a certain sequence; therefore, all DOS disks are organized in a uniform fashion. This allows DOS to obtain information about the use of the disk, how space is to be allocated on the disk, and the files in use on the disk.

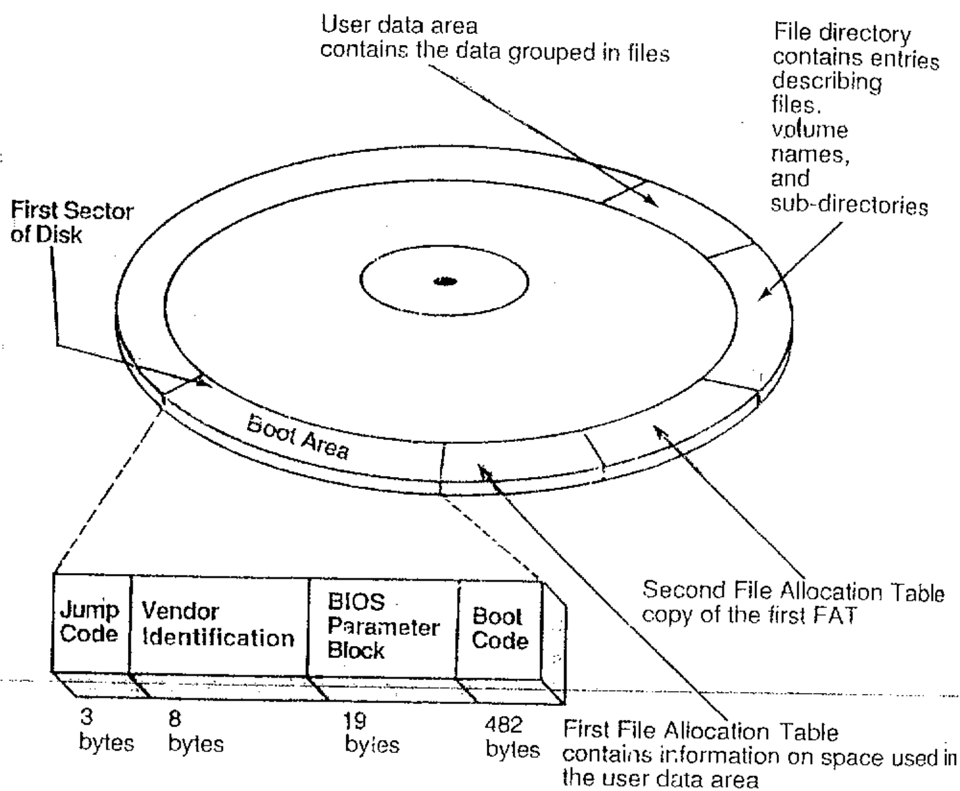


Figure A-1: The relative positions of the four components of a typical formatted disk.

There are four components to a disk layout. The first is the reserved area commonly referred to as the boot record. The second component is the File Allocation Table (FAT), which is used to indicate the usage of space on the disk. The third component is the File Directory, which is used to store the size, location, date, and time information about files on the disk. Finally, the last component is the user data area, in

which the user files are actually stored. The relationships among these four components are shown in Figure A-1.

The Boot/Reserved Area, FAT, and Clusters

The boot or reserved area in the first section on the disk. Because disks vary in their number of sides, tracks, and sectors, DOS needs to determine these disk characteristics the first time it accesses a disk.

DOS assumes that this information describing the disk is always in a certain physical location, usually track 0, surface 0, and sector 1-- the first sector of the disk. Although the boot area is usually only one sector in length, it can be larger. For this reason, this area is now more generally referred to as the reserved sectors area.

Figure A-1 shows the boot area's four parts: a jump code instruction, the vendor identification code, the BIOS Parameter Block, and the boot code area.

The first part of the boot area contains a jump (jmp) instruction. If the disk is a DOS system disk, booting it causes the PC to load the data in the boot area into memory and to execute this jump instruction, which skips over the vendor identification and BIOS Parameter Block areas directly to the boot code.

The second part of the boot area is an 8-byte field that contains the vendor identification. This field is not used or required by DOS. Normally, a PC manufacturer will fill this field with the name of the vendor plus the DOS version on the disk. Examples of vendor identification fields are:

IBM	3.1	PC-DOS supplied by IBM
PSA	1.04	MS-DOS supplied by ATT (6300)
PC88	2.0	MS-DOS supplied by popular clone manufacturer
CCC	2.1	MS-DOS supplied by Compaq
MSDOS	5.0	MS-DOS supplied by Microsoft

The third part of the boot area is the BIOS Parameter Block. This is table of special disk parameters that DOS requires to determine the size of the disk and the relative locations of the FAT and the File Directory. The BIOS Parameter Block is often called the BPB and is always present on every disk. We will describe the contents of the BPB later in this chapter.

The fourth and last part of the reserved boot area is called the boot code area because it contains the actual code for the bootstrap program that starts the PC. This bootstrap program has the job of "pulling itself up by the bootstraps"; in the case of DOS, this means getting DOS to bring itself into memory. Although this bootstrap code is always present in the reserved boot area, regardless of whether the disk contains the DOS system files, it is meaningful only when the disk has been set up as a system disk.

Typically, a system disk is created by the FORMAT program supplied with MS-DOS. If the FORMAT command is executed with a special command switch)usually/S), two additional files will be copied to the disk. These files (typically IO.SYS and

MSDOS.SYS) contain the code for the MS-DOS operating system and are hidden from you; they do not appear in a directory listing of the disk. However, the bootstrap program knows they are there and will load them into memory when the disk is accessed at system start-up time. When a disk has been set up to make it possible to boot from that disk, the disk is referred to as a system disk.

Whenever any disk is formatted for use by the DOS FORMAT program, the four sections comprising the boot area are written to the reserved area of the disk, which always begins at the first sector of the disk.

Clusters as the Unit of Storage for a File

Before we describe the File Allocation Table, you need to know how sectors are used to hold data. When your program writes new data to a disk file, DOS needs to find an unused sector on the disk in which to store the new data. Conversely, when your program reads from a disk file, DOS needs to locate the sector on the disk in which the data is stored. DOS requires each disk to have a File Allocation Table in order to keep track of where sectors for a file are located.

Disk Type	Sectors per Cluster
3 1/2 double-sided floppy	2
5 1/4 single-sided floppy	1
5 1/4 double sided-floppy	2
10Mb hard disk	8
20Mb hard disk (AT)	4

Table A-5: The typical cluster sizes for different types of disks.

Keeping track of files on a sector-by-sector basis can be inefficient, however. For example, a 10Mb hard disk has more than 20,000 sectors, and keeping the location of each would make the File Allocation Table very large. Searching this table would take a relatively long time. If the File Allocation Table were smaller, the searches would be faster, and, as a result, the file accesses would be faster. A better solution would be to group sectors together in a pool so that when a new space on the disk is required, a group of sectors is allocated for the file. This concept of grouping is called clustering sectors; it allows DOS to be more efficient in terms of the memory required to manage the File Allocation Table. A cluster is simply a fixed number of sectors; clusters add a second layer of organization and make access easier.
Whenever a file requires disk space, DOS allocates a single cluster and marks the File Allocation Table to indicate this. Clusters (also called allocation units) are the basic units of storage for disk files. The number of sectors per cluster is determined by the disk type and is established by the FORMAT program when the disk is formatted. Table A-5 shows the cluster sizes for different disk types.

The File Allocation Table

Let's learn how the File Allocation Table works.

The File Allocation Table (FAT) is the section of the disk that stores information on disk-file space usage. This table contains information on all the clusters that are unassigned (free for allocating to files), assigned (those that are in use by a particular disk file), or marked as bad (not usable because of media effects).

Note that although the FAT records information on disk space used by your files, the boot area, the two FATs, and the File Directory areas are not themselves represented by clusters in the FAT.

Within the FAT there is an entry for each available cluster on the disk. A floppy might have over 700 clusters. These entries indicate whether the cluster is in use, free, or bad. Bad clusters are found through the FORMAT program some good sectors are less.

As we said earlier, there are two identical copies of the FAT. The second copy provides some insurance against the possibility of the first copy being damaged. This is an old trick that has been borrowed from other operating systems. However, DOS doesn't use the second copy to fix the first if it is damaged.

Recording Clusters in the File Allocation Table

As you saw earlier in this chapter, when a disk file grows, DOS allocates space on the disk in clusters rather than one sector at a time. This causes the FAT to be updated to indicate that a previously free cluster is now in use. Conversely, when a file is deleted, the clusters once occupied by data are marked in the FAT as being free again.

As a file grows, DOS allocates clusters of disk space, and the use of these clusters is marked in the FAT. The list of clusters that form the disk space used by the file is called a *chain,* because of the way that DOS stores the cluster information in the FAT. You will see more of this shortly.

FAT entries contain a value to indicate the status of each cluster. the cluster may be reserved for use by DOS, free for allocation, bad, or in use. A cluster is in use when it is part of a chain. The values of the FAT entries are listed in Table A-6.

For disk sizes of 10Mb or smaller, the size of the FAT entry is 12 bits in the length, or 3 hex digits. For disks larger than 10Mb, the FAT entry is 16 bits long, or 4 hex digits.

The first available space in the user-data area of the disk is the first cluster, which is assigned a cluster number of 2. The reason it is not called 0 or 1 is that the first two entries in the FAT, normally cluster 0 and 1, are reserved for a media descriptor. A media descriptor is a value that uniquely identifies a particular type of disk and allows DOS to distinguish a single-sided 5 1/4 -inch disk from a double-sided one. You will see more of this media descriptor in the section on the BIOS Parameter Block. Figure A-2 shows the entries in the FAT that point to or represent the clusters in the user-data area.

12-bit Entry	16-bit Entry	Cluster Description
000h	0000h	Free
001h-fefh	0001h-ffefh	In- use
ff0h-ff6h	fff0h-fff6h	Reserved
ff7h	fff7h	Bad
ff8h-fffh	fff8h-ffffh	End of cluster chain

Table A-6: The various FAT entries and what they mean .

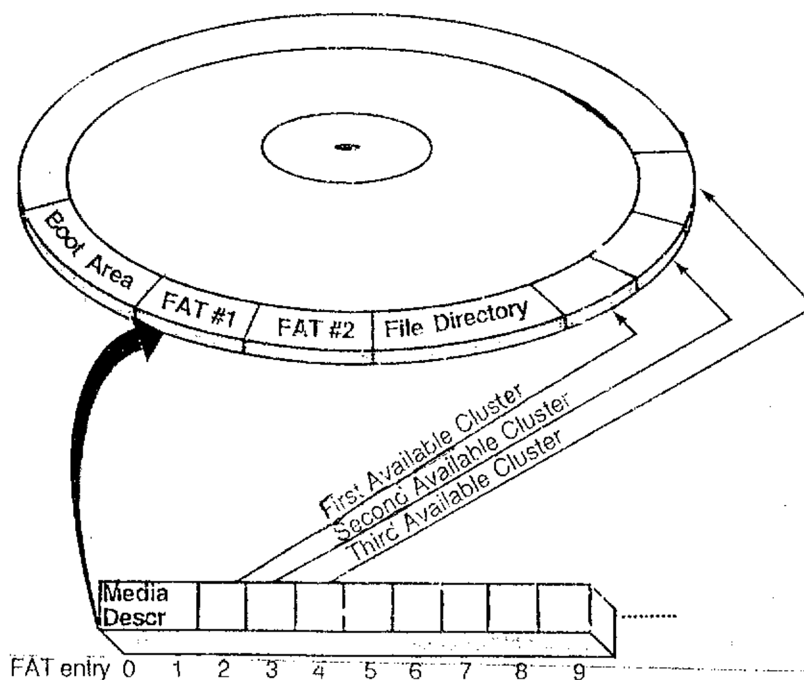


Figure A-2: The relationship between FAT entry and cluster. Each cluster is assigned a position in the FAT and will indicate whether the cluster is part of the chain (in use) , free, bad, or reserved. Note that the clusters are numbered starting at 2.

Clusters, Chains, and the FAT

Suppose a file were large enough to require two clusters of disk space. DOS could simply mark each of two entries in the FAT with a value to indicate which clusters were in use, but this wouldn't allow DOS to determine which cluster was first and which was

second in the table. It would also be difficult to distinguish this particular file's use of the disk from that of another file. It follows that just marking used clusters via the FAT is insufficient for keeping track or what files exist where on the disk; we need a better method.

Consider the following: when the first cluster is allocated to the file, we could store the cluster number outside the FAT, in the file directory. (We will explain later in this chapter what the exact format of the File Directory entry is for each file, but let it suffice now to say that the disk directory will maintain, for each file on the disk, information about the file, including its name and starting cluster number.) Then, as the file grows and the second cluster is allocated, we could use the FAT entry for the first cluster to note which cluster was assigned at the second cluster. For example, if the file used clusters 5 and 10 , we would note (outside the FAT) that the file's first cluster was cluster 5; then in the fifth entry of the FAT, we would store the number 10 to indicate that the next cluster in the file was cluster 10. It follows that if the file grew larger, thus requiring another cluster, we would find a fee (unallocated)cluster in the FAT and store its number in the 10th entry of the FAT. This could continue indefinitely , or at least until there were no more available clusters to be found. In all cases, the last cluster allocated to the file would always have a special value in it to indicate that there were no more clusters following it. This value would represent the end of the file.

The concept of having each cluster essentially point to the next cluster in use by a file is called a *cluster chain*. The idea is that the contents of each FAT entry in use contains a value (also called a pointer) that points to the next cluster, unless the FAT entry represents the last cluster for a file, in which case it would contain an end-of-file indicator. The only thing we would then have to know for a file to find all its sectors is the number of the first cluster assigned to it.

As mentioned earlier, the first cluster assigned to a file is stored in the most sensible place: the File Directory.

Figure A-3 shows how each FAT entry points to the next, thus forming a chain. The start of the chain, or the first cluster, is kept in the file Directory with the entry for the file *myfile* . It contains the value of 4, which means that the first cluster of the file in the FAT is cluster number 4. The entry in the fourth FAT position contains the value 5, which indicates that the next cluster is cluster number 5. At the entry of cluster number 5 we find the value 6, which points to cluster number 6 as the next cluster. Finally , at entry number 6, we find it contains an *ffffh*. This marks the end of the clusters allocated for *myfile*. Thus, *myfile* is composed of clusters 4, 5, and 6 and is three clusters in length .

The Number of FATs Is (Almost) Always Two

The number of FATs is normally two, as shown in figure A-3. When DOS updates the FAT, the first copy is updated, and then the second copy. As we said earlier, using a second identical copy of the FAT provides insurance against the first copy being damaged. The theory is that if the first copy is bad, then the operating

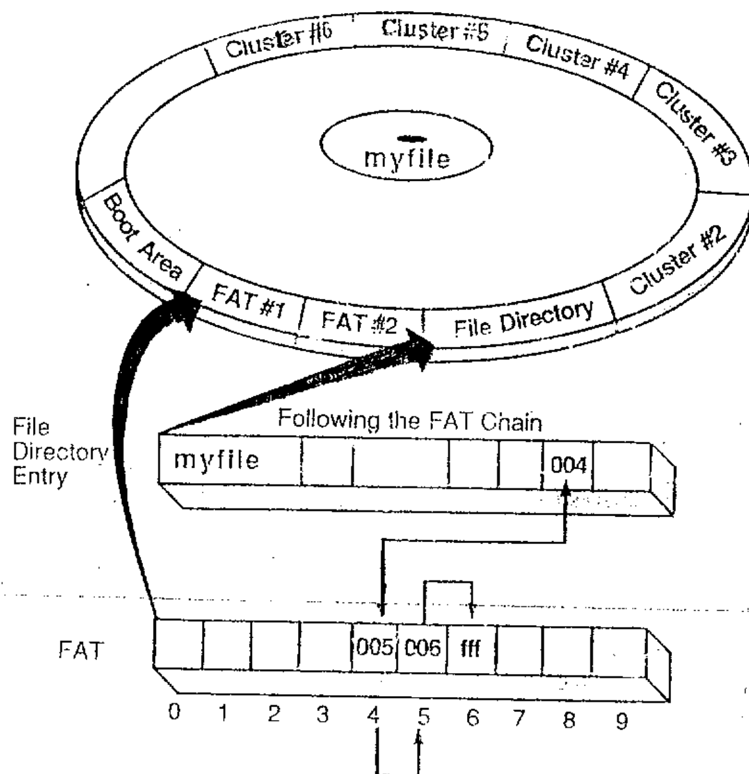


Figure A-3: The clusters used by myfile.

system will use the second copy. Without this mechanism, a damaged FAT would render the disk inaccessible. In practice, however, with a PC, if the first copy of the FAT is damaged, DOS doesn't use the second copy of the FAT to access file information, and the entire disk is not usable. The authors of DOS simply forgot to implement a means to fix the FATs.

Because DOS really uses only one FAT, disks can be built with only one of them. To build disks with only one FAT, you cannot use the standard DOS FORMAT program, which build two FATs on each disk to be formatted. You will need to write a special FORMAT program to build only one FAT on each disk.

The specification of the number of FATs is defined in the BIOS Parameter Block. The overhead of a second copy and the necessity of always updating this second copy can be eliminated if you specify only one copy of the FAT.

The FATs are built for each disk during the formatting process using the FORMAT program. Each entry in the FAT is set to 0 if the corresponding cluster is available for data storage. A FAT entry is marked bad if the corresponding cluster has one or more sectors that are not usable. This occurs when read or write errors are found during the formatting of the disk.

The File Directory

As shown in figure A-3, the File Directory follows the boot area and the FAT and contains the names for all disk files, names for subdirectories, and the volume label.

The file Directory itself is a variable number of entries specified for the disk. Every File Directory entry requires 32 bytes; thus, a 512-byte sector will have 16 such directory entries. The exact number of directory sectors in the number of sectors is 0. Thus, the number of files a File Directory can have is dependent on the type of disk used. Table A-7 lists the disk types and the number of file entries possible. Popular double-sided disks allow 112 entries in the directory, and the hard disk allows 512.

The fields for each File Directory Entry are described in table A-8

Filename The filename field contains a file name that is up to 8 bytes (or characters) in length and is left-justified in the field. DOS expects file names that are less than 8 bytes to be filled out with blanks. It a file has been deleted, the first byte of its filename field is changed to a hex E5. This signifies to DOS that the entry is available for reuse. When a directory entry has never been used, the first byte of the file name field will contain a hex 00.

The distinction between a deleted file name entry and an unused entry is that during directory searches DOS will stop when it encounters the first hex 00 in the first byte of any filename field but continues when it encounters a hex E5, which is merely a deleted entry. If a deleted entry contained a hex 00 in the first

Directory Entries	Directory Sectors	Description
64	4	Single-sided disks
112	7	Double-sided disks
224	14	AT high-density disks
512	32	Hard disks

Table A-7 : The number of File Directory entries and the number of directory sectors for each type of disk.

Position, DOS would have to search all the directory sectors, because it could not distinguish between a deleted file and a entry that had never been used.

Filename Extension The filename extension is an optional field; files may or may not have extensions. Filename extensions are up to 3 bytes in length and, like the filename field, must be left-justified in the field and right-filled with spaces.

Start	Length	Description
0	8	File name
8	3	File name extension
11	1	File attribute
12	10	DOS reserved
22	2	Time of last update or creation
24	2	Date of last update or creation
26	2	Initial allocation unit/cluster
28	4	File size

Table A-8 : the File Directory entry consists of eight fields.

File Attributes File attributes tell what kind of file this is: read/write, read-only, hidden, etc. Table A-9 describes each of the attributes that are possible for a File Directory entry.

Value	Description
00h	Normal read/write file
01h	Read-Only file
02h	Hidden
04h	System file
08h	Volume label
10h	Subdirectory
20h	Archive bit

Table A-9: The various attribute bits that can exist for File Directory entries.

Setting the attribute for read-only prevents a modification of the file through DOS standard file write calls. The hidden attribute will prevent a display of the entry when the DIR command is issued. The attribute for system file is set for the special DOS files that reside on a system disk (IO.SYS and MSDOS.SYS). These two files are brought into memory during a boot of the PC. The attribute for volume label indicates to DOS that the File Directory entry is not a filename but a volume name. The attribute for subdirectory indicates that the file name and extension entry is the name of a subdirectory. The archive bit indicates to DOS that when the BACKUP.COM utility is used to off-load files from the disk, the contents of this particular entry are to be written out. Once of the file is backed up, the archive bit is turned off.

Field	Hex Offset	Decimal Offset	Bits within Offset
Hours	17h	23	7 through 3
Minutes	17h	23	2 through 0
	16h	22	7 through 5
Seconds	16h	22	4 through 0

Byte	<--23-->		<--22-->	
Bits	1	11		
	5	10	54	0
Value	hhhhhmmmmmmsssss			

Table A-10: How to decode the 2-byte time field.

Time of Last Update or Creation Whenever a file is created, the time of creation of the file is entered into the File Directory entry. This includes all directory entries, such as file names, subdirectories, and the volume label. If a file has been updated, this file directory will be updated to reflect the time of the last update. This is not true for subdirectory entries; additions within the subdirectory do not cause an update of the time for the entry. The 2-byte time field is described in table A-10.

Date of Last Update or Creation The date of last update or creation is set with the file-creation date or the date of the last modification. This 2-byte field is similar to the time field except for the date. Table A-11 describes the 2-byte date field.

Field	Hex Offset	Decimal Offset	Bits within Offset
Year	19H	25	7 through 1
Month	19H	25	0
	18H	24	7 through 5
Day	18H	24	4 through 0

Byte	<--25-->		<--24-->	
Bits	1			
	5	98	65	0
Value	yyyyyyymmmmddddd			

* Year is years since 1980

Table A-11: How to decode the 2-byte date field.

Initial Allocation Unit/Cluster The initial allocation unit or cluster field contains the cluster number of the first cluster allocated to the file. For subdirectories,

this is the cluster that will contain the File Directory for the entries in the subdirectory. Table A-12 indicates the format for the start cluster number.

File Size The file-size field contains the size of the file in bytes. It is a double-word entry with the words reserved and the bytes within each word reserved. This double word allows file sizes of up to 32 bits, which is much larger than the DOS limit of 32MB. You will see why DOS has this limit in a later section of this chapter. Table A-13 describes the file-size field.

Hex Offset	Decimal Offset		Description
1AH	26		Least significant
1BH	27		Most significant
Byte	<– 27 –>	<– 26 –>	
Hex value	0X	XX	

Table A-12: How to interpret the start cluster number for the File.

Hex Offset	Decimal Offset				Description
1CH	28				Low-order word Least-significant byte
DH	29				Low-order word Most-significant byte
EH	30				High-order word Least-significant byte
1FH	31				High-order word Most significant byte
Byte	<–31–>	<–30–>	<–29–>	<–28–>	
Hex value	XX	XX	XX	XX	

Table A-13: The 4-byte file-size field. Note that the bytes are reserved in each field and the words are reserved.

Disk Sizing

In previous sections of this chapter, you have seen the different sections that comprise a DOS disk. We will now cover the various aspects of DOS disk sizing, including 12- or 16-bit FAT entries. Then we will see how to calculate the number of clusters for a disk. Lastly, you will see how DOS limits the size of disks.

FAT Entries: 12 or 16 bits?

As we saw earlier, FAT entries are either 12 or 16 bits in length. That length will depend on two factors: the capacity of the disk and the cluster size. You will need the size of the disk in sectors and the cluster size in number of sectors per allocation unit.

Disks will use 12-bit FAT entries until it is no longer possible to store cluster numbers in a 12-bit quantity. FAT entries of 12 bits can contain a number up to 4,096 (0 to fffh). Subtracting the 16 values that constitute reserved, bad, and end-of-file indicators (see table A-6) yields a maximum of 4,080 clusters. Because clusters are numbered from 2, this results in a range of 2 to 4,080 or 4,079 clusters. If the number of clusters exceeds 4079, 16-bit FAT entries are used to mark each cluster.

For example , if a disk used 8 sectors of 512 bytes each per cluster, and the maximum number of clusters is 4,079, the largest disk using 12-bit FATs would be 16Mb (512bytes * 8 sectors/cluster * 4079 clusters). Therefore, to make life easier, disks larger than 10Mb use 16-bit FATs.

Note that, whether 12-or 16-bit FATs are used, the FAT, File Directory, and the Boot Record are not counted in the total number of clusters available. See table A-14 for a summary of the typical cluster and overhead values for various types of disks.

Disk size	5 1/4	5 1/4	5 1/4	5 1/4
Disk Type	Floppy	Floppy	Hard	Hard
Surfaces	1	2	Varies	Varies
Disk Capacity	180K	360K	10Mb	20Mb
Total # sectors	360	720	20K	40K
Sectors/cluster	1	2	8	4
Maximum clusters	360	360	2560	10k
12/16-bit FATs	12	12	16	16
Boot area sectors	1	1	1	1
FAT sectors	2	2	8	40
# FATs	2	2	2	2
Total FAT sectors	4	4	16	80
Directory entries	64	112	512	512
Directory sectors	4	7	32	32
Overhead sectors	9	12	49	113

Table A-14: The various calculations for determining the size of the FAT entries and the amount of overhead the disks can have.

DOS Disk Size Limits

PCs have grown in every way, and disk storage is no exception. The original hard disks of 10Mb have given way to 420- and 540Mb drives as standard equipment. DOS is extremely versatile in its handling of disks, but there are some limits built into the software.

The critical number that limits the amount of disk storage per disk drive is the total number of sectors per drive. This number is contained in a singe word that allows for a maximum of 64K sectors. With a sector size of 512 bytes, this yields a maximum disk size of 32Mb.

DOS can provide support for disks that are larger than 32Mb in two ways. The first way is to use larger sector size. For example, using a sector size of 1,024 bytes moves the disk size limit up to 64Mb. However, this requires special software that changes the DOS system files to override the default 512 bytes per sector. The second method is much more easier. DOS offers the capability to divide the hard disk into one or more partitions. Each partition of the disk is treated as if it were a separate and distinct physical drive. Thus, you can have multiple 32Mb partitions on one disk.

Beginning with DOS 4, the maximum size of a disk partition is no longer limited to 32 Mb. The location within the BIOS Parameter Block specifying the number of sectors per disk was expanded from a single word to a double word, thus allowing disk partitions in excess of 500Mb. You will see more of the disk partition in the next sections.

Critical Disk Parameter

With a large variety of disks to support, DOS needs a mechanism to determine the logical and physical characteristic of each disk in the PC. These disk parameters must be recorded on the disk and ready by DOS before the first access. The best location is within the Boot Record, because it is common to all disks and it is always at the beginning of the disk.

We will examine the disk parameters stored on each disk by taking a closer look at the Boot Record.

The Boot Area Revisited

As you may recall, the boot area is the first part of a disk or, in the case of a partitioned hard disk, the first area in the partition. As we discussed earlier, the boot area contains a 3-byte jump instruction, the vendor identification, the BIOS Parameter Block, and the boot code (see figure A-1).

The BIOS Parameter Block

The 19 bytes that make up the BIOS Parameter Block (BPB) contain more information that allows DOS to understand how the disk has been built. The BPB contains physical information about the disk media, as well as the location and sizes of the FATs, the File Directory, and the user data area.

Table A-15 shows the format of the BPB. Names or labels are assigned to each field to make it easier to refer to these fields.

Name	Start	Length	Description
SS	0	2	Sector Size in bytes
AU	2	1	Allocation Unit size (sectors
RS	3	2	Number of Reserved Sectors
NF	5	1	Number of FATs on this disk
DS	6	2	Directory Size (number of fil
TS	8	2	Number of Total Sectors
MD	10	1	Media Descriptor
FS	11	2	FAT Sectors (each FAT)
ST	13	2	Number of Sectors per Track
NH	15	2	Number of Heads
HS	17	2/4 *	Number of Hidden Sectors
LS	21	4*	Large Sector Count
*=DOS 4.0-5.0			

Table A-15: The fields that comprise the BIOS Parameter Block (BPB).

The BPB is read off each disk by DOS before the very first access. As you will see, the values of the BPB allow DOS to translate physical to logical sectors and vice versa. Additionally, the FATs, File Directory, and the user data can be found using the BPB.

Let's examine each of these fields one at a time.

Sector Size (SS) The sector size field contains the number of bytes per sector for this media. Although possible sector sizes are 128, 256, 512, and 1024 bytes per sector, DOS doesn't make full use of this parameter. There are numerous places in the BIOS itself that assume sector sizes are 512 bytes per sector.

Allocation Unit Size (AU) As we mentioned above, a cluster, or allocation unit, is the basic unit of DOS disk storage and represents a certain number of sectors.

Reserved Sectors (RS) This field contains the number of reserved sectors for the disk. Recall that each floppy disk or hard- disk partition or a reserved or boot area. This parameter specifies To DOS how many sectors are reserved as the boot area. This field generally contains a value of 1 and is always at the beginning of the disk or the partition.

An important point to note here is that in DOS, sectors are numbered starting at 0. You may recall that the BIOS routines use a sector- numbering scheme that starts at 1. You will see how DOS uses sector numbering in the section called "Hidden Sectors."

Number of FATs (NF) The number of FATs for a disk, usually two, is contained in this field.

Directory Size (DS) This field contains the maximum number of files in the File Directory. The size of the File Directory in sectors will be dependent on the number of files and the size of each sector. Because each file requires a 32-byte entry in the File Directory, and because the number of bytes per sector is contained int the sector size (SS) field, dividing the sector size by 32 gives the number of directory entries per sector. Then dividing the directory size by the number of directory entries per sector will give the number of directory sectors. This number is rounded up if necessary.

Normally, 512-byte sectors are used, so 16 directory entries are available per directory sector.

Total Sectors (TS) The number of total sectors is the total size of the disk in sectors. This number must include the sectors in the boot or reserved area, the two FATs, the File Directory, and the user data area. Because this word can contain a number equal to 64K, the largest disk that DOS can support is 32Mb using 512-byte sectors. For hard disks, this number is the same as the number that appears in the partition table as the last entry.

For disks larger than 32Mb, using DOS version 4.0 or greater, this field is set to 0 and the actual sector count is specified in the large sector (LS) field.

Media Descriptor (MD) The media descriptor field is a single byte that describes the disk for DOS. Table A-16 explains the various media descriptor bytes.

FAT Sectors (FS) The FAT sectors field contains the number of sectors in each FAT. DOS will use this number to calculate the total number of sectors occupied by the reserved sectors (boot area) and the FATs to determine the start of the File Directory.

Sectors per Track (ST) This field contains the number of sectors per track for a disk. For floppy disks, this number is 8,9,15, or 16. For hard disks , this number is usually 17.

Number of Heads (NH) This field contains the number of heads or usable recording surfaces for the disk. This value is 1 for single-sided disks and 2 for double -sided disks.

Hex Value	Description
f8h	Hard disk
f9h	Double- sided 5 1/4 - inch disk (15 sector HD)
	Double -sided 3 1/2 - inch disk
fah	RAM disk (used by columbia Data Products)
fch	Single -sided 5 1/4 - inch disk (single density)
fdh	Double -sided 5 1/4 -inch disk (9 sector)
feh	Single -sided 5 1/4 - inch disk (8 sector)
	Single -sided 8 - inch disk (single density)
	Single - sided 8-inch disk (double density)
ffh	Double -sided 5 1/4- inch disk (8 sector)

Table A-16 : The various values for the media descriptor field.

The value for hard disks will vary depending the hard-disk drive. Typical values range from 2 to 6.

Hidden Sectors (HS) The field that contains the number of hidden sectors for the disk typically is used for partitioning hard disks. Hard disks have the ability to be partitioned into several independent logical drives. In order for DOS to locate the start of a partition, it needs to know the number of sectors from the beginning of the disk to the start of the partition that is being used. The sectors preceding the active partition are known as the hidden sectors, because they are invisible to the active partition. The number of hidden sectors is an offset that is added to the number that is calculated for file operations that are within the active partition to derive the precise physical location on the disk. This is shown in figure A-4.

Each of the partitions is treated by DOS as a contiguous block of sectors starting with sector 0, even thought it is not the absolute 0th sector. Do not confuse this with the physical sector scheme, in which sectors are numbered starting at 1, repeating the sector numbering for each track. DOS partitions start at sector 0 and do not repeat any the sector numbers.

The number of hidden sectors is always 0 for floppy disks, because there is no partition. For hard disks, the number of hidden sectors for each partition will depend on the size of the preceding partitions (each partition has its own BPB). The first partition will generally have 17 hidden sectors, because the first track is occupied by the partition sector and the first must start on a track boundary; therefore, the existence of the partition sector forces the first partition to be on the second track, or 17 sectors from the beginning of the disk.

This field is 2 bytes long for DOS versions up to 4.0. To accommodate larger disks, DOS 4.0 and 5.0 extend this field to 4 bytes.

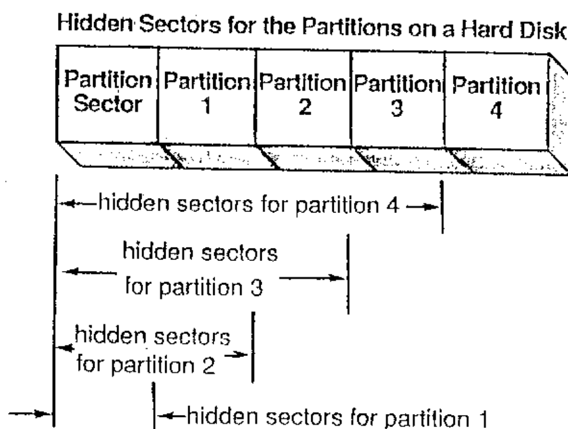
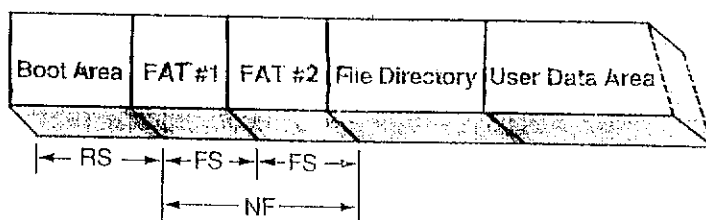


Figure A-4: The number of hidden sectors for the four partitions of a hard disk.

Large Sectors (LS) This field is used by DOS version 4.0 or greater to specify the total number of disk sectors if the disk is larger than 32Mb. In addition, if this field is used, then the total sector (TS) field must be set to 0. This field is set to 0 when the disk size is less than 32Mb



Start of:	**Formula for sector number**
Boot Area:	Sector 0
FAT #1:	Sector RS
FAT #2:	Sector (RS + FS)
File Directory:	Sector (RS + (NF*FS)
User Data Area:	Sector (RS + (NF*FS) + (DS/(SS/32)))

Where:

RS	is the number of hidden sectors
FS	is the Fat Size in sectors
NF	is the number of FATs
DS	is the number of files in the File directory
SS	is the number of bytes per sector
32	is the size of each File Directory entry

Figure A-5: DOS calculations of the start sectors for the FATs, the File Directory, and the User Data Area.

Using the BPB to Find Information

The BPB that must exist on each disk allows DOS to find the important and necessary parameters about the physical characteristics of the disk. For example, DOS can divide the total sector count (TS) by the number of sectors per track (ST) to determine the total number of tracks for the disk or partition.

In addition, the BPB contains enough information for DOS to determine where the FATs, the File Directory, and the user data area are located. Because the sizes of each of these sections of the disk can be found in the BPB or calculated, it is a simple matter for DOS to add up the space occupied by the user data area. This is shown in figure A-5.

Table A-17 shows typical values that are found in the vendor identification and the BPB for a 5 1/4 inch single-sided disk.

DOS Disk Device Drivers

You are probably wondering why we have gone to such detail in describing the FATs, BPBs, and so on. This detail is required to help you understand how DOS interacts with a disk media, so that our disk driver will make sense. It is also necessary to look at the other side of the disk interface, from DOS and the device driver. This is done in the next sections.

DOS and the Disk Device Driver

Whenever DOS needs to read or write to the disk, the standard disk device driver (the one that is loaded into memory with DOS) is called. In addition to read or write calls, DOS makes some calls to the disk device driver to get answers to questions about the disk.

Which Disk Is It?

DOS recognizes that disks fall into two categories: those that are removable and those that are not. Removable disks are the familiar floppy disks that can be removed and replaced easily. Nonremovable disks are, for the most part, hard or fixed disks. Another type of nonremovable disk is a RAM disk. A RAM disk uses memory to store data.

During disk operation, DOS always checks to see whether the disk has been changed. For nonremovable disk, there are fewer checks than for disk units that contain removable disks. DOS perform this check through a call to the DOS Media Check function. Recall from the previous sections of this chapter that all disks have a media descriptor. DOS uses to identify the disk and to check whether the disk has changed. For example, if you have been using a single-sided disk, the media descriptor would be FCh. Then, if you

swapped a double- sided disk for a single-sided disk, DOS would update the media descriptor and it would contain FDh.

Field	Typical Value
Vendor Identification	MSDOS 5.0
BIOS Parameter Block (BPB)	
Sector Size in Bytes (SS)	512
Allocation Unit size (AU)	4
Number of Reserved Sectors (RS)	1
Number of FATs (NF)	2
Directory Size in files (DS)	512
Total Sectors for disk (TS)	0
Media Descriptor (MD)	F8
FAT Size in sectors (FS)	81
Sectors per Track (ST)	17
Number of Heads (NH)	5
Number of Hidden Sectors (HS)	17
Large Sectors (LS)	82943

Table A-17: The typical values found in the vendor identification field and the BIOS Parameter Block for a 40Mb hard disk.

However, this is not full-proof method of determining if the disk has changed-- you could fool DOS by changing to another single-sided disk! Therefore, you cannot rely on the media descriptor as the only method of determining whether a disk has changed.

The only place to determine whether a disk has changed is within the disk device driver. DOS will pass the media descriptor of the disk it has worked on to the disk device driver. The disk device driver, in turn, will determine whether the disk has changed by comparing the particular disk parameters; it then will return this information to DOS.

If the disk has been changed, DOS cannot assume that the FATs, the File Directory, and the user data area are still in the same relative locations. Recall that single- and double-sided disks have different number of sectors for the FATs and the File Directory. Thus, another function of the disk device driver is to return to DOS the BPB for any newly inserted disk. This allows DOS to calculate the positions of the FATs and File Directory for the new disk.

In short, each disk access by an application can cause DOS to perform a media check on the disk. If the disk has changed, DOS will request the BPB for the new disk from the disk device driver so that it can know where everything is stored.

At this point, a real-life example might help illustrate what happens between DOS and the disk device driver. Let's assume that you have inserted into the B: drive a disk that has just been formatted. Then you issue the following DOS command:

DIR

Here is the output that appears on the screen:

```
A>DIR  b:

    Volume in drive B has no label
    Directory of B:\
    File not found
A>
```

Even for this tiny amount of information, DOS has to perform many steps. After the DIR command is issued, DOS has to check whether the disk in drive B: was accessed. Then DOS has to read the directory sectors for the volume label and the file information. Note that the File Directory sectors may be read twice; pass 1 searches for the volume label, which doesn't have to be in the first directory sector; pass 2 retrieves the file names. Lastly, DOS reads the FAT for the amount of space used on the disk. This process is shown in Figure A-6.

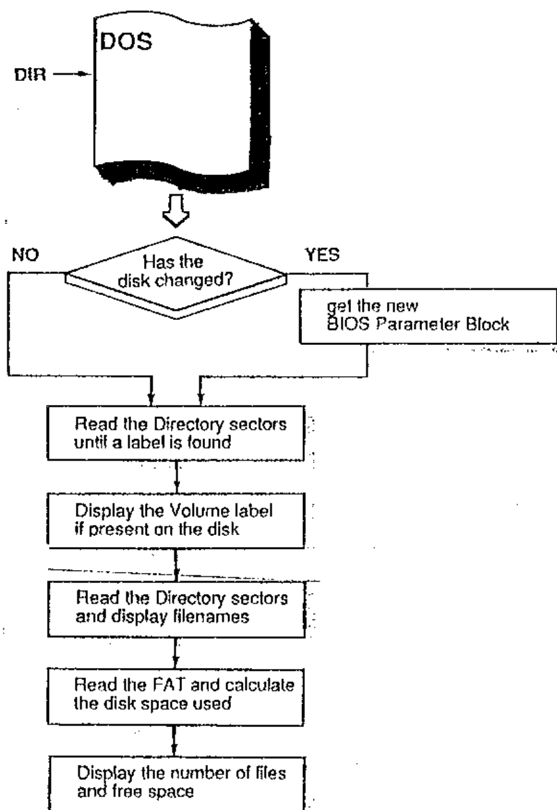


Figure A-6: The steps DOS takes to display the contents of the disk on a DIR command.

DOS	Disk Device Driver
Media Check ------------ >	Has the disk change ?
Yes < ---------	Newly formatted disk in B: therefore the disk has changed.
Get BPB ------------ >	DOS needs the new BIOS Parameter Block for the new disk to determine where the Directory starts.
Read ------------ >	DOS requests the first Directory sector in order to find the volume label.
Media Check Get BPB ------------ >	DOS may make these requests several times depending on the amount of memory DOS has available to store information on the disk.
Read ------------ >	Read the Directory sector for the file Name and size information.
Media Check Get BPB ------------ >	Retrieve the current BPB if needed for calculating where the File Directory is.
Read ------------ >	Read the Directory sector for calculating number of files on the disk.
Media Check Get BPB ------------ >	Retrieve the current BPB if needed for determining where the FAT resides.
Read ------------ >	Read the FAT sector to calculate the amount of space available on the disk.

Table A-18: The typical called DOS makes to the disk device driver in order to process the DIR command on a newly formatted disk.

So far, the simple DIR command has DOS reading many sectors of the disk. What other calls can the disk device driver expect? Recall that DOS always checks to determine whether the disk has changed. This is reflected in the fact that each disk read requested of the disk device driver is preceded by a Media Check call.

Let's take the example above the DIR of a freshly formatted disk and expand the steps DOS has to take to arrive at the message "file not found". The typical calls that DOS makes to the disk device driver to perform this task and the responses it receives are shown in table A-18.

Note that, in table A-18, there are a lot of Media Check and Get BPB calls to ensure that the disk has not been changed. There are generally fewer of these calls for hard disks. This is because the disk device driver knows that the hard disk is non removable and can tell DOS the media has not changed. Therefore, DOS will not request the BPB except when the hard disk is initially accessed.

Now that we have covered the amount of work that a disk device driver has to do on request from DOS, we can review the commands that a device driver has to perform. This will help us understand what is expected of our Disk Device Driver.

Disk Device Driver Commands

As you have learned, when DOS requires a service from a device driver, the packet of data that is passed to the device driver with the call is referred to as the Request Header. Contained within this packet of data is a command number that corresponds to the service required by DOS. This command number instructs the device driver to perform a certain action.

There are 26 commands for device drivers in DOS version 5.0. We will now describe each of these commands and what is required to write code especially for disk device drivers. The list of applicable commands is shown in table A-19.

The Initialization Command

The Initialization command is the first command issued to the disk device driver after it has been loaded into memory. This call is issued because DOS needs several pieces of information from the device driver. The first is how many disk drive units this particular disk device driver will be supporting. For disks, this number is usually read through switches set on the PC motherboard.

The next piece of information that the device driver must return to DOS is the Break Address, which is the next available memory location after the driver. Because the driver knows its location, it can easily return this information. DOS then knows where to load the next device driver, if there is one; if not, DOS continues loading other routines.

The next item returned to DOS is the address of a table of BPBs. For 5 1/4- inch floppy disk units there are five types of disks : single -sided disks of 8 or 9 sectors per track, and special double-sided (high capacity) disks of 15 sectors per track. These five types of disks will have five different types of BPBs, varying in media descriptors, number of heads, FAT sectors, and File Directory entries. DOS needs to access this table of BPBs to determine the various sector sizes of each type of disk supported. The steps involved in finding the address of the BPB table are shown in figure A-7.

Number	Command Description
0	Initialization
1	Media Check
2	Get BPB
3	IOCTL Input
4	Input
5-7	Not Applicable
8	Output
9	Output With Verify
10-11	Not Applicable
12	IOCTL Output
13	Device Open
14	Device Close
15	Removable Media
16	Not Applicable
17-18	Undefined
19	Generic I/O Control
20-22	Undefined
23	Get Logical Device
24	Set Logical Device
25	IOCTL Query

Table A-19: All of the applicable commands for block device drivers.

The Media Check Command

The Media check Command in table A-19 is always called before disk reads and writes for other than file I/ O operations. When directory or FAT information is accessed, Media Check is called to determine whether the disk has changed. If so, DOS must read in new information on the disk.

DOS passes the media descriptor for the current disk in a particular disk drive, and the device driver can use this to determine if the disk has changed. Normally, as you saw earlier, this is not sufficient information because two similar types of disks (both single-sided, for example) will have the same media descriptor.

The device driver can return an indication of one of three possible conditions. The first condition is the media has not changed. This will be the case for nonremovable hard disks and RAM disks. The second condition is that the device driver has determined that the media has changed. The driver could determine this by checking to see if a disk door open signal has been received from the disk controller or by simply calculating the time since the last access of the drive. If the Media check is sent to the

driver within a very short time interval since the last access, it is not likely that a disk has been changed.

The last Media Check condition occurs when the device driver does not know if the media has changed. For example, if the time since the last access of the drive has exceeded a short predetermined time interval, the device driver assumes that a disk change could have occurred and returns a "don't know" condition.

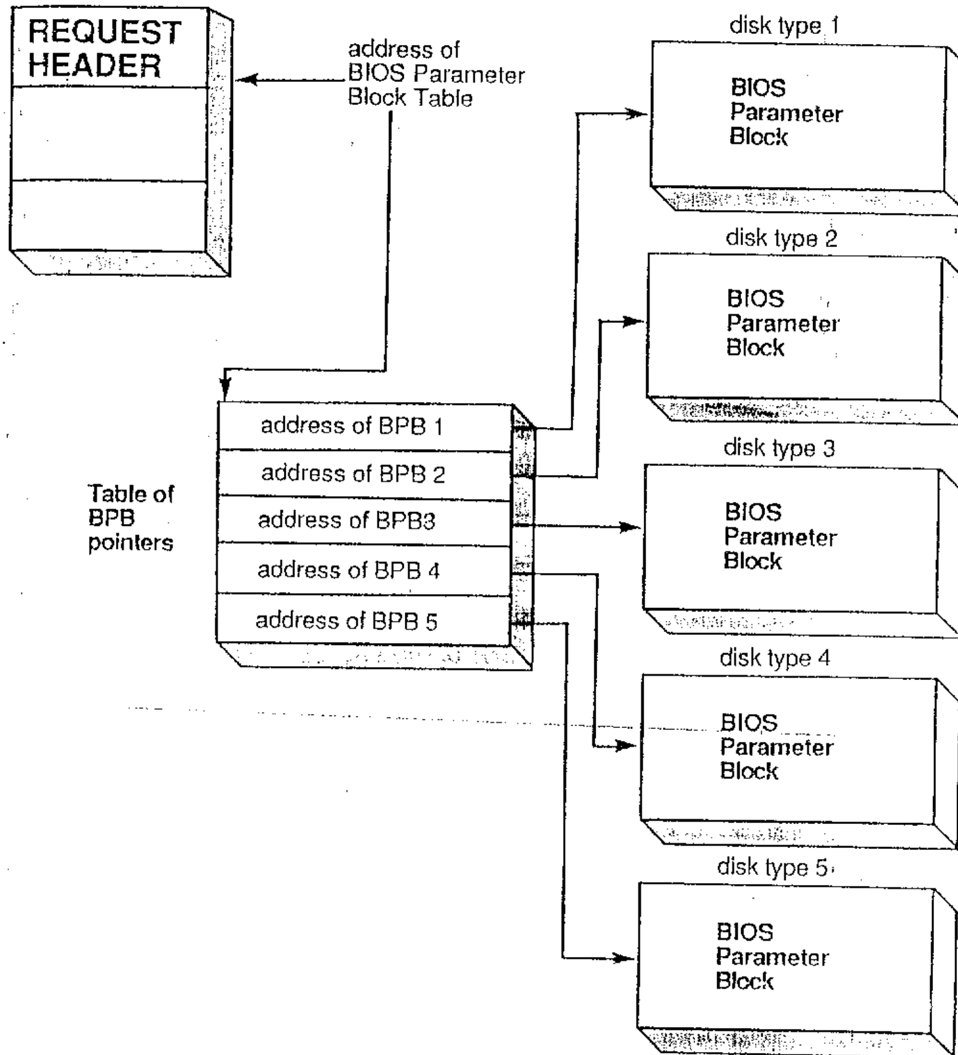


Figure A-7: The Initialization command requirement to return the address of the BIOS Parameter Block Table.

The GetBPB Command

The GetPBP Command is requested of the device driver whenever a media is changed condition is returned to DOS from a Media Check call. GetBPB is called for hard disks only once.

When the Media Check command returns a :don't know" condition, the GetBPB command is called only if DOS has no dirty buffers. Dirty buffers are those buffers that contains modified data for the disk that needs to be written. DOS assumes that if there are dirty buffers (modified data waiting to be written to disk), the disk has not changed.

If the device driver receives a GetBPB command, it will have to read the reserved or boot sector from the disk to access the BPB at offset 11 (decimal) of the boot sector. The BPB will end up in DOS's work area, and the device driver will return a pointer to this BPB to DOS. DOS can then use the BPB to calculate where the FATs and File Directory are on the disk.

The IOCTL Input Command

The IOCTL Input Command stands for I/O Control. This command is used by the device driver to return control information to the program regarding the device. For example if the device is a printer you can have the device driver return status information, such as the baud rate at which the printer device is set to receive data. When the driver returns I/O control information to the program, it is called input. Although this is quite useful, it is not a normal feature of the device drivers. There are many reasons for this. The first is that there is only one DOS call that allows I/O control - DOS service 44h. Most programs do not use this DOS service, because they do not expect a device driver to return this type of information. The second reason is that adding I/O control to a device driver is not easy; the device driver does not know what type of information to return. For I/O control to work properly, both the program issuing an IOCTL call and the device driver accepting IOCTL calls must agree on the information to be passed back and forth. For block devices, this does not have much meaning.

The Input Command

The Input command is sent to the device driver whenever DOS needs to read data from the disk. DOS will pass to the driver the number of sectors to read, the starting sector number, and the address of the data-transfer area in which the data is to be placed. DOS will have previously read in the FAT and File Directory and used these to calculate the needed sectors.

The starting sector number is numbered from 0 to the highest sector number for the disk and is relative to the start of the partition if it is a hard disk. For floppy disks, the start sector is always the reserved or the boot sector. It is up to the device driver to

translate this starting sector number into the appropriate track, head, and sector for the actual physical unit.

The Output Command

The Output command tells the device driver to write one or more sectors onto the disk. As it does for the Input command, DOS passes the starting sector number, the number of sectors to write, and the data-transfer address from which to write. The driver is responsible for translating this logical sector address to a physical disk address.

The Output With Verify Command

The Output With Verify Command is the same as the Output Command except that after the data is written out, the device driver is responsible for reading the data back in. This insures that the data has been properly written to the disk.

The VERIFY command in COMMAND.COM is used to set VERIFY ON or OFF. If it is set ON, all writes to the disk are passes as Output With Verify commands to the device driver.

The device driver can set a variable to indicate that VERIFY is ON. After writing to the disk, the driver can jump to the Input routine to read back in the previously written data and ensure that it is valid.

The IOCTL Output Command

The IOCTL Output command is similar to the IOCTL Input command, but the direction of data transferred is reserved. This command allows the program to pass an I/O control string to the device driver.

Again, the disk device driver can use this feature to implement just about anything. The I/O control string is not treated as normal data to be written out to the disk but is information that device drivers do not normally get. Without I/O control strings, it would be impossible to communicate with the device driver. The device driver would only get data to be written to the disk or read from the disk.

For instance, we could use I/O control strings to suspend disk operations temporarily and perform some maintenance diagnostics. However, this would involve a large amount of programming.

The Device Open Command

This disk driver command is new for DOS version 3.0 and is designed to signal the device driver that a file open for the disk has occurred. The device driver could keep counts of file opens to ensure that any reads and writes to the disk were preceded by file open commands. If not, we could be writing to the disk when there is no file opened. This would be the situation if a disk were removed before the file that was opened was properly closed.

In order to be able to receive Device Open and Device Close commands, the device driver must set the Open/Close/Removable bit in the Attribute word of the Device Header. Recall that the device Header is the table that occupies the first memory locations in the device driver.

The Device Close Command

The Device Close command is sent to the device driver whenever a program has closed the device. For disks, this happens when a file is closed on the disk.

The disk device driver, in conjunction with Device Open commands, could keep a counter of open files. When a Device Open command is sent, the driver would increment the counter. When a Device Close command is sent, the device driver would decrement this same counter. Then whenever a read (Input command) or a write (output command) is sent to the driver, we could check to see whether a file has been opened for the device. If not, we could disallow any I/O to the disk until files are properly opened or closed.

Unfortunately, this approach to enforcing proper disk usage is not very practical. Let's assume that a user has removed a disk before properly closing the file. The counter is set at 1, because the file is opened, so it will not disallow reads and writes to the disk. In other words, the problem has already occurred and there is no practical way of catching and remedying the situation.

The Removable Media Command

Removable Media is another command that is available for DOS version 3.0 or greater. This command is sent to the device driver only if the Open/Close/Removable bit is set in the Attribute word in the Device Header. With this command, a program could ask the device driver whether the media is removable. This could save time within a program, because if the media is not removable. This could save time within a program, because if the media is not removable, the program could assume that there would not be any disk changes. When the device driver is sent a Removable Media command, it will return an indication that the media is either removable or nonremovable.

Appendix B

Network Protocols & Communication

IEEE Standard 802 for Local Area Networks

IEEE produced several standards of LAN known as IEEE 802, include CSMA/CD, token bus, and Token Ring. They differ at the physical layer and MAC, and compatible in Data Link layer. These 802 standards are divided into parts:

802.1 : standard gives an introduction and defines the interface primitives.
802.2 : standard describes the upper data link layer.
802.3 ---> 802.5 : describes CSMA/CD, Token Bus, and Token Ring.

IEEE standard 802.3 and Ethernet

802.3 is for 1 persistant CSMA/CD LAN. When a station wants to transmit it listen to the cable, if the cable is busy the station waits until it goes idle, otherwise it transmits immediately, if two or more stations simultaneously begin transmitting on an idle cable, they will collide. Any station detecting a collision abort its transmission and send a signal burst to warn all other stations to terminates their transmission, wait a random time, and repeat the whole process again. It can use coaxial cable, or twisted pairs for transmission. All 802.3 use Manchester encoding, the presence of 1 bit in the middle enables the receiver to synchronize with the sender.

IEEE standard 802.4: Token Bus

for 802.3 a station due to a bad luck must wait a long period, and 802.3 frames have no priorities, so it is unsuited for the real time systems. A simple system is a Ring in which the station take turns in sending frames, it is a good idea, but if one station goes down the whole systems goes down. So they create a combination of Ring and Bus called Token Bus.
Physically, the Token Bus is linear or tree shaped cable into which the station knowing the address of its left and right stations. At initialization, the highest station send the first frame, and after it is done, it passes permission to its neighbor by sending a control frame called token, only the token holder being permitted to transmit frames, collision do not occur.

IEEE standard 802.5: Token Ring:

A ring is not really a broadcast medium, but a collection of individual point-to-point links that happen to form a circle, it can run on twisted pair, coaxial, and fiber optic. Each bit arriving at an interface is copied into a one bit buffer and then copied out onto the ring again, in the buffer the bit can be modified, and expected before being written out, this copy introduce a 1 bit delay in each interface.
In a Token Ring a bit pattern, called the token circulates around the ring whenever all stations are idle. When a station wants to transmit must seize the token and remove it from the ring. After the station transmits what it has, it must wait for the data to come back compare it, and remove it, and regenerates a new token and go back to listen mode. The problem of cable breaks in Token Ring is solved by center wiring.
We have different types of Ring:

-Slotted Rings: (is not part of 802) it is slotted into fixed size frames, each frame contains a bit that tell if it is full or empty, when a station wants to transmit, it simply waits for an empty frame to come around, marks it as full and puts its data in the frame.
-Register Insertion Rings: The interface contains two registers, a shift register, and output buffer, when a station has a frame to transmit, it loads it into the output buffer, frames maybe of variable length up to the size of the output buffer.

Comparison of LAN

A- 802.3:

Advantages:

- most widely used.
- the algorithm is simple.
- station can be installed in the fly without taking the network down.
- a passive cable is used and modems not required.
- delay is zero, station do not have to wait for a token, they just transmit immediately.

Disadvantages:

- each station has to be able to detect the signal of the weaked other station.
- collision.
- overhead because frame size at least 64 bytes.
- nondeterministic.
- it has no priorities.
- cable length is limited to 2.5km.
- CSMA/CD is inefficient at high speed.
- at high speed presence of collision becomes a major problem.
- 802.3 is not well suited to fiber optics due to the difficulty of installing taps.

B- 802.4:

Advantages:

- uses highly reliable cable television equipment.

- it is more deterministic than 802.3.
- it can handles short minimum frames.
- it supports priorities.
- it can be configured to provide a guaranteed fraction of the bandwidth to high priority traffic.
- it has excellent throughput and efficiency at high load.
- broaband cable can support multiple channels not only for data but also for voice and televisions.

Disadvantages:

- broaband systems use a lot of analog engineering.
- include modems and wideband amplifiers.
- the protocol is extremely complex and has substantial delay at low load.
- it is poorly suited for fiber optic implementation.

C- 802.5:

Advantages:

- priorities are possible.
- short frames are possible .
- large ones are possible also, limited only by the token holding time.
- the throughput and efficiency at high load are excellent. like the token bus and unlike 802.3.

Disadvantages:

- the presence of centralized monitor function, eventhough a dead one can be replaced, a sick one can cause headaches.
- there is some delay at low load because the sender must wait for the token.

D-ISDN PBXes versus LAN:

Advantages:

- PBX can be used to connect all the stations in a building.
- it can use external telephone wiring.
- it can also carry voice and data over the same network.
- it can connect station not only to local stations but also to those far away in a totally transparent way.
- the total throughput exceeds 500 Mbs.

Disadvantages:

- they do everything in a big way.
- the minuscule 64Kbps bandwidth on each ISDN channel.
- trying to page virtual memory to a remote disk over 64Kbps channel would be slow.
- trying to read a file at remote file server at 8000 bytes/sec would be slow.
- they are circuit switched, which is fine for continuous traffic but terrible at bursty traffic that computers generates.
- it is highly complex, centralized components.

Fiber Optic Networks

Fiber has a high bandwidth thin, and lightweight, it is not affected by electromagnetic interface, power surges, or lightning, and has excellent security. Fiber is important not only for WAN point to point, but also for metropolitan and LAN.

FDDI

It is a high performance fiber optic Token Ring LAN, 100Mbps over distances up to 200km with up to 1000 stations connected. It used the same way as in 802, another common use is a backbone to connect copper LAN, it uses multimode fibers, and LED rather than lasers. FDDI cabling consists of two fiber rings, one transmitting clockwise and the second transmitting counter clockwise, if either one is broken the second one can be used as backup. The basic FDDI protocols are closely modeled on the 802.5 protocol.

Fibernet II

Fiber LAN compatible with Ethernet at the transceiver interface so stations can be plugged into it using existing transceivers cable. Schmidt et Al used an active star instead of passive one, each transceiver has 2 point to point fibers running to the central star, one for input, and one for output.

S/NET

It is a Fiber Optic network with an active start for switching, each computer has 20Mbps, fiber running to the switch one for input and one for output, the fiber terminates in a Bib(bus interface board). When a word is written to that device register, the interface board in the CPU transmits the bits serially to the Bib, where they are reassembled in Bib memory, the CPU writes the command to another I/O device register to cause the switch to copy the frame to the memory of the destination Bib and interrupt the destination CPU.

Fastnet & Expressnet:

Fastent is a high performance network suitable for LAN, MAN, it uses 2 linear unidirectional buses, each station can send or receive on either one. When a station wants to send a frame to a higher numbered station it transmits on bus A, when it wants to transmit to a lower station it transmits on bus B.

Expressnet uses a single folded bus, each station attaches to the bus in 2 places, one on the outbound portion for transmission, and one for the inbound portion for reception.

DataKit

It is a single integrated network to be used as LAN, MAN, WAN, it allows copper and fiber to be intermixed in arbitrary ways. It contains switches, each with various kinds of line coming out of it, it is multiple interconnected stars rather than a bus or a ring, as various cards can be plugged into it.

Satellite Networks

Satellite have dozen of transponders, each has a beam that covers some portion of the earth below it. Stations within the beam areas can send frames to the satellite on the uplink frequency. The satellite shifts these frames to the downlink frequency and rebroadcast them. Transponders allocation is the major problem, carrier sensing can't work due to 270 msec propagation delay.

SPADE

used for telephone connection, we have two methods.

FDM: is the simplest; each transponder channel is divided into disjoint subchannels at different frequencies, which guard bands between subchannels to keep two adjacent channels from interfering. When a telephone call requiring a satellite channel is placed and deallocated when the call ends.

TDM: Satellite channel is not divided into subchannels by frequency, but by time. The channel is divided into slots; which are grouped into frames, for digital telephony a frame would be 125 micro sec, when a call is place, a free slots is allocated and dedicated to the call in every frame until the call is finished.

Packet Radio Networks

Broadcasting packet differ from satellite packet. Because stations have limited range, introducing the idea of radio repeaters, if 2 stations are faraway they can't hear each other's transmission. Thus making CSMA impossible; there is no common clock as in satellite for synchronization. All communication is either from a station to the computer center, or otherwise. They use 2 bands in the UHF, and they use 9600bps.

Appendix C

The OSI Layers

OSI Layers Description

The OSI model is shown in table C-1, and the role of each layer is described below:

7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

Table C-1: OSI Layers

Physical Layer

The Physical layer is the most fundamental layer in the OSI model. It is the interface to the physical medium and provides standards for electrical, mechanical, and functional transmission parameters.

Data Link Layer

The Data Link layer provides services to the network layer. It deals with procedures and services related to the transfer of data from node to node. Its main purpose is to ensure error-free delivery of data packets from point-to-point network viewpoint. Hence, it is concerned with such problems as error detection, error correction, and retransmission. However, since the data link layer is highly dependent upon the physical medium, there is not a universal protocol at this level. Individual protocols include HDLC for point-to-point and multipoint connections, as well as IEEE 802.2 with Media Access Control (MAC) and/or Logical Link Control (LLC) for Local Area Network (LAN).

Network Layer

The network layer ensures that a packet generated at the source arrives at its destination in a reasonable amount of time. Therefore, this layer handles routing procedures as well as flow control functions. It hides the physical implementation of the network from the upper layers that need not know whether fiber optics, twisted pair, or satellite communications are used. Hence, it creates a media-independent transmission from the point of view of the upper layers.

Transport Layer

The task of the Transport layer is to provide reliable, end-to-end data transfer for users of the transport layer. It establishes a connection between two endpoints, and it negotiates the parameters during connection establishment. It concerns also, with retransmission, multiplexing, time-out, and flow control.

Session Layer

The Session layer controls communication between applications. It provides services to the session above it such as, session establishment and session release. Another service that can be provided by the session layer is dialog management. This feature provides a half-duplex, flip-flop form of data exchange.

Presentation Layer

The presentation layer is concerned with the representation of the data that is being exchanged. This can include conversion of the data between different formats (ASCII, EBCDIC, binary), data compression , and encryption. Additionally, the presentation layer must make the services of the session layer available to the application.

Application Layer

The Application layer is the only one that does not provide any service to another layer, it deals with the user applications and handles communication at a semantics level. The application layer is concerned with problems such as interprocess communication, file transfer, virtual terminal and manipulation services, and job transfer and broadcast communication.

Appendix D

Glossary

BIOS : (Basic Input/Output System) a set of programs, usually in firmware, that enables each computer's central processing unit to communicate with printers, disks, keyboard, consoles, and other attached input and output devices.

Client : A workstation that boots with DOS and gains access to the network. With DOS client software, users can perform networking tasks, such as, mapping drives, capturing printer ports, etc.

DASD : (Direct Access Storage Device) The primary storage device e.g. disk drives.

DOS : (Disk Operating System) it contains information that ROM-BIOS uses to determine which device to boot from. The boot record can be on either a floppy diskette, a local hard disk, or a remote boot chip. ROM-BIOS Then runs a short program from the boot record to determine disk format and location of system files and directories. Using this information, ROM-BIOS loads the system files (including two hidden files, IBMBIO.COM and IBMDOS.COM) and the command processor (COMMAND.COM).

FAT : (File Allocation Table) The FAT is the mechanism DOS uses to manage \ DASD space. DOS allocates non-contiguous space for a file in a sequence of clusters, or allocation units (AU). Each AU in the FAT has the index of the next AU for the file. The Last AU contains 0xFFFF when DOS uses 16-bit cluster numbers.

Print Server : A computer that takes print jobs out of a print queue and sends them to a network printer.

Reentrancy : The ability for a second function call to successfully complete before a previous call to the same function has finished.

Server : A computer that regulates communications among personal computers attached to it and to manages shared resources such as printers., hard disks.

TCP/IP : (Transmission Control Protocol / Internet Protocol) an industry suite of networking protocols, enabling nodes in a heterogeneous environment to communicate with on another. TCP/IP is built upon four layers that roughly correspond to the seven layer OSI model. The TCP/IP layers are:

Process/application
Host to host
Internet
Network access

IPX : (Internetwork Packet eXchange) A Novell communication protocol that sends data packet to requested destinations (workstations, servers, etc.). IPX addresses and routes outgoing data packets across a network. It reads the assigned address of returning data and directs the data to the proper area within the workstation's or Netware Server's operating system. IPX is closely linked with other programs and routines that help in the network data-transmission process.

References

[1] Lai, S.R., Writing MS-DOS Device Drivers (Second Edition), Addisson Wesley 1992, pp 20-23-29-30.

[2] Adams, M. P., and Tondo, L.C., Writing DOS Device Drivers in C, Printice Hall 1990, pp 28-33-34-38-92.

[3] Williams, Al., DOS 5: A Developer's Guide, Advanced Programing Guide to DOS, 1989, pp 22.

[4] Schildt, H., Turbo C/C++ The Complete Reference, Osborne, McGraw-Hill, 1988.

[5] Halsall, F., Data Communications, Computer Networks & OSI, 1988.

[6] Stevens,W.R.,UNIX Network Programming, pp 6-15.

[7] Keringham, B. W. and Ritchie, D. M., The C programming Language, Prentice Hall, Englewood Cliffs, N.J., 1984.

[8] Postel, J. ed. 1981c, "Transmission Control Protocol," RFC. 793, 85 pages, Sept. 1981, pp 6.

[9] PC/TCP Development Kit, System Call Reference, Rel. june, 1991. Version 2.05 for DOS, FTP Software, Inc. Wakefield, MA.

[10] Novell Netware 4.0 Concepts, NetWare Network Computing Products, pp2.

[11] The Waite Group's Turbo Assembler Bible, Gary Syck, SAMS.

[12] Giles, B.W., and Giles M.M., Assembly Language Programming for the Intel 80XXX Family, International Editions 1991.

[13] Williams, Al., DOS 6: A Developper's Guide, Advanced Programming Guide to DOS, 1990.

[14] Comer, D.E., Internetworking with TCP/IP: Principles, Protocols, and Architecture, Prentice Hall, Englewood Cliffs, N.J., 1988, pp 3.

[15] Mogul, J. and Postel, J. 1985, "Internet Standard Subnetting, Procedure," RFC 950, 18 pages Aug. 1985, pp 7.

[16] Novell 1986, LAN Evaluation Report, Novell, Inc., Provo, Utah, 1986

[17] Postel, J. 1980, "Internet Protocol", RFC 791, 45 pages, Sept. 1981.

[18] Postel, J. ed. 1981c, "Transmission Control Protocol," RFC 793, 85 pages, Sept. 1981.

[19] Rose, M.T. 1990, The Open Book: A Practical Perspective on OSI, Prentice Hall, Englewood Cliffs, N.J.,1990, pp 8.

[20] Stallings, w. Mockapetris, P., McLeod, S., and Michel, T., Handbook of Computer Communications Standards, Volume 3:Department of Defense (DOD)Protocol Standards, Macmillar, New York, 1988, pp 5.

[21] Tanenbaum, A. S., Computer Networks, Second Edition, Prentice Hall, Englewood Cliffs, N.J., 1989, pp 5-6.

[22] Louis A. Delzompo, NFS and RPC, chap. 4, Sun Microsystems, Inc, pp 2.

[23] Croft, W. and Gilmore, j.1985, "Bootstrap Protocol (BOOTP)," RFC 951, 12 pages, Sept. 1985.

[24] Finlayson, R. 1984, "Bootstrap Loading using TFTP," RFC 906, 4 pages, june 1984.