3rd Edition

TANDY®

VOLUME 2

ADVANCED APPLICATIONS

By David A. Lien

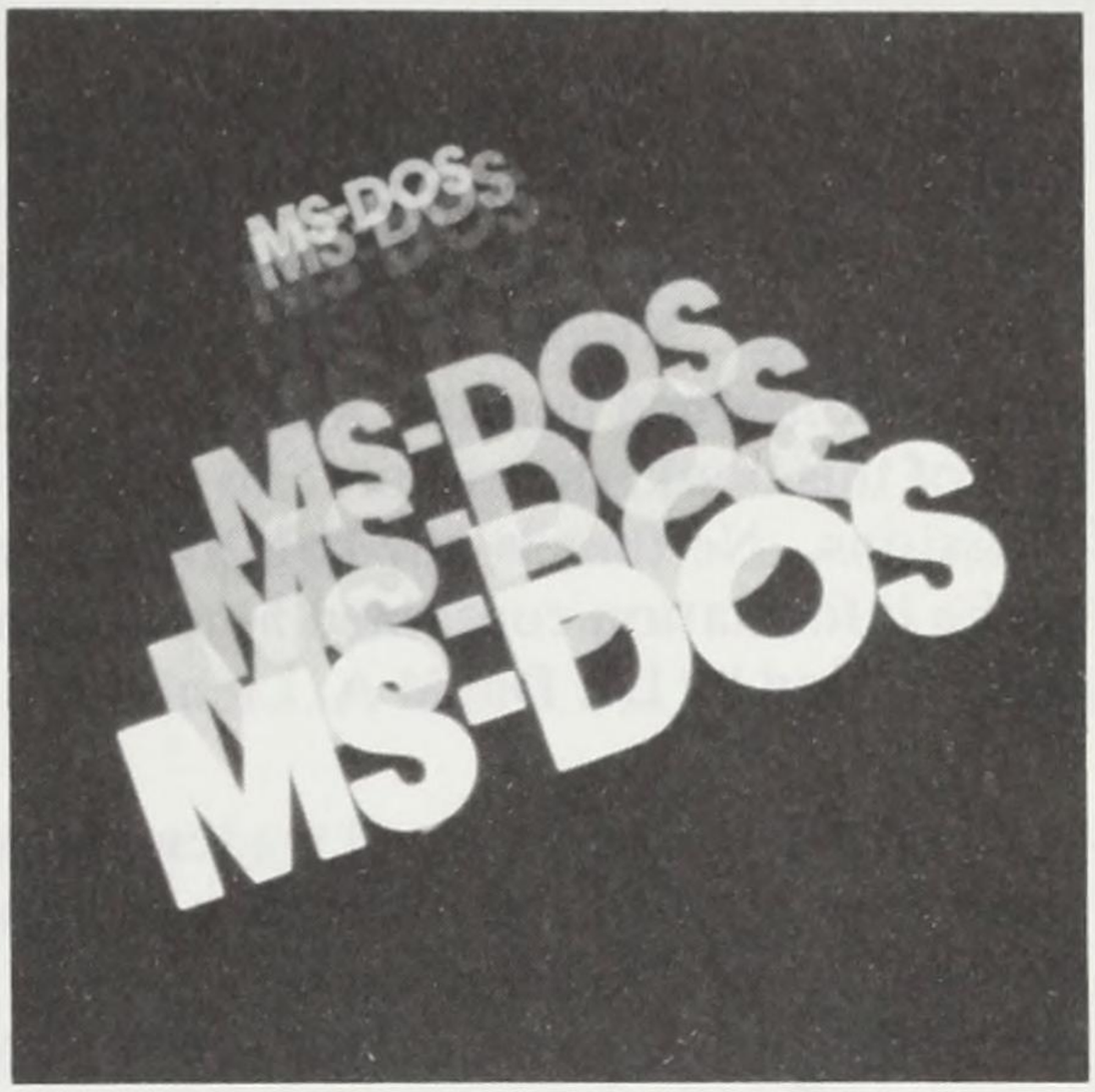
Cat. No. 25-1507

CREATING BATCH FILES

Copy con autoexec.bat
date ↵
tme ↵
echo off ↵
Path c:\;c:\dos;c:\desk;a:\;b:\;d:\
F6

endlin

3rd Edition



VOLUME 2

ADVANCED APPLICATIONS

By David A. Lien

COMPUSOFT® PUBLISHING

International Standard Book Number: #0-932760-46-5

Library of Congress Catalog Card Number: 88-70085

10 9 8 7 6 5 4 3 2

Printed in the United States of America, 1988.

Contents

Acknowledgments

Project Coordinator
Jaqueline K. Bohan

Production Coordinator
Janice Scanlan

Editors
Inez Goldberg
Joel Bailey

Cover and Book Design
Masar/Johnston Advertising and Design

Composition Design
Gary Williams

Illustrations
Martin Lindsay, Masar/Johnston Advertising and Design

Introduction

"Know how to use MS-DOS, and you'll know how to use your personal computer."

Oversimplified? Perhaps, — but look at it this way:

MS-DOS is the operating system that guides your modern Tandy computers. An *operating system* is a group of programs, loaded into memory every time you turn the computer on. It's the "brain" that controls how things are done.

You and MS-DOS interact. Request a directory and MS-DOS displays a list of the files on a disk. Tell MS-DOS to format a diskette, and it prepares the diskette so files can be saved on it. Give MS-DOS the name of a program you want to run, and it fetches that program from disk and runs it. Ask MS-DOS to copy a file, and it makes a duplicate. Direct MS-DOS to erase a file, and it does.

MS-DOS is so much a part of the computer's personality it's easy to forget it's there. But while you run programs to handle your own special tasks, MS-DOS is quietly at work behind the scenes. It decides how and where to store information on disk. It accepts entries from the keyboard and passes them along. It sends data to the video display screen. Did you forget to turn on the printer? Did you forget to insert a diskette? MS-DOS sends the message to alert you.

This book is Volume 2 of a two-part course on MS-DOS as implemented on the Tandy computer. Volume 1 taught the basics — those minimum things needed to perform routine day-to-day tasks. Volume 2 takes you on the grand tour, moving quickly through the basics, then on to the advanced features.

Guidelines and Conventions

In order to make this book easy to read and understand, we've adopted several conventions you should be aware of. Please note the following:

- MS-DOS is often abbreviated simply as "DOS." This book is about MS-DOS, and all references to DOS refer only to MS-DOS. While there are other operating systems in use in the world of microcomputers, "DOS" is recognized universally today as meaning "the DOS by Microsoft."

- An "off the shelf" Tandy 1000 TX with 640K of memory and a single 720K drive, running MS-DOS version 3.20 and Tandy version 3.20.21 is the "design center" machine.

- The disk drive which is in use or at the ready at any given time is known by a number of names. Since you need to be comfortable with all of them, we use the words *current, logged, default,* and *active* interchangeably.

- Reference is sometimes made to Tandy 1000s, 3000s and 4000s to indicate the entire series as opposed to any single computer model.

- We occasionally use the phrase the 3.x series or the 2.x series to refer to all the Tandy computers using version 3 or version 2 of MS-DOS. It is assumed that you have upgraded to the latest version of MS-DOS available for your computer.

- We will also, on occasion, use x's in place of numbers to represent the amount of bytes free, bytes in bad sectors, or the year in which a file was created. With the vast combinations of disk capacities available on Tandy computers, it would not be feasible to indicate how many bytes would be available on any given computer.

The theme of *MS-DOS, Advanced Applications* is managing your personal computer. To know MS-DOS is to know how to use your personal computer. Learn what's in this book, and you'll *really* know MS-DOS — and you'll be the boss.

Personal Note from the Author

This completely revised, expanded, and updated edition of *MS-DOS, Advanced Applications* reflects a major commitment to support the standard microcomputer disk operating system. It makes the power of MS-DOS available to everyone, without the need to first become a "tekkie."

This book covers every Tandy MS-DOS Computer.

There are 2 books in this series:

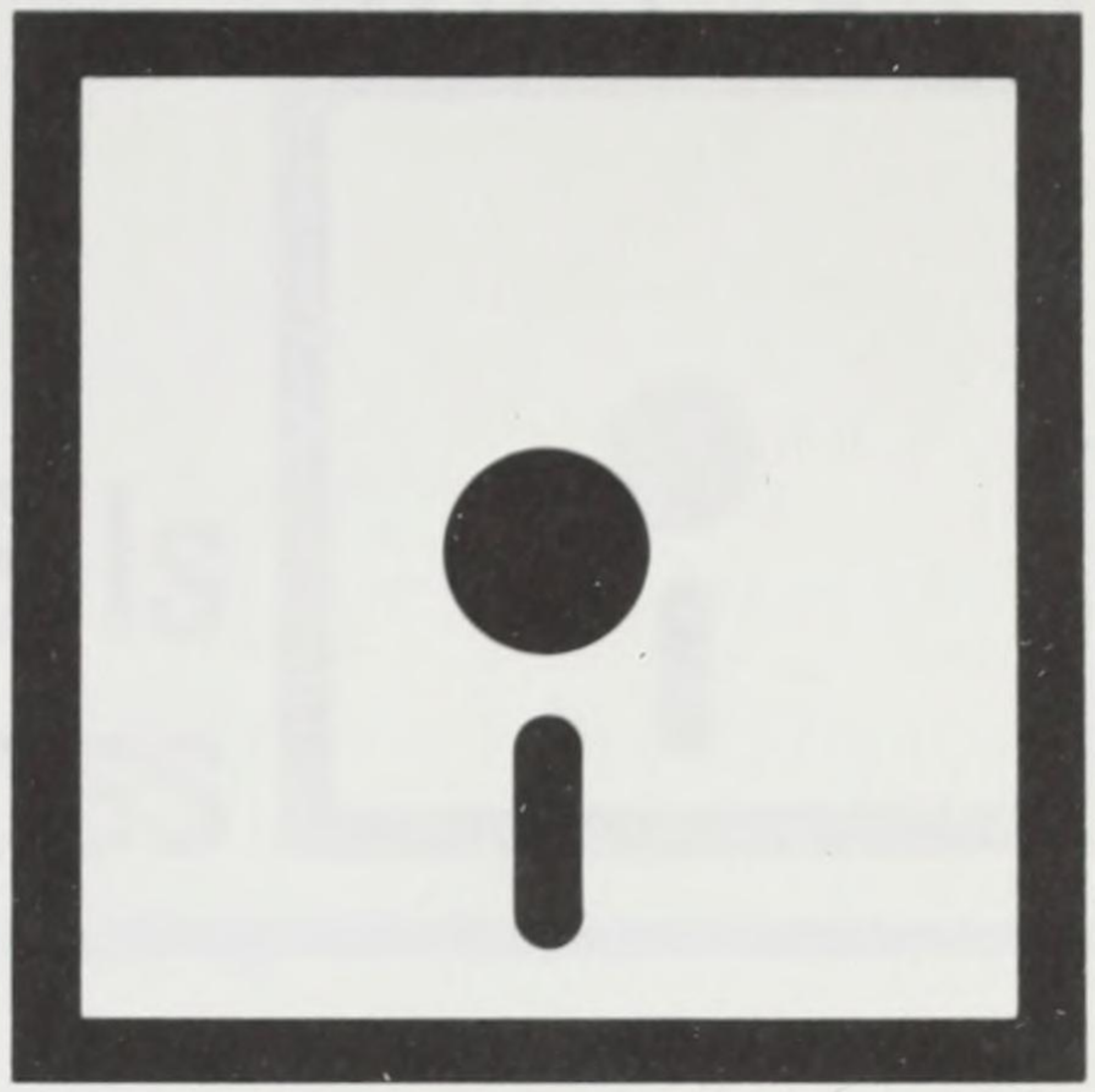
Volume 1 -- The Basics meets the introductory needs of secretaries and executives (the difference being that secretaries learn MS-DOS on the job while their bosses sneak it home in a brown paper bag).

Volume 2 -- Advanced Applications is for "power users," people who need advanced features, have hard disks, or are responsible for the office microcomputers.

Well over three hundred thousand readers have learned MS-DOS from these books. I'm confident that you too can master MS-DOS.

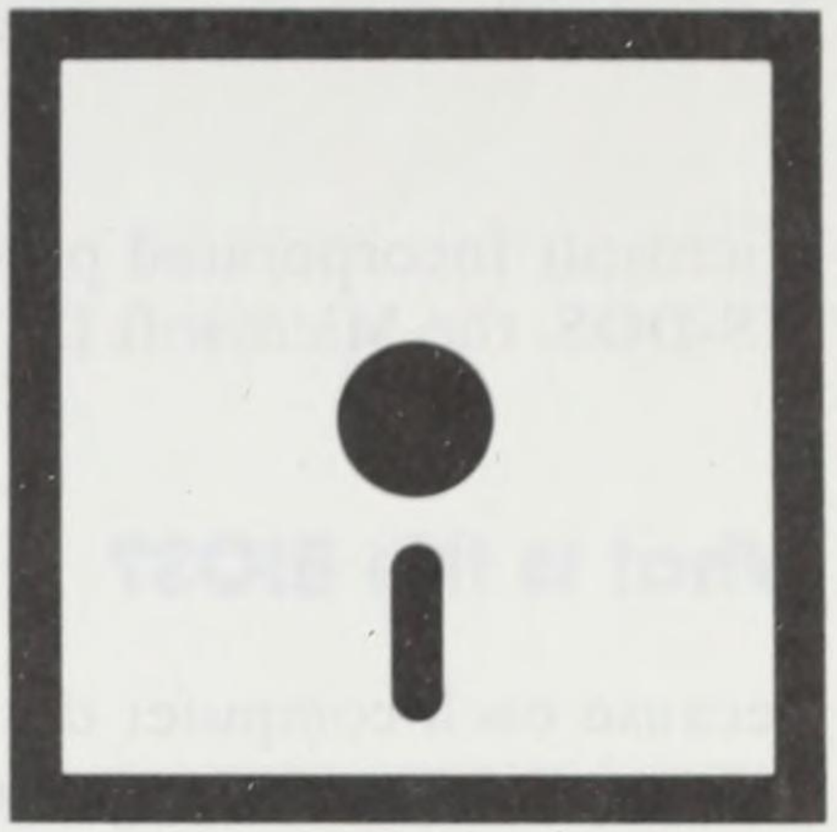
Dr. David A. Lien

PART 1



Intro to MS-DOS

CHAPTER 1



DOS Is the Boss

The Primal Scream

"Aaagh! This frustration is *too much!* I really *want* to learn MS-DOS, but I need a tutor!"

Welcome to the class. And *congratulations* on your ambition. You've decided to go beyond the basics and learn how to *really* take command of your computer. You've come to the right place. By the time you finish this tutorial, using MS-DOS will be a pleasure, not just another unfathomable mystery.

What Is DOS?

DOS (rhymes with Boss) is an acronym for **D**isk **O**perating **S**ystem. It is a cluster of programs that control the computer. DOS supervises the disk drives, organizes activity within the computer, gives permission for certain programs to do certain things, and equally important, DOS interacts with human operators.

DOS insulates you from the mysterious "ones and zeros" inside the computer and communicates on a more "personal" level. DOS can't reach around and scratch a favorite spot on your back, yet, but maybe someday...

Microsoft Incorporated produces and distributes this DOS, so it's called **MS-DOS**, the **M**icrosoft **D**isk **O**perating **S**ystem.

What Is the BIOS?

Because each computer design is slightly different, and MS-DOS is now the standard, something is needed to "fine tune" each computer to the DOS. That something is the BIOS. BIOS stands for **B**asic **I**nput **O**utput **S**ystem (no relation to the BASIC programming language). This is the chain of command: We talk to DOS, DOS talks to BIOS, and BIOS talks to the innermost guts of the computer.

The BIOS program is buried deep inside the computer, on unchangeable **R**ead **O**nly **M**emory chips. It is very carefully customized for each computer's hardware design.

Just remember that you give commands to MS-DOS, MS-DOS gives commands to the BIOS, and the BIOS handles the technical details of running the equipment. What it all boils down to is, "Learn MS-DOS once, and you can use *any* MS-DOS computer."

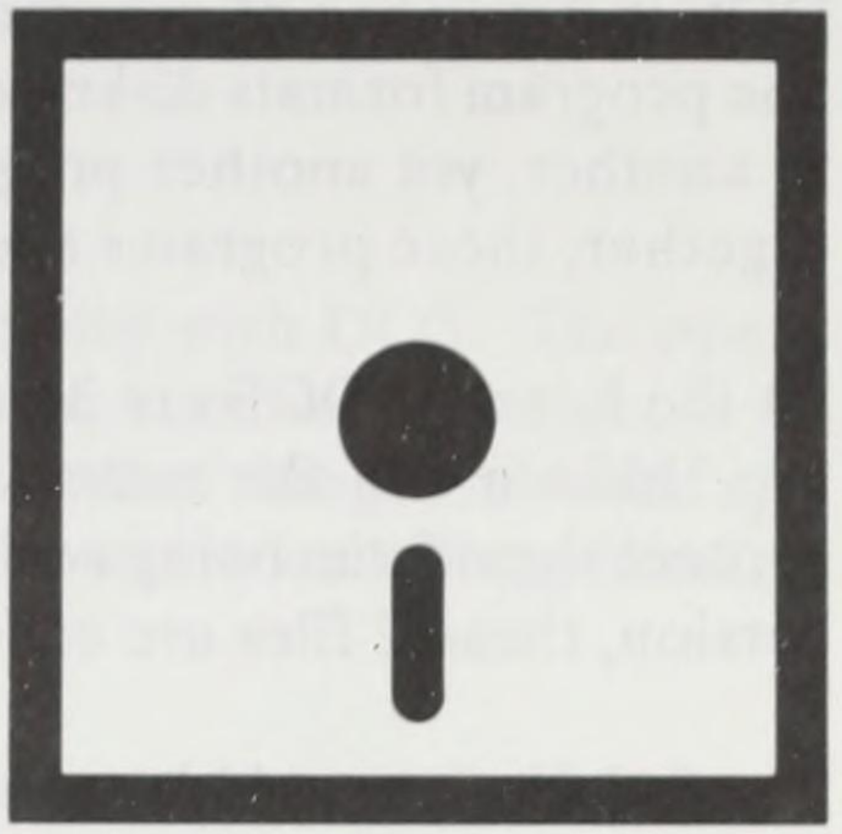
Chapter 1 Summary

MS-DOS, the Microsoft Disk Operating System, is a collection of programs that control the computer and its peripherals.

BIOS is a computer's Basic Input Output System. It "translates" standard DOS commands to communicate with a specific computer's design.

CHAPTER 2

How DOS Makes a Living



To the uninitiated, the computer is a magic machine. You ask it a question; it gives you an answer. Magic.

A lot goes on behind the scenes, however, and DOS doesn't just wake up smart. We have to tell it what to do, and speak in phrases it can understand.

My Kingdom for a File

One of DOS's primary purposes is to work with information stored on a *disk*, or *diskette* (we will usually use the words interchangeably). This information is stored in *files*, like file folders in a filing cabinet. DOS orchestrates the use of these files, placing them at specific spots on the disk, keeping track of which file is where, and remembering which file does what.

There are many types of files. Files can be programs, files can hold data for programs, or files can be part of DOS itself. Knowing the types of files and how DOS uses them is fundamental to understanding how the computer works.

The Whole Is Greater than the Sum of the Parts

DOS is a number of programs and files working together. For example, one program formats diskettes, another copies information from one place to another, yet another program checks computer memory, etc. Lumped together, these programs and others make up DOS.

At the heart of DOS are 3 programs. IBMBIO.COM and IBMDOS.COM are "hidden" on the *system* diskette (which we will soon use), mainly to protect them from being accidentally erased or tampered with. In DOS 2.x version, these 2 files are called IO.SYS and MSDOS.SYS.

The 3rd file is not hidden. Called COMMAND.COM, it is directly responsible for our personal interaction with DOS and is customized for each computer model. Its primary job is to read the commands we type on the keyboard and interpret them for MS-DOS.

But MS-DOS is much more. It includes different *types* of programs and files.

Internal commands are DOS commands and control words contained in the COMMAND.COM program. They include most of the fundamental commands: **dir** to examine a diskette's directory, **copy** to copy files and programs from one place to another, and so on. We will study them in detail.

External commands are separate programs. The FORMAT.COM program, which formats disks, is part of DOS, but it is a separate program.

Programming aids are included on the DOS disk for users who write their own programs. We'll discuss what programming aids are and how to use them near the end of the book.

Device drivers deal with computer control, organization, and customization. A device such as a disk drive or printer can be controlled directly by DOS through these device drivers.

Filters, as the name implies, filter the output of one file for use as input to another file.

Batch files are powerful files, almost programs, that we can write to control DOS. We'll concentrate on batch files and their many advantages in Part 4 of this book.

Each of these types of files is part of DOS. By learning to use the right combination of files, we can create our own "ideal system."

Application Programs

It's rare that someone buys a computer just to play with DOS. The operating system controls the machine and obeys our commands, but most of us bought a computer to do other things. These other things are called *applications*, and the software that performs them are called *application programs*.

Application programs are specialized programs that you purchase separately to do word processing, manipulate numbers in a spreadsheet, or keep track of your butterfly collection.

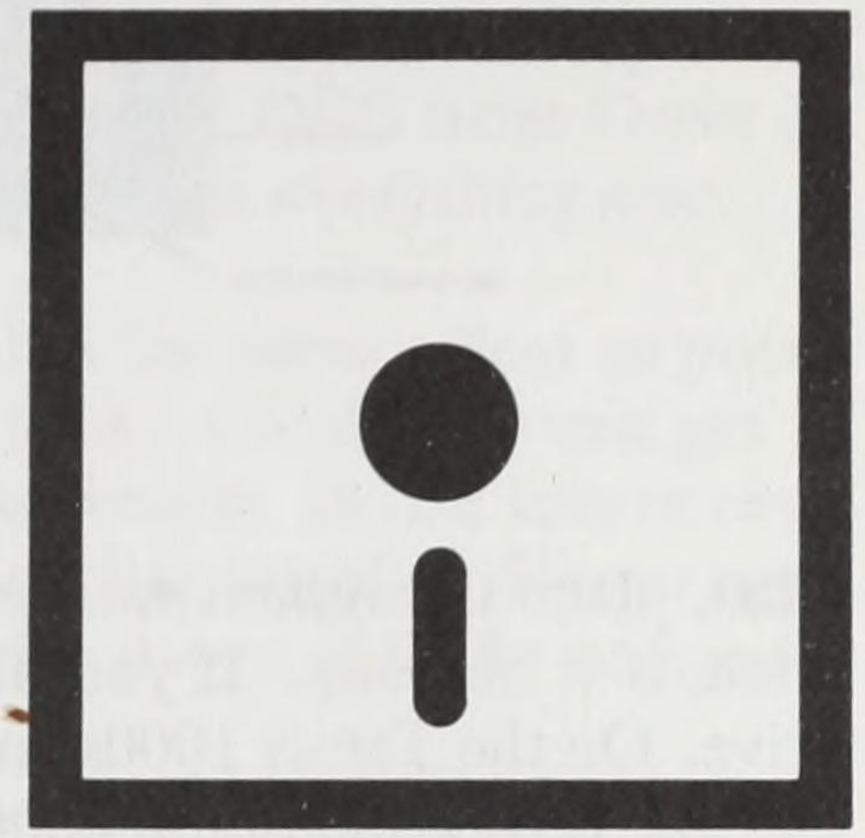
A large part of DOS's job is executing, or running, application programs. Once the application program takes over, it interacts with its own set of programs or files, and DOS becomes pretty much invisible to the user. When the application program is finished, control of the computer is returned to DOS.

Chapter 2 Summary

DOS manipulates data that is stored in files. It consists of dozens of files, but 3 files are at its core: IBMBIO.COM , IBMDOS.COM (IO.SYS and MSDOS.SYS in DOS 2.x), and COMMAND.COM.

Application programs are specialized programs created to meet specific user needs and are purchased separately.

CHAPTER 3



Getting Started

Up by the Bootstraps

To *boot* a computer means to start it up "from scratch." Booting up is to a computer what waking up is to humans. (And it may be equally traumatic, but we'll never know . . .)

Before turning on the computer, make sure the computer, monitor, printer, and hard drive (if it's external) are properly hooked up. One by one, turn everything on, the computer last. Or, if you have a power strip, plug everything into it and turn on the whole works at the same instant.

Find the diskette with the letters "MS-DOS" on the label. Depending on which computer you have, it might say "MS-DOS/BASIC" instead. This is your *system diskette.*

Because this system diskette is the original one from the factory, we'll call it the *master diskette*. To ensure that no harm comes to our *system master*, apply a *write-protect tab* over the slot or open the hole as shown in Figure 3-1. This prevents us from accidentally erasing the diskette by writing to it.

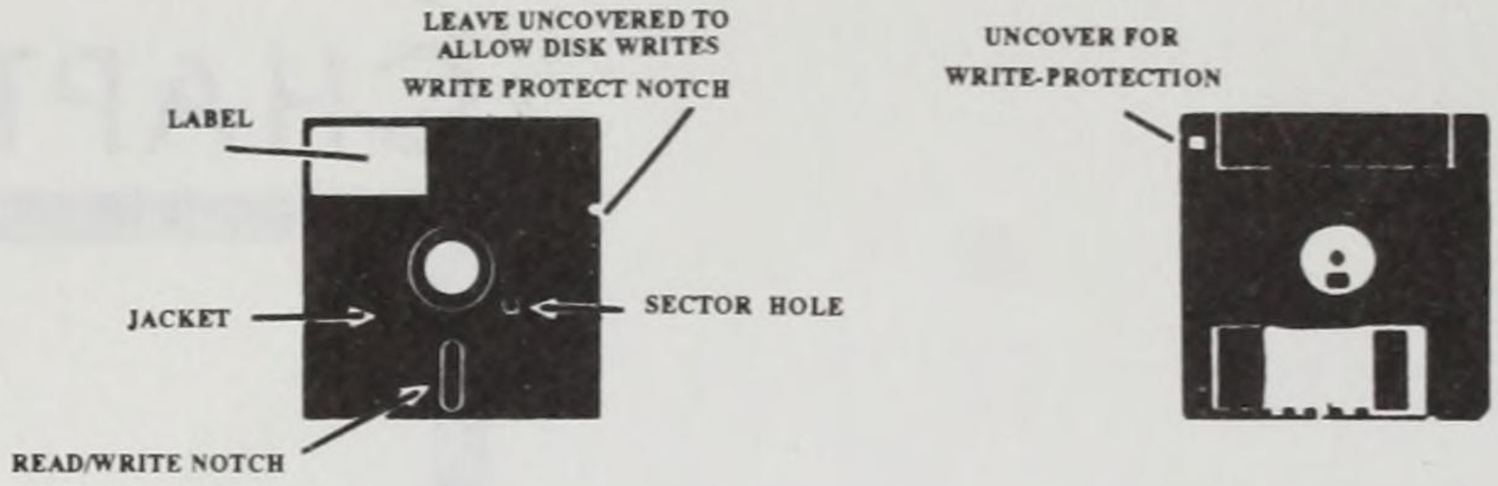


Figure 3-1

Next, place the *system master* diskette in Drive A and close the drive door latch, if it has one. If you have a one drive system, Drive A is your *only* drive. On the Tandy 1000s and 1400 LT, Drive A is the *bottom* or *left* drive. On the Tandy 3000s and 4000s, it's the *top* one.

Boot-up by pressing the CTRL , ALT and DELETE keys all at the same time. If your computer has a RESET button, you may boot by pressing the RESET button, instead.

Good Morning!

The computer does two things when booted. First, it performs a quick self-check, verifying the amount of memory installed and taking an inventory of its internal parts, printer, serial ports, etc. If something's wrong, an error message is displayed, or the computer simply beeps. If the computer does report an error at this point, jot the message down, and give it to the repair folks if you can't solve the problem.

The second thing the computer does is to check Drive A, the "main" disk drive, for the presence of a *system* disk. Unless a diskette with the required system files is in Drive A, the computer will not boot.

If your computer has a hard drive, it will look **there** for the system files if a system disk is not found in Drive A. For the present, hard drive users also boot from Drive A so you can follow the examples.

RESET! Computerese for "Panic"

When you pressed the RESET button, you told the computer to drop everything, erase its own memory, and load (or re-load) DOS from Drive A. RESET is the "panic button", and it always overrides everything else.

Pressing CTRL ALT DELETE simultaneously has the same effect as pressing the RESET button, but there is a draw back. Conditions can get so messy that the computer may ignore its own keyboard. In this severe case, the RESET button is the only escape short of pulling the plug. If your computer lacks a RESET button, open all disk drive doors, turn the computer off, wait fifteen seconds, and turn it back on.

Greetings

Let's look at the screen. What you see will depend on which computer you have. Since we assume throughout this book that most readers have a Tandy 1000 Series computer running DOS version 3.x, we will use its responses in most of our examples. Other Tandy computers display slightly different messages, but the DOS concept remains the same.

First comes a memory count. Our 1000 TX says:

```
Memory size =  640k
```

(meaning 640K of memory is installed.) At this point, the computer's BIOS is in control. It's taking inventory of the computer's memory. On some computers the BIOS displays a count of memory size from 64 right on up to the total amount of memory installed, followed by a beep, when finished.

Beep!

Drive A spins as the BIOS looks for DOS on the diskette. After DOS is loaded, the computer displays something like:

```
BIOS ROM version 01.03.00
Copyright (C) 19xx,19xx,19xx,19xx
Phoenix Software Associates Ltd.
and Tandy Corporation.
All rights reserved.

Microsoft MS-DOS version 3.20
(C) Copyright Microsoft Corp 19xx, 19xx
Tandy version 03.20.21
Licensed to Tandy Corp.
All rights reserved.

Current date is Tue  1-01-1980
Enter new date: (mm-dd-yy)
```

The release and copyright information varies among models. Looks good so far.

What Time Is It?

Time to take care of formalities. When we see:

```
Current date is Tue 1-01-1980
Enter new date:
```

DOS is asking us to enter the *correct* date.

Unless your system has a battery-powered clock installed, DOS assumes it's "day one," January 1, 1980, each time the computer is turned on or reset.

3000/4000 Series users, and others with an internal battery-powered clock, see the Installation and Operation Manual to learn now to set it.

DOS uses the date as a "stamp" to keep track of which files are oldest, etc. Some application programs need to know the date and can simply ask DOS rather than bother you for it.

Enter today's date — the month, day and year, as digits, separated by dashes, slashes or periods — and press ENTER. For example:

1/01/92
11-10-89
12.27.88

If you want to skip the date or accept the date from an internal clock, just press ENTER and DOS accepts the `Current date` shown.

After the date, DOS asks for the time:

```
Current time is 0:00:16.18
Enter new time:
```

Simply enter the hour and minutes, separated by a period or colon. DOS uses a 24-hour clock, so if it's past noon, add 12 to the present hour. For example, 4:30pm would be entered as **16:30**. The time can be set to a hundredth of a second, but few applications require such accuracy.

As with the date, you can skip all this by pressing ENTER.

Where Are We?

We've booted up, learned how to reset the system and informed MS-DOS of the date and the time. Now we see:

```
A>
```

This is the *system prompt* and it tells us that DOS is finally ready for action. We'll learn what to do with it in the next chapter.

Chapter 3 Summary

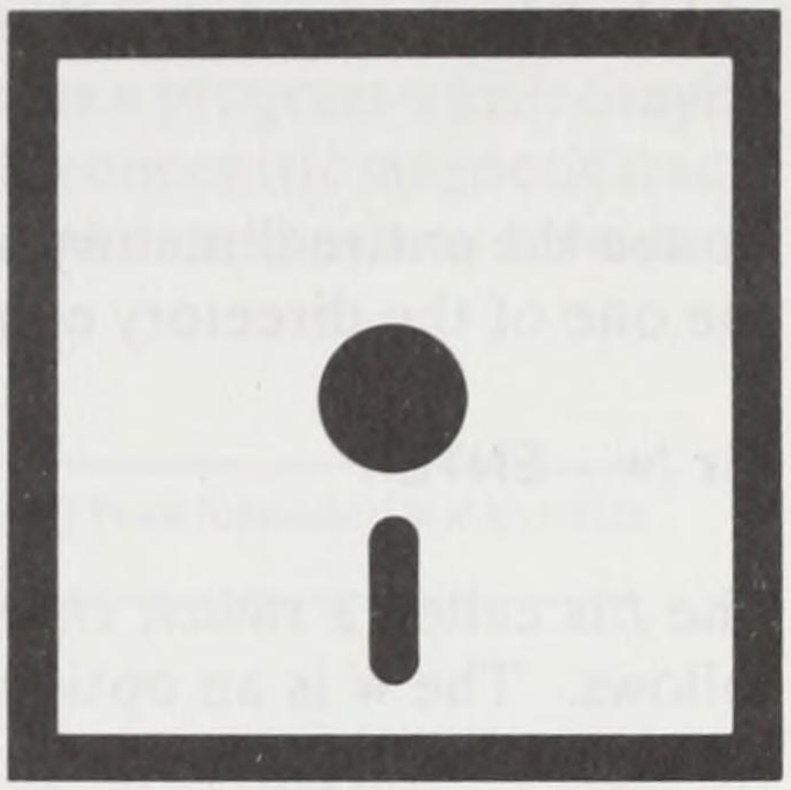
To boot up a computer means to power it up and load in the *system* files from disk.

Pressing the RESET button, or the CTRL ALT DELETE keys simultaneously, stops the computer, erases its memory (RAM), and reloads the *system* files.

Date and time can be entered following boot up or later.

When the *system prompt* appears on the screen, DOS is ready to accept commands.

CHAPTER 4



Initial Explorations

If you turned the computer off after the last chapter, turn it back on and be sure the system prompt and cursor are on the screen.

```
A>_
```

The system prompt is DOS's way of saying, "Ok boss, what do you want me to do?"

The letter A indicates that we are *logged on* to Drive A. That means DOS will read from and write to Drive A unless told otherwise. The *logged* drive is also called the "active," "default," or "current" drive.

Displaying a Directory

In this chapter we will make a copy of the DOS *system master* diskette — one for everyday use. Then we'll tuck the original away for safe keeping.

But first, let's see a listing of the files and programs which make up DOS. To display, or "pull," a directory of the DOS master diskette in Drive A, type:

dir ENTER

dir is a DOS command. It told DOS to display the directory of the files and programs on a particular diskette. Because we didn't specify from which drive to read the directory, DOS assumed we meant A, the default drive.

To see the entire directory of DOS files on the screen at the same time, we use one of the directory command options. Type:

dir /w ENTER

The / is called a *switch character* and means a special optional command follows. The **w** is an optional command, and it told **dir** to display a *wide* listing of the files.

The wide listing sacrifices some directory information in the interest of showing more in less space. **/w** only displays the file's name and its *extension*. (The extension is the three characters following the file's name, for example, COM, EXE, SYS.)

To see the *complete* directory again, but this time without having it scroll up and off the screen, use the **/p** option. Type:

dir /p ENTER

The **/p** stands for *pause* or for *page*. It lets us view the directory a page at a time while displaying full information about each file. Press ENTER to display each succeeding page.

How to Read a Directory

Without going into excessive detail, here's what a typical directory entry contains:

```
COMMAND.COM    23612    7-16-xx    3:00p
```

COMMAND is the *name* of the file or program, and COM is its file name *extension*. Next is the *size* of the file, measured in bytes. Finally, the *date* and the *time* the file was created, or last changed.

The **dir** command has other interesting options which we will learn in future chapters.

A Format to Stick To

A new diskette right out of the box is useless. It must be formatted to specifications DOS can work with. DOS contains a program which can *format* the diskette by recording a series of invisible concentric magnetic tracks on it. This process erases any previous tracks and data that may have been there.

CAUTION: DOS will not inform you if a diskette has already been formatted and has data on it.

Look for the FORMAT.COM program on your DOS diskette. (Type **dir** or **dir/p** again if necessary.)

```
FORMAT.COM    11122    7-16-xx    3:00p
```

Because it is a separate program, FORMAT.COM is called an *external* DOS command. *Internal* commands, such as **dir**, are actually loaded into computer memory at the same time that DOS is loaded, and they remain there even if the system diskette is removed.

To format a diskette, at the `A>` prompt, type:

format a: ENTER

Have a blank diskette ready for the next step.

DOS loads the FORMAT.COM program into memory and displays:

```
Insert new diskette for drive A:
and strike ENTER when ready
```

Remove the DOS *system master* from Drive A and replace it with a fresh, new diskette. Press ENTER.

On some Tandy 1000s, a row of dashes changes to a row of dots as each diskette track is formatted. 3.x Series computers display the head and cylinder number as each track is formatted.

The `Format complete` message is followed by a report of the diskette's status:

```
xxxxxx bytes total disk space
xxxxxx bytes available on disk
```

When formatting is complete, the program asks if we want to:

```
Format another (Y/N)?
```

Since you'll need additional formatted diskettes for use with this book, format 2 more and set them aside.

Bad Times

The FORMAT.COM program will occasionally report sectors (small segments of a diskette track) which contain "bad bytes." It is rare with good quality floppies, but you might eventually see something like:

```
xxxxxx bytes total disk space
xxxx bytes in bad sectors
xxxxxx bytes available on disk
```

This usually indicates a faulty diskette, but might be a dirty disk drive. Try formatting the diskette again. If bad sectors show up a second time, discard that diskette and try another.

Multiple Drives

We can format a disk in any drive by specifying that drive's letter designation. To format a diskette in Drive B (if you have one), type:

format b: ENTER

and follow the directions on the screen.

Using DISKCOPY

The diskette we just formatted is ready for use. We can use it for storing data, copying programs, or as we will now do, making a duplicate of any other diskette of the same size. Because the *system master* is critical to operation of the computer (and we only have one master), we'll make a copy of it for everyday use.

DISKCOPY.COM makes a "mirror image" duplicate of a disk by copying the information, track-by-track, from the *source* diskette to a *target* diskette only if both drives are identical (5-1/4" to 5-1/4" and 3-1/2" to 3-1/2"). Put the DOS system master back into Drive A. (If you have a 2-drive system with identical disk drives, place one of the newly formatted disks in Drive B.)

For single drive systems, or systems whose drives are *not* identical, type:

diskcopy a: a: ENTER

or just **diskcopy** ENTER

With 1-drive systems, this will mean swapping the *source* and *target* diskettes back and forth when instructed, so both diskettes can use the same drive. The drive both reads the original disk and writes to the copy.

On dual-floppy-drive systems with identical drives, type:

diskcopy a: b: ENTER

When **diskcopy** asks if we want to copy another disk, answer N, and the A> prompt returns.

With the *system* diskette (the duplicate, not the original) in Drive A, study the directory to make sure **diskcopy** did its job. Type:

dir /w ENTER

Does this duplicate look the same as the original? Good.

Remove the duplicate system disk. Use a felt tipped pen to write "MS-DOS SYSTEM DISK" on the label. Write "DATADISK" on the remaining, formatted diskettes, numbering them 2 and 3. (You'll see why later.)

Comparing Two Diskettes

The only certain way to be sure a diskcopy worked is to compare the two disks in question, a byte at a time.

Put the original MS-DOS disk back in Drive A and type:

diskcomp a: ENTER

You will be prompted when it's time to swap disks.

> **Two-Drive Systems:** Type **diskcomp a: b:** ENTER, and follow the instructions displayed on the screen.

DOS asks you to insert the disks you want compared.

When the comparison is finished, if all is well, DOS reports:

```
Diskettes compare ok
Compare more diskettes (Y/N)?
```

Answer with an N.

Now you know you have an exact duplicate of the MS-DOS *system master* diskette— a safety copy. *Hide the original away!* Almost all our practice exercises will be done with the duplicate diskette, hereinafter known as the *DOS* or *system* disk.

CHKDSK

Put your new *system master* back into Drive A, and look for the "command" program called CHKDSK.COM. Hard to spot? Type:

dir chkdsk.com ENTER

Following **dir** with a specific file name requests directory information for only that file. If it's present, a message like this appears:

```
CHKDSK   COM   9819   7-16-xx   3:00p
    1 File(s)      xxxxx bytes free
```

This CHKDSK.COM file occupies 9819 bytes of space. You also see when it was created and how much free space is available on the diskette.

Since CHKDSK has the COM extension, it can be executed by DOS. To run a COM program, just type the first part of the file name and press ENTER. Since file names are limited to 8 characters (plus a 3-character extension), it is common to abbreviate. CHKDSK is short for CHecK DiSK. Besides reporting information, it does a quick "checkup" on the disk in a drive. Request it by typing:

chkdsk ENTER

On my 640K Tandy 1000 TX, it says:

```
730112 bytes total disk space
44032 bytes in 2 hidden files
588800 bytes in 67 user files
97280 bytes available on disk

638976 bytes total memory
590048 bytes free
```

The numbers you get may differ depending on which version of DOS you have, how much memory is in the system, and what files are on the diskette.

Total disk space is the capacity the disk had before anything was copied onto it.

The bytes in the *2 hidden files* are the DOS system files.

User files are the ones we're usually concerned with. They are the ones that appear in a directory display.

Bytes available on disk tells how much space is unused, thus available.

Finally, CHKDSK.COM shows how much memory the *computer* has available for loading and running programs. This has nothing to do with checking a disk, it's just a bonus statistic. The difference between "bytes free"

and "total memory" is the overhead used to hold DOS, and other operating system information.

> Besides giving the "stats" for a diskette, CHKDSK.COM verifies that the directory is properly organized. Sometimes, after much disk use, CHKDSK.COM reports a problem. For now, use the old "one more try" approach if errors are reported.

Chapter 4 Summary

The *system prompt* indicates the *default* drive, the one DOS will read from or write to, unless otherwise specified.

A disk's *directory* is a list of its files and programs.

dir displays a disk's directory. Adding the /**w** option displays it in *wide* form. Adding /**p** displays the directory a *page* at a time.

format a: formats the disk in Drive A.

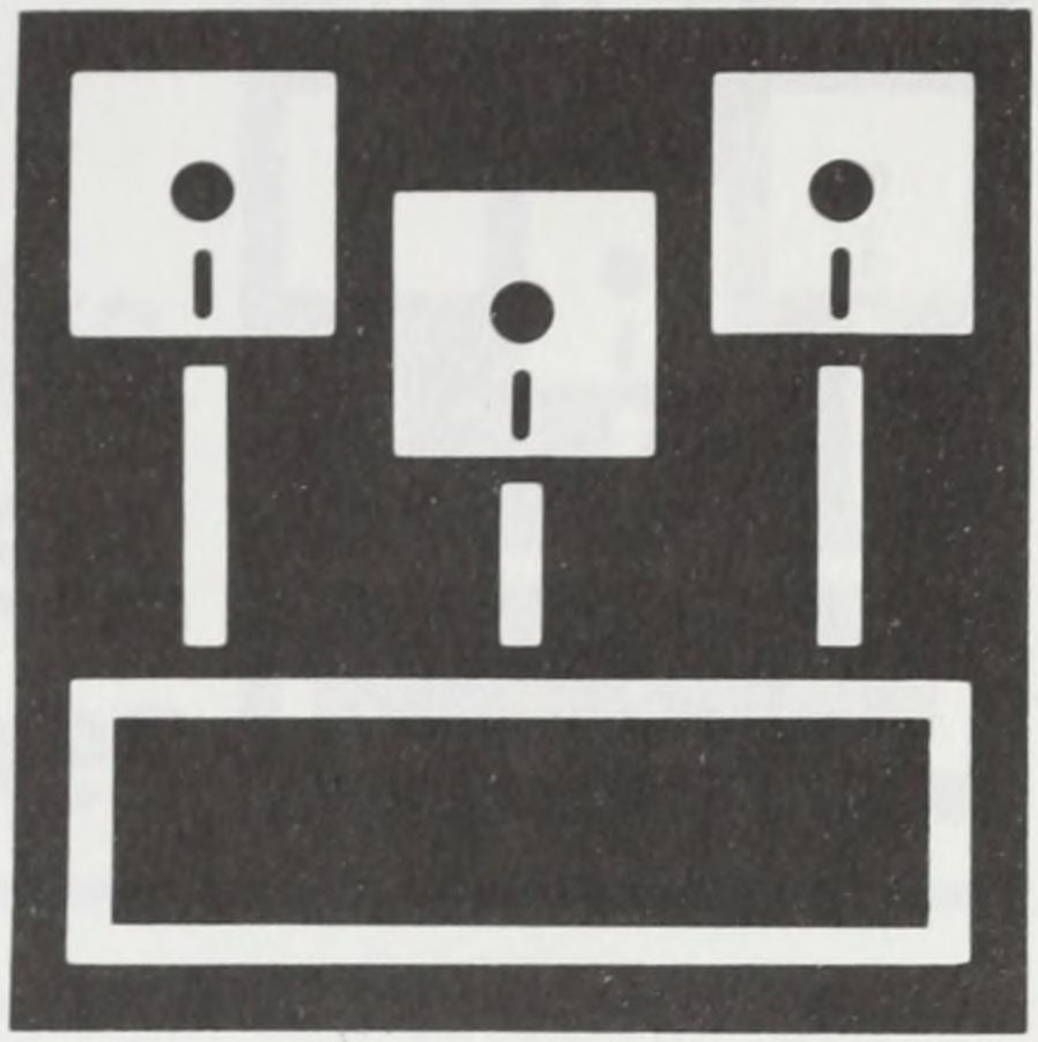
diskcopy or **diskcopy a: a:** makes a copy of the disk in Drive A. (You will have to swap diskettes.) On 2-drive systems, use **diskcopy a: b:** to copy the disk in Drive A to an identical disk in Drive B. If the target diskette is not already formatted, **diskcopy** will format it prior to copying.

dir somefile displays complete directory information for a file named SOMEFILE.

Type **diskcomp a:** to compare two diskettes (**diskcomp a: :b** for two-drive systems).

chkdsk checks a diskette and displays its capacity and usage statistics.

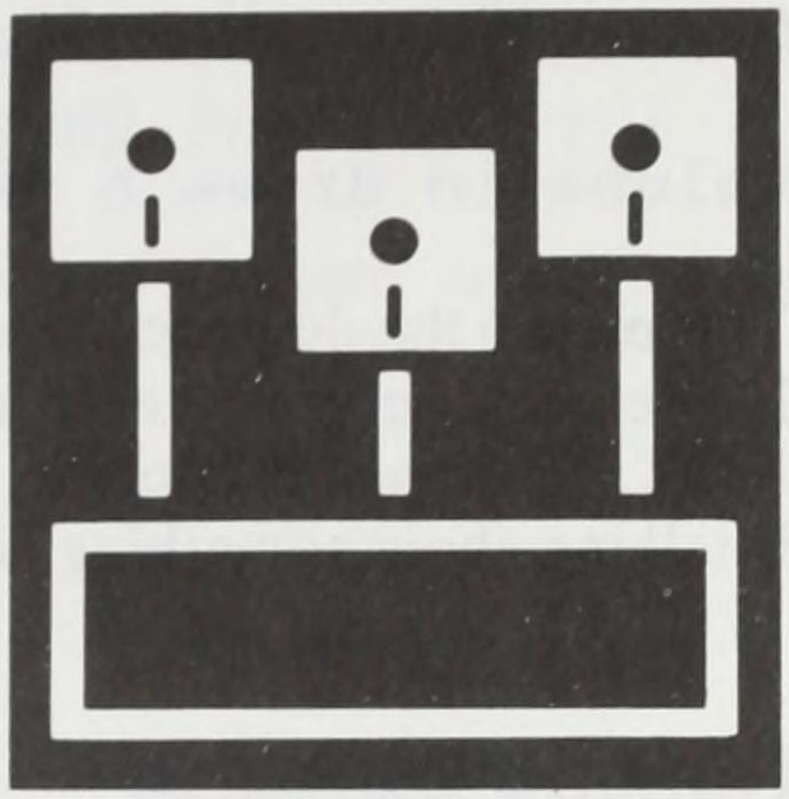
PART 2



Working with files

CHAPTER 5

Disks, Volumes, and Files



Collections of data and programs are called *files*. The keyboard, video display, disk drives, and printer are *devices*. DOS controls *files* and *devices*.

Names to Remember

A device is anything controlled by the computer: disk drive, printer, monitor, keyboard, serial port, etc. DOS has its own names for them:

First floppy drive	A:
Second floppy drive (if present)	B:
First hard drive (if present)	C:
Second hard drive (if present)	D:
Keyboard and display	CON:
First printer port	LPT1: or PRN:
Second printer port (if present)	LPT2:
Serial (RS-232) port	COM1: or AUX:

DOS considers the keyboard and display to be a single device, known as "CON" for *console*. They can be separated as KYBD: and SCRN: for keyboard and screen, but this is not commonly done. It is customary to end device names with a colon. DOS *requires* the colon when designating disk drives, but its use is optional with CON and PRN.

Though each disk drive is referred to as a device (A, B, and C), the disk inside the disk drive is referred to as a *volume*. Do you recall this message from our directory listing?

```
Volume in drive A has no label
```

This means the diskette in Drive A has not been assigned a name, or label. It does not mean we haven't put a sticker on the diskette saying "DATA DISK," or some other name. Those types of labels humans can read. DOS is talking about something else.

Naming Diskettes

The **format** command can place a *volume label* on the diskette at the time it is formatted. Insert a new unformatted diskette in Drive A and type:

format a:/v ENTER (**format /v** ENTER on the 2.x Series)

With a 2-drive system, use:

format b:/v

/v is a **format** command option which allows adding a volume label to the disk. *After* the formatting is complete, DOS asks:

```
Volume label (11 characters, ENTER for none)?
```

Your label may be up to 11 characters long, including spaces, numbers, and some symbols. Type:

DATADISK#1 ENTER

and do *not* format another at this time. Type **dir** to see the new label.

DOS version 3.x includes an external command, **label**, which can change or remove, as well as assign, a volume label. If you have a 2-drive system, insert the *system* diskette in Drive B and type:

b:label a: ENTER

DOS comes back with:

```
Volume in drive A is DATADISK #1
Volume label (11 characters, ENTER for none)?
```

To erase the volume label from the just-named diskette, simply press ENTER.

Use the **dir** command to verify that the diskette no longer has a label.

To put a label back on the diskette, type:

b:label ENTER or **b:label a:** ENTER

and when the screen shows:

```
Volume in drive A has no label
Volume label (11 characters, ENTER for none)?
```

type in this label:

DATADISK#1

Use the **dir** command again to verify that the diskette once again has a volume label.

Once a label is "attached" by DOS 2.x, there's no easy way to remove or change it. Several companies sell special software programs which allow adding or changing the volume name.

In any case, remove the diskette, and using a felt-tip pen write "DATADISK #1" on the label. Put it with your other formatted diskettes.

The 3-1/2" Diskette

The 3-1/2" diskette is now standard in many Tandy MS-DOS computers. This diskette is more compact and sturdier than the 5-1/4" diskettes and holds 720K bytes. The high-density 3-1/2" version holds 1.44 megabytes.

3-1/2" diskettes are formatted using the same **format** command as for any other DOS disk. Internally, the computer hardware reports the size of the diskette to DOS, which passes the information on to the FORMAT.COM program.

If Drive B is the 3-1/2" drive, simply type:

format b:

Different Diskette Formats

DOS 3.x has special diskette formatting options. On the Tandy 3000 HD, a high capacity drive capable of reading and writing over 1.2 million bytes is standard for Drive A. Drive B can also be 1.2M (like A), or you may elect to install a 360K drive similar to those used on the 1000s.

The 360K diskette is still accepted as the "standard" throughout MS-DOS land.

Unless told otherwise, **format** will format a diskette for whichever type of drive is installed. On the 3000 HD, a diskette in Drive A is formatted to hold 1.2M unless a special DOS 3.x **format** command option is used. When using:

format a:/4 ENTER

the **/4** tells **format** to create a standard 360K diskette on a 1.2M drive. 3000 HD users insert a 5-1/4" blank floppy disk into Drive A and press ENTER.

The resulting 360K diskette can be read and written to by the 3000 HD in its 1.2M drive, but has the added advantage of being readable by all other Tandy MS-DOS computers with 5-1/4"drives.

NOTE: An industry wide problem exists at this time with some 1.2M drives used to format 360K diskettes. On some computers, the 360K diskettes formatted on 1.2M drives are not compatible with other 360K drives. We have extensively researched this problem at CompuSoft and have found no compatibility problems among Tandy computers. If you need to interchange diskettes with other computer brands, we suggest you purchase an optional 360K disk drive and have it installed as Drive B on your 3000 HD or 4000.

The high density 3-1/2" drive in the Tandy 4000 can format high density 3-1/2" diskettes with the **format a:** command. It can also format standard 3-1/2" diskettes with the special command:

format a: /N:9 /T:80

where the standard diskette is in the high density Drive A. It is very important to use the proper formatting command.

Which Is Which?

This chart shows which diskettes can be read by which "off the shelf" Tandy computers.

	If formatted on: 1000 SX (360K)	1000 HX,TX 1400LT (720K)	3000 HL (360K)	3000 HD (1.2M)	4000 (720K)	4000 (1.44M)
Can be read by:						
1000 SX	X		X			
1000 HX, TX, 1400 LT		X			X	
3000 HL	X		X			
3000 HD	X		X	X		
4000		X			X	X

What Is a System Disk?

When formatting a diskette, you need to decide whether or not it needs to be a *system* disk. In other words, do you want to be able to insert that disk

in Drive A and boot the computer with it? Or are you willing to use a different disk to "boot up," inserting this one later?

If you have a hard drive, you will almost always boot directly from it. There is rarely need for a *system* diskette because the hard drive holds all the required files. You'll probably just use the floppy drive for making safety backups and for transferring files to and from other computers.

With a floppy-only system, the decision depends on what you intend to store. If a diskette is to hold only *data* files (and no programs), there's no need to make it a *system* disk. Or if you'll always be using it in Drive B, it needn't be a *system* disk because the one in Drive A will contain the DOS system files.

We referred to our original DOS *system master* and its copy(s) as *system* diskettes. However, most of the files and programs which are on these diskettes are *not* needed to run DOS. In fact, a bare bones *system* diskette needs only 3 files:

3.x Series	**2.x Series**
IBMBIO.COM	IO.SYS
IBMDOS.COM	MSDOS.SYS
COMMAND.COM	COMMAND.COM

The 2 hidden files, IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS), are always the *first* 2 files on any system diskette. The 3rd file, COMMAND.COM may be stored anywhere on the diskette.

To make a bare bones *system* diskette, from a *system* diskette, type:

format a:/s ENTER

The /**s** option formats a new diskette and automatically copies all 3 *system* files to it. After formatting, you see the message:

```
System transferred
```

```
xxxxxx bytes total disk space
xxxxx bytes used by system
xxxxxx bytes available on disk
```

To verify that this is a real *system* diskette, reboot, with it in Drive A, by pressing CTRL ALT DELETE.

Sure enough, there's DOS!

The /v option can also be included when formatting a *system* diskette, as in:

format a:/s/v

After the system is transferred, DOS asks for a volume label, as before.

This system diskette is all that's needed to run DOS, as long as you don't need to do special things like formatting. Unlike the complete *system master* diskette, the 3 mandatory files occupy very little space on the diskette.

When you're not sure if you want to include the system files on a particular diskette at this time, another **format** command option is available. Typing:

format a:/b

tells **format** to reserve space on the diskette for the system files in case they are to be added later. It simply puts "place holders" in the first two directory spots on that disk.

Why COMMAND.COM?

COMMAND.COM does many things. It is in control when you see the system prompt. It checks to see that your commands, such as **dir**, **copy**, and **delete** are valid.

Without it, the computer displays `Bad or missing Command Interpreter` when you attempt to boot and the keyboard "locks up." COMMAND.COM is the "steering wheel" for a computer running under DOS.

In a directory request, COMMAND.COM interprets the **dir** command and passes it to DOS. Then DOS tells COMMAND.COM what files are present. Finally, COMMAND.COM does the "cosmetic" part of the job—displaying the directory on the screen in a neat, organized way.

If you enter a command it doesn't recognize, COMMAND.COM tells DOS to check the disk for a program matching your request. If the program is found, COMMAND.COM tells DOS to copy it into memory. If not, COMMAND.COM provides the `Bad Command or file name` message.

When Must You Have It?

IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS) are loaded into memory when you boot. The computer won't need to reload them until your next session. COMMAND.COM is different in that sometimes it must be reloaded *during* a session. (This all happens automatically.)

When DOS loads COMMAND.COM, it divides the logic into two parts: a *resident* part and a *transient* part. The resident part, about 2,800 bytes, contains the bare essentials. It goes into the lowest available area of memory, just above IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS). The transient part, about 13,000 bytes, goes to the very top of memory. Between these two extremes is the free space, where programs are loaded and executed.

If you run a program that requires lots of memory, the upper portion of COMMAND.COM may be erased by it. (That's why it's called "transient.") This poses no problem until you exit back to DOS. At that point, the *system* disk is accessed and COMMAND.COM is reloaded. You probably won't notice it happening, unless, while running the program, you swapped diskettes and COMMAND.COM is no longer present. A message will tell you:

```
Insert COMMAND.COM disk in default drive
and strike any key when ready
```

You are more likely to get this message on a floppy-only system, especially if it has less than 256K memory. To avoid the interruption, consider putting COMMAND.COM on all diskettes that will be used in Drive A, even

those that are not system disks. Better yet, add more memory to the computer.

If you have a hard drive, COMMAND.COM is always available on it.

About Files

Besides working with devices and volumes, DOS manipulates files. A file can be any collection of information on a diskette, like a file in a filing cabinet, or a book on a shelf. Files are programs, data storage clusters, etc.

DOS is not concerned with what is in a file, nor with file names. As long as you follow a few rules, a file can be named nearly anything (though it might be smart to use names that are both descriptive and appropriate.) Here are the DOS rules:

- A file name may be up to 8 characters long, plus an optional 3 character extension. The file name is separated from the extension by a dot (period).
- Anything past the 8th character in the name and the 3rd character in the extension is ignored.
- Upper and lower case letters are all read as upper case by DOS.
- The file name and extension may contain any letters and numbers and the following symbols:

  ! @ # $ % & ( ) - { } ‘ ’ ~ _

The 3-character extension may be used to indicate what the file is used for or what type of file it is. For example, TXT is often used to indicate a “text” file. To a certain extent, the use of, as well as choice of extensions is arbitrary. However giving all files of a certain type the same extension has some advantages, as we will see later. Many application programs use specific extensions, and won’t work if their file extensions are changed.

There are 3 file extensions which definitely do matter. COM and EXE indicate *command* programs and *executable* program files, and BAT indicates an MS-DOS *batch* files, respectively. Do not use these extensions until

you really understand what they mean. A few other common extensions are:

SYS	System files
ASM	Assembly language files
OBJ	Object files
BAK	Backup files
BIN	Binary files
BAS	BASIC language files

Chapter 5 Summary

DOS controls both *files* and *devices*.

A *file* is a collection of data on a disk.

DOS refers to disks as *volumes*.

format a:/v allows user to name the disk while being formatted. 2.x Series DOS, uses **format /v**.

label (DOS 3.x only) assigns, changes, or removes a volume label.

Although standard 3-1/2" disks hold more information than the standard 360K floppies, the same commands are used to format them.

format a:/4 formats a 360K disk in a 1.2M drive.

format a: /N:9 /T:80 formats a 720K 3-1/2" disk in a 1.44M drive.

The 1.2M diskettes formatted in the Tandy 3000's 1.2M drive may only be read by a computer equipped with similar high capacity drives. 360K diskettes formatted by the 3000 are compatible with *all* Tandy's MS-DOS computers with 5-1/4" drives.

1.44M diskettes formatted in the Tandy 4000's 1.44M drive may only be read by a computer equipped with a similar high capacity drive. 720K diskettes formatted by a Tandy computer are compatible with *all* Tandy's MS-DOS computers with either a 720K or 1.44M 3-1/2" drive.

A *system* disk can start, or *boot*, the computer. Not all disks need to be *system* disks.

A *system* diskette has 3 required files. IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS) are the first 2 files on a *system* disk. They are hidden. COMMAND.COM is not hidden.

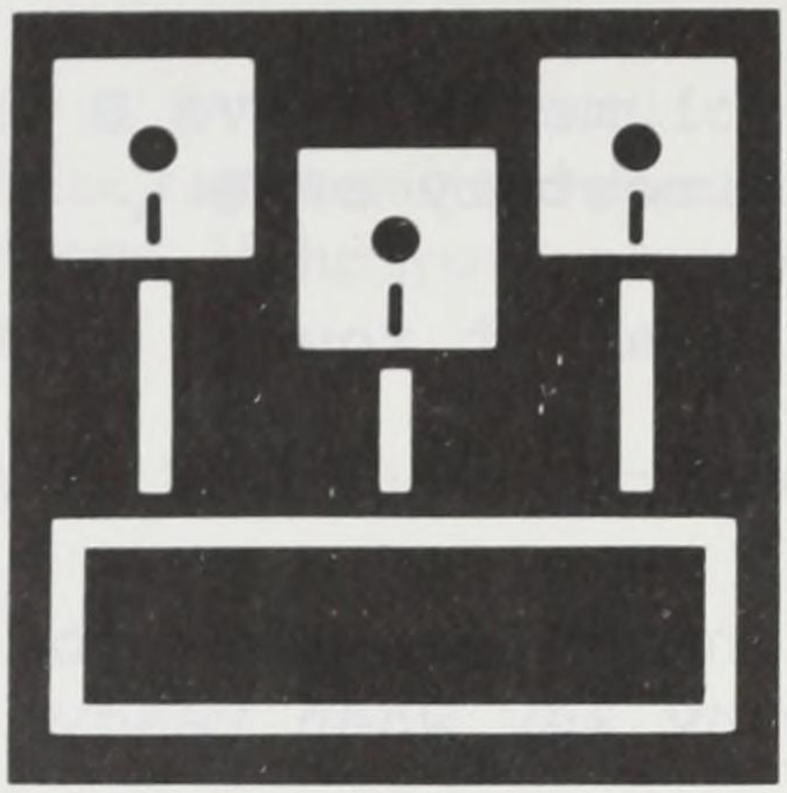
format a:/s formats a disk and copies the 3 required system files to it. The **/s** option may be used with **/v**.

format a:/b reserves space on the disk being formatted so the system files can be added later, if desired.

COMMAND.COM should be present even after you've booted because DOS sometimes needs to reload it when exiting from a program that requires a lot of memory.

File names may be up to 8 characters long plus an optional 3-character extension.

CHAPTER 6



The Electric File Cabinet

Let's try something different. Make sure a *system* diskette is in Drive A. (Users with two drives, put the DATADISK #1 diskette in Drive B.)

Now, everyone type:

dir b: ENTER

---

b: is a **dir** command option. It instructs **dir** to pull a directory of Drive B. Without **b:**, **dir** would simply list the files on the default drive, which is A.

Users with a 2nd floppy drive will see the directory of DATADISK #1 in Drive B. Users with only Drive A will see:

```
Insert diskette for Drive B: and strike
any key when ready
```

Too Smart to Fool

MS-DOS keeps track of which disk drive is which. It knows when you only have one drive but strives to make you happy by *pretending* Drive A is Drive B. Go ahead and insert DATADISK #1 disk in A, in place of the *system* disk.

Because we have formatted (and labeled) the *data* diskette but have not placed files on it, its directory reads:

```
Volume in drive B is DATADISK #1
Directory of B:\

File not found
```

Computers using DOS version 3.x will display:

```
Insert diskette for drive A: and strike
any key when ready
```

Regardless of your disk configuration, place the *system* disk back in Drive A.

Copying Around

The **copy** command is used to make duplicates of *files*, copying them from one place to another, even between diskettes. Contrast **copy** with **diskcopy**, which copies *entire diskettes* at a time.

Let's now copy the COMMAND.COM file from Drive A (where the DOS *system* disk is) to Drive B and DATADISK #1. Type:

copy a:command.com b:command.com ENTER

This reads, "Copy the file in Drive A named COMMAND.COM to Drive B, and give that file in B the same name, COMMAND.COM."

Single-drive users will be prompted to swap disks at the appropriate times.

Shortcuts

We don't have to give the copy of a file the same name as its original. In fact, we can name a copy anything as long as we stay within the file name limits set forth earlier. (But, since COMMAND.COM is a special system file, the copy cannot replace the original with the correct name.) With the *system* disk in A, everybody type:

copy a:command.com b:irving.com ENTER

and follow the instructions.

Since DOS remembers which drive is logged, the **copy** command doesn't need to specify Drive A as the *source* for the copy. If the system prompt shows A>, we would have only needed to type:

copy command.com b:irving.com ENTER

Another Shortcut

If the duplicate is to be named the same as the original, we need only include the *target* drive's letter.

copy command.com b:

reads, "Copy the file COMMAND.COM on the logged drive (A) to Drive B and name it COMMAND.COM."

Conversely

If we copy back from Drive B to Drive A, with A still the default drive, we do not need to specify the logged drive letter. So:

copy b:irving.com a:irving.com

can be shortened to:

copy b:irving.com irving.com

or even:

copy b:irving.com a:

In every case, the file named IRVING.COM is copied from the diskette in Drive B to Drive A and named IRVING.COM. The following may look like it won't work, but it does the same as above:

copy b:irving.com

This is about as much as DOS will assume.

COPY /V and VERIFY

The **copy** command's /**v** switch is used to verify the target file by comparing it with the source. Verify is to individual files what CHKDSK is to entire diskettes. Type:

copy a:command.com b:berlin.com /v ENTER

This copies the COMMAND.COM file from Drive A to Drive B and verifies the file, insuring the copy was recorded properly. Because **copy/v** does more work, it's a bit slower than an ordinary copy. But for valued files, the increased level of confidence in the integrity of the copy is worth it.

The **verify** command accomplishes the same thing. Typing:

verify on

at the command prompt causes DOS to check everything it writes to disk.

verify off

switches DOS back to its normal mode.

Just Between Devices

You may recall from the last chapter that A: and B: are *device names*. Can the **copy** command copy between *any* two devices? Let's find out.

Make sure your printer is connected and ready to run.

What would happen if we were to copy characters from the keyboard to the printer? Wouldn't we have an electric typewriter (more or less)? The keyboard is part of the device named CON. The printing device is PRN. Type:

copy con prn ENTER

The cursor is blinking at you. It's waiting for input from the keyboard (CONsole). Type the following:

The input is con ENTER
The output is printer ENTER
To get it on paper . . . ENTER
Control-Z and enter. ENTER

To tell DOS we're done, type CTRL Z (hold down the CTRL key, tap Z, and release CTRL) then ENTER. This is a special key combination which tells DOS it's the "end of file." After pressing CTRL Z, you'll see:

```
^Z
   1 File(s) copied
```

And zippity zip zip. (Readers with laser printers hear click - hummm - splat). The printer and computer work just like a typewriter. The difference is that DOS stores what's typed until it receives the "end of file" signal, Control-Z. Then it *dumps* the keyboard characters to the printer. (This "saving up" is called *buffering*.) DOS is no dummy. Why make 80 trips to the printer when one will do?

Creating Files

If we can copy characters from the keyboard to the printer, what would happen if we copied from the keyboard to a disk? Put your DATADISK #1 in Drive A and type:

copy con memo.txt ENTER

Study the above command.

The command asked DOS to copy from the console to a file named MEMO.TXT. No drive was specified, so DOS assumed you meant the logged drive, A. (To copy from the keyboard to a disk in Drive B, what should we have done? Right! **copy con b:memo.txt.**)

DOS is twiddling its thumbs, waiting for data to come in through the keyboard, or console. Type the following, pressing ENTER after each line. Use BACKSPACE (the <-- on the 3000s and 4000s) to correct errors.

I know just enuf MS-DOS to be dangerous.
I'm storing this memo on diskette.
It will be there whenever I need it.

That's enough. Tell it you are finished by typing CTRL Z, then press ENTER. DOS reports successful completion:

```
    1 File(s) copied
```

To see what happened, display the directory of DATADISK #1:

```
A> dir  ENTER

Volume in drive A is DATADISK #1
Directory of A:\

COMMAND  COM     15957   06-27-xx    1:00p
IRVING   COM     15957   11-10-xx    1:04p
BERLIN   COM     15957   11-10-xx    1:10p
MEMO     TXT       116   11-10-xx    1:14p
         4 File(s) xxxxxx bytes free

A>
```

Sure enough, MEMO.TXT is there, complete with the number of bytes, or characters, in the file (116), and the date and time it was created.

"But I only counted 110 letters, spaces and punctuation marks. How did DOS come up with 116 bytes?" Each character is a byte. Each press of the ENTER key is 2 bytes, one byte for the carriage return (back to the leftmost column) and one byte for the line feed (down to the next line). 110 + 2 + 2 + 2 = 116.

And Back Again

Now let's retrieve that memo. We stored it by copying from the console to a disk file. How about copying from the disk file back to the console? (Remember, the console is both the keyboard and the display). The **copy** command is the same, but the *source* and *target* are switched:

copy memo.txt con ENTER

And there it is, back on the screen.

Building Blocks

Having made the round trip from console to file and back again, it's time to pause and reflect.

The **copy** command simply takes data from one file or device and sends it to another. In the first case, the console was the source of input and the MEMO.TXT file was the output. In the second case, MEMO.TXT was the input and the console was the output.

Since the console can be used for input as well as output, it is known as an I/O device, one that is capable of both computer Input and Output (I and O). The same is true for disk drives. When data is *written to* a disk file, the disk drive is an *output* device. When data is being *read* from it, it's an *input* device.

Output-Only Devices

Not every device can handle both input and output. The printer is only capable of output. As far as DOS is concerned, the printer is an *output-only* device.

We copied from the console to the printer in our electric typewriter example. We'll now attempt something similar, but the source for the printer will be the MEMO.TXT file. Everyone type:

copy memo.txt prn ENTER or **copy memo.txt lpt1** ENTER

The first printer is designated LPT1, and our memo is on paper!

Other Devices

When a 2nd printer is attached to port 2, DOS refers to it as LPT2. If you have a 2nd printer attached to a 2nd printer port, type:

copy memo.txt lpt2 ENTER

One other device is part of DOS, and it's called NUL. It is like a door that leads nowhere. When you copy data to the NUL device, DOS pretends it is there . . . but it really isn't. Type:

copy memo.txt nul ENTER

Even though the screen says:

```
1 File(s) copied
```

just try to find the copy of the file! MEMO.TXT is still in the directory, but its copy is nowhere to be found. Believe it or not, NUL does come in handy when running certain programs, and we'll encounter some good applications later.

The TYPE Command

DOS has another command similar to **copy**, which is used when CON is the output device. The **type** command moves data from a file to the console. Type:

type memo.txt ENTER

And there it is again on the screen. The **type** command is a short-cut way of saying, "Copy the contents of this file to the console (screen)." It's just like:

copy a: memo.txt con

DOS will attempt to type any file you specify, but some are easier to type than others. We'll learn about them soon.

Changing Minds

Sometimes we need to give files different names than those assigned when they were first created. DOS allows both P1083.M71 and TOMOM.TXT as file names, but the second one is a little more descriptive of a letter home to Mom. The DOS command to change the name of a disk file is **rename**. With DATADISK #1 in Drive A, type:

rename memo.txt first.txt ENTER

Pull a directory to verify the name change. Now, the acid test: Use the **type** command to examine the contents of FIRST.TXT. Type:

type first.txt ENTER

There's our same memo!

The shorter and more common command is **ren**, an abbreviation for **rename**. Many of the DOS commands have abbreviations. To change FIRST.TXT back to MEMO.TXT, type:

ren first.txt memo.txt ENTER

Pull a directory and use **type** to verify the change.

The most important thing to remember about **ren** (and **rename**) is that only the name of the file changes. The contents remain unchanged. Also, **rename** can only rename files, not devices.

Clean Up Crew

The last of the "simple" file commands we'll learn in this chapter is **delete**. As the name implies, **delete** erases files from a diskette. **delete** is ruthless. Once gone, it's next to impossible to resuscitate a deleted file by using DOS.

There are special utility programs on the market which will often recover deleted files. Talk to your Radio Shack dealer about obtaining one. It might let you sleep better at night. In any case, don't take **delete** lightly!

Pull a directory of Drive A again (DATADISK #1 should still be there.) Since we don't need the COM files on a *data* diskette, we'll use them as practice targets. Type:

delete command.com ENTER

True to the pattern we see developing, this is the same as:

delete a:command.com

On a 2-disk system DATADISK #1 might be in Drive B, and we could type:

delete b:command.com

Type **dir** to verify the file is gone.

DOS not only removes the file name from the directory but also releases the disk space the file occupied. It is a good idea to kill off unused files when running short on diskette space. But, due to the unforgiving nature of **delete**, keep a second safety disk created by **diskcopy** until you are sure nothing catastrophic went wrong while erasing.

The **delete** command can be abbreviated **del**. To delete the IRVING.COM and BERLIN.COM files from the data disk, type:

del irving.com ENTER and
del berlin.com ENTER

Again, pull a directory to verify that IRVING.COM and BERLIN.COM are gone. Note the extra disk space. Only MEMO.TXT should be left on the diskette.

Chapter 6 Summary

Single-drive systems have only one *physical drive*, but MS-DOS can "pretend" there are two, A and B.

copy a:fileone b:fileone or **copy fileone b:** makes a copy of a file (named FILEONE) in Drive A to a disk in Drive B. The duplicate file will have the same name as the original.

copy a:oldfile b:newfile or **copy oldfile b:newfile** gives the copied file in Drive B a different name from the original.

copy a:oldfile b:newfile /v copies OLDFILE from the disk in Drive A to the disk in Drive B, naming it NEWFILE, and verifies that NEWFILE is properly copied.

verify on is similar to using the /v option with **copy**. It causes MS-DOS to check everything written to disk. **verify off** returns DOS to its normal mode.

copy con prn sends keyboard output to the printer.

copy con somefile creates a file named SOMEFILE on the screen.

copy somefile con displays the disk file SOMEFILE on the screen.

The keyboard, an *input* device, and the video display, an *output* device, are considered a single device called the console. Because the console is capable of both input and output, it is called an I/O device.

copy somefile prn or **copy somefile lpt1** sends the file SOMEFILE to the printer.

NUL is an imaginary device that can be used when no real input or output is wanted.

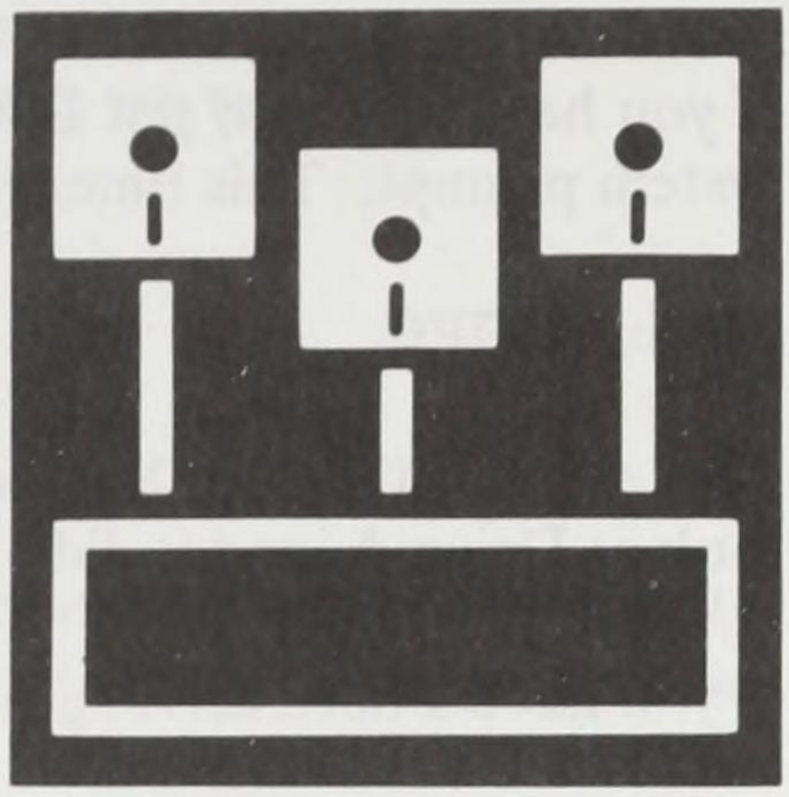
type somefile takes SOMEFILE from disk and displays it on the screen.

rename fileone first or **ren fileone first** changes the name of FILEONE to FIRST.

delete somefile or **del somefile** removes SOMEFILE from the disk.

CHAPTER 7

Exploring the Territory



We've confined most of our activity to Drive A. If your computer has more than one drive, that's like going to the movie and staying in the lobby.

Pick Your Vantage Point

With the `A>` prompt showing, type:

b: ENTER

Typing **b:** (or any drive letter followed by colon) and tapping ENTER changes the default drive to that letter. In this instance, we switched from Drive A to Drive B. The system prompt now reads:

`B>`

Take the *system* diskette out of Drive A, put DATADISK #1 in B, and type:

dir ENTER

If you have only one floppy drive, you can still display the directory of Drive B. Remember, DOS can pretend that Drive A is Drive B.

The directory of Drive B is displayed. MEMO.TXT should be the only file present.

If you have 2 drives, put DATADISK #1 in Drive A, but leave B> as the system prompt. This time, type:

dir a: ENTER

There's the MEMO.TXT file again. The default drive is B, but you said to look at Drive A, and it did.

If you have a hard drive, go ahead and make it the default one:

c: ENTER

C>

Now:

dir a: ENTER

There's the MEMO file again.

The purpose of this exercise is to show how DOS can look at a diskette from different vantage points. You can make the default drive whichever one is most convenient. Floppy drive users usually make it A. Hard disk users usually designate C.

Notice that we have not used the *system* diskette at all in this chapter. The reason is that **dir** is an *internal* command, loaded at the same time the computer was booted. The *system* diskette will be needed only when we require *external* DOS files.

Files from Afar

You've seen several examples using drive letters in a command:

dir a:
dir b:
format b:
diskcopy a: b:

In each case, the drive letter designators overrode the default drive. This same approach can be used when handling files. Just tack the drive specifier onto the *front* end of the name. With the system prompt at B> or C>, to copy from a non-default drive to the logged one, we would type:

copy a:memo.txt ENTER

Programs on Other Drives

DOS can also execute programs which are not on the active drive. After all, programs are stored as files. Just make the drive specifier the *first* part of the program name.

For example, to format a disk in Drive B with the *system* disk in Drive A, all the following commands are acceptable. (Note the system prompts indicating the logged drive.) :

A>**format b:**
C>**a:format b:**
B>**a:format b:** or simply B>**a:format**

In the above examples, the **format** command, which exists as an *external* file on the *system* disk in Drive A, is issued from another drive. Observe how the example with the B> system prompt can assume the **b:** option, because B is the default.

Assumed Parameters

You only have to tell DOS where to find programs and files if they are not on the active drive:

B>**a:diskcopy a:**

tells DOS to load DISKCOPY.COM from A, and duplicate the diskette in A onto a diskette in B. The active drive is assumed to be the target.

B>**a:diskcopy b:**

This one is tricky but completely consistent. It tells DOS to find DISKCOPY.COM on A and to copy the diskette in B onto another diskette in B. It is a 1-drive disk copy, using B as both source and target.

Actually, these examples are much like the **dir** and **copy** commands. The main difference is that the **dir**, **copy**, and **type** are *internal* commands and were loaded into memory from the COMMAND.COM file at the last boot. DOS doesn't need to load them in again. **format** and **diskcopy** are *external* commands, stored as programs on the *system* disk. DOS needs to be told on which drive they can be found.

Two-Drive Copy

With 2 floppy drives, you can copy any file from the diskette in one drive to a diskette in the other. Try this: put the *system* diskette in A and DATADISK #1 in B. Knowing that the file MEMO.TXT is on the disk in B, there are at least 8 different ways to copy it to A.

With the system prompt set to A>, the choices are:

```
A>copy b:memo.txt  a:memo.txt
A>copy b:memo.txt  memo.txt
A>copy b:memo.txt  a:
A>copy b:memo.txt
```

With the system prompt set to B>, the choices are:

```
B>copy b:memo.txt  a:memo.txt
B>copy memo.txt  a:memo.txt
B>copy b:memo.txt a:
B>copy memo.txt  a:
```

Study each of the possibilities. The end results are identical. DOS always assumes the active drive to be both the *source* and *target* and uses the same file name you specify for the *source* as the file name for the *target*, unless told otherwise. That's why:

```
A>copy b:memo.txt
```

works. Insert DATADISK #2 in Drive A. Then select one of the methods above, and copy MEMO.TXT to A. Check the Drive A directory to verify that it worked.

One-Drive Copy

You can copy individual files almost as easily with only one floppy drive, but diskette swapping is required. Here's one way:

```
A>copy b:memo.txt    ENTER
```

The computer responds with:

```
Insert diskette for drive B: and
strike any key when ready
```

Insert DATADISK #1 in A (now logical B), and press a key. The MEMO.TXT file is read into memory and back comes a second message:

```
Insert diskette for drive A: and
strike any key when ready
```

Put in DATADISK #2. When you press a key, MEMO.TXT is copied from memory. One-drive copying does require keeping the diskettes organized.

Floppy to Hard Drive Copy

You can copy any file from a hard drive to a diskette, or from a diskette to a hard drive, using the same technique as with 2 floppy drives. Only the drive designations change.

Those Wild, Wild Cards

To make copying groups of files easier, DOS employs the *wild card* concept. The asterisk character (*) and question mark (?) are used as the wild cards.

The * replaces a *group* of characters on either side of the dot (.) in a file name. With the *system* disk in Drive A and the system prompt showing A>, try these commands:

dir *.com ENTER

All the files on Drive A ending with the COM extension are listed.

dir *.sys ENTER

shows all files with the SYS extension.

dir spooler.* ENTER

All files with the file name SPOOLER are displayed — regardless of their extension.

Try these commands:

dir c*.* ENTER

All files which start with the letter C are displayed. The first * replaces the *remaining* characters (no matter how many) in the file name, and the second * replaces all characters in the extension. Now try this:

dir c* ENTER

Same thing. The * replaces *all* characters which could possibly be to its *right,* including the extension, since we didn't specify anything for the extension. This means:

dir *.* ENTER

(which displays all files on the diskette) is the same as:

dir *d.* ENTER

Since the left * replaces all characters to the right, any filename placed after it is ignored.

While * is used to replace a group of characters, a question mark (?) replaces only a single character in a filename or extension. Try these:

dir diskco??.com ENTER

DISKCOPY.COM and DISKCOMP.COM are displayed. The ? wild cards replaced the PY and the MP. Note next how ? and * can be used in tandem:

dir c?m*.* ENTER

shows only those filenames beginning with "C" and having "M" as the 3rd letter. The question mark instructs DOS to accept any single character in the 2nd position of the file name. Positions 4 through 8 and the extension can contain anything.

Try:

dir c?m.* ENTER

File not found? That's right. There are no "C?M" files with only 3 letters in their name

dir ????????.??? ENTER (That's 8 question marks, a period, and 3 more question marks.)

All the files are listed. This is the same as **dir** or **dir *.*** . The first part of the file name has 8 characters, followed by a 3 character extension. This format allows anything in each of the character positions (including *nothing*).

A question mark says, "I don't care what is in this specific position." An asterisk says, "I don't care what's in the position(s) from here to the end of the filename or extension."

Wildcard characters can be combined with drive specifications.

dir b:*.txt

lists all files on Drive B having a .TXT extension.

dir a:x*

lists all files on Drive A which have "X" as their first letter, regardless of extension.

Wildcard characters also apply to **copy**. Put the *system* diskette in A and have DATADISK #1 ready. You'll be asked for it shortly. (If you have 2 drives, put DATADISK #1 in Drive B.) Set the system prompt to `A>` and type:

copy *.sys b: ENTER

All files with the SYS extension are copied from A to B, including:

```
ANSI.SYS
KEYCNVRT.SYS
LPDRVR.SYS
SPOOLER.SYS
VDISK.SYS
```

Pull a directory of B to see that they are all there, along with the MEMO.TXT file.

A REALLY B-I-G System Diskette

This section is for 3000 users with 1.2M floppy drives and no hard drive. 4000 users with a 1.44M drive can use the same technique.

When we made our safety copy of the *master disk*, we used **diskcopy**. However, because the master diskette came in the standard 360K format, **diskcopy** made a corresponding copy. To put the DOS system files on a 1.2M diskette and thereby have lots of space left to store our own files, we need to take a different approach.

We first need to format the high-density disk, using the /**s** option to transfer the system files. Type:

format a:/s ENTER

Replace the *system* disk with a new high-density diskette, and press ENTER.

The next job is to transfer the external DOS commands to the new 1.2M *system* disk. Place the 360K *system master* back in Drive A. We'll use **xcopy** to make the transfer:

xcopy a:*.* a: ENTER or **xcopy *.* a:** ENTER

We'll learn more about the **xcopy** command in Chapter 29.

When the copying is completed, pull a directory of the 1.2M diskette. You now have a new working *system* diskette for 1.2M drives, and look at all that empty space!

Chapter 7 Summary

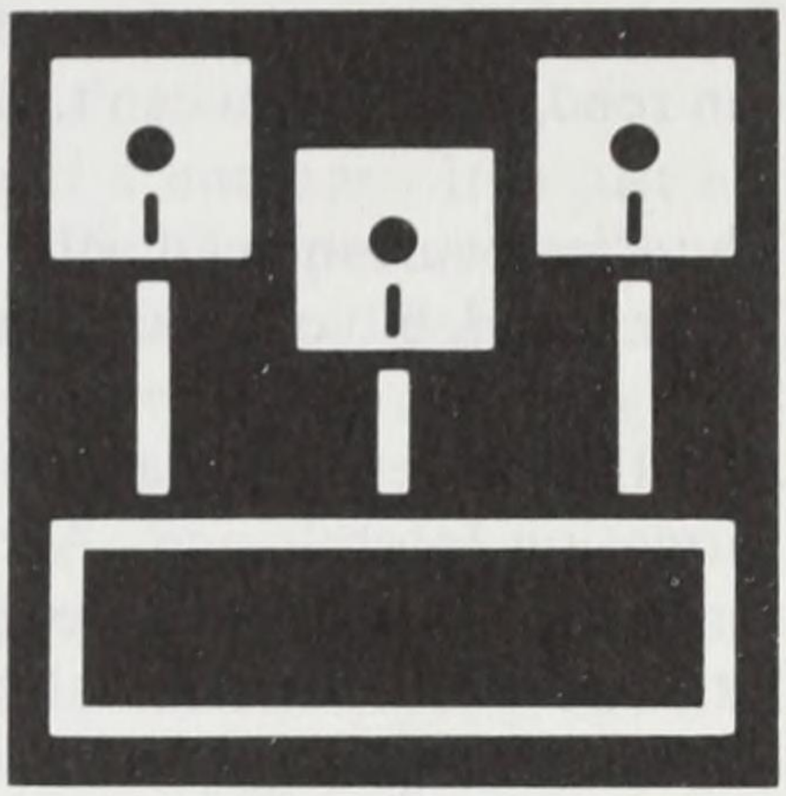
Typing **a:**, **b:**, or **c:** changes the active drive to A, B, or C respectively.

Commands and file names may be specified in many ways. A *drive* specifier before a file or program name overrides the active drive. Omitting a drive letter from a command causes MS-DOS to assume the active drive.

The wildcard characters, * and ?, are used with commands when multiple file names need to be designated. The * replaces all characters to its right; the ? replaces only a single character.

3000 users with 1.2M floppy drives and no hard drive can create a 1.2M *system* disk by using **format a:/s** to format a high-density disk and **xcopy** to transfer the external DOS commands. 4000 users can do the same thing with their 1.44M high-density drive.

CHAPTER 8



Say It in ASCII

So far, using DOS has been like being a warehouse manager. When you display a directory, you're just taking inventory. When you change the active drive, you walk into another storage room.

This is appropriate because DOS is not concerned with the contents of the files it manages. It doesn't know what the files are for and doesn't know what to do with them until you tell it. It is up to *you* to specify the right file for the situation.

ASCII vs. Binary

Try this simple experiment. Put the *system* diskette in Drive A, change the system prompt to `A>`, and:

type format.com ENTER

That jumble of letter, numbers, and symbols looks like it is from a wall inside the Great Pyramid at Giza, but it's only the contents of the FORMAT.COM file.

Now put DATADISK #1 in A and:

type memo.txt ENTER

Ahh! Nice, neat, and readable.

Try the same thing with any file in the directory of any diskette. Some you can read; others you can't. The difference is very important.

The files you can read with the **type** command are known as *ASCII* files. The unreadable ones are *binary* files.

ASCII (rhymes with passkey) stands for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. According to the ASCII standard rules, each letter or digit has its own identifying number. The number codes for all the letters, digits, and punctuation marks are standardized, so they are the same on other computers that follow the same rules. Other standardized ASCII codes represent commands like "end of line," "new page," and "end of file" (Control-Z, remember?). An ASCII file is highly predictable. The **type** command lets you read ASCII files. Glance quickly at Appendix C.

On the other hand, binary files don't conform to any standard rules. Binary file codes have special meanings known only to the program using it or, in the case of a program file, to the particular type of computer intended to run it. When you attempt to **type** a binary file, DOS doesn't know any better. It tries to display it, assuming the file is in ASCII.

When displayed by the **type** command, an "A" seen in a binary file might not represent a letter at all. Perhaps it stands for the number 65. The code "n&" in a binary file might be meaningless to you, but to the program that uses it, this code might be a compact way of representing the number 9838. "H" might not even represent a letter or a number. Depending on its location and how the file is used, it might be a code that means, "Subtract 1 from the current total in the accumulator." Any letter or symbol in a binary file can have a number of meanings. The exact meaning depends on where it is in the file, what codes are before and after it, and how the computer is to interpret the data.

The ASCII Codes

An ASCII file consists of printable letters, digits, and special symbols, along with invisible control codes that tell how the file is to be displayed, printed, or processed. It is important to know a few of these codes

First, what is a byte? A byte may be thought of as a unit of measure, like a pound or meter or minute. We speak of files as being so many "bytes long". But how big is a byte?

Think of a byte as a little bucket that can hold a number. It is just big enough to hold any whole number from 0 to 255. Computers love numbers, but they know we need letters too. They work only with numbers internally, translating them to letters and symbols when communicating with humans.

Try this. Hold down the ALT key with a finger of your left hand. Then, with your right hand on the number keys in the numeric keypad, press 6 then 5. Nothing shows on the screen, yet. Now release the ALT key. What do you see?

You should see the letter A. If not, try it again.

Keying in a number from 0 to 255 while holding down the ALT key is a special way to enter any ASCII number. You just entered the byte of information with the ASCII value of 65. In ASCII, the number 65 stands for an A. (See Appendix C.)

Now try ALT 6 6 and ALT 6 7. There's B and C. Bytes 65 through 90 represent the uppercase letters A through Z.

Key in ALT 9 7, ALT 9 8, and ALT 9 9. You see abc. Bytes 97 through 122 represent the lower case letters a through z.

It might seem that the digits 0 through 9 are already numbers so they don't need translating. Not so. In ASCII, they are assigned to numbers 48 through 57. Key in ALT 4 8. There's the 0 character.

Punctuation marks also have codes. Key ALT 3 2 and see a blank. Try ALT 3 3, ALT 3 4, and ALT 3 5.

Control Codes

In ASCII, code numbers 0 through 31 have their own special meanings. With at least several characters on the screen, try ALT 8. The 8 means backspace. Try it again two or three times.

Here's a shortcut. Hold down the CTRL key and press H. Backspace again!

Why H? "H" is the 8th letter of the alphabet. CTRL with any letter key sends a byte to the computer equal to the letter's position in the alphabet. In other words, CTRL H is the same as ALT 8.

Try ALT 1. You see ^A. Try CTRL A. Same thing. Control-A has no meaning to DOS, so rather than "doing something," it just "echoed" its symbol for CTRL (the caret) and A.

Other codes from 0 to 31 have special meanings. Try ALT 1 3 or CTRL M.

What happened? CTRL M or ALT 1 3 is the code for ENTER. Upon receiving the code for ENTER, DOS thought you wanted to execute a command or program, but since it was waiting for an ENTER, nothing happened.

ASCII values 0 through 31 are called *control codes* for two reasons. First, they are usually entered with the CTRL key. Second, they control the way data is displayed and printed. They control the computer and printer.

A few of them are important to know.

ASCII Value	Ctrl Key	Same As	Meaning
3	^C	Break	Cancel current operation
8	^H	Backspace	Erase last character
9	^I	Tab	Skip right to the next tab stop*
10	^J	Ctrl-Enter	New Line
12	^L	Ctrl-Home	New Page
13	^M	Enter	End of line (Carriage Return)
26	^Z	F6	End of file
27	^[	Esc	Void current line or Escape**

* Tab stops are at every 8th column on the display.

** In some situations, ASCII 27 is used to increase the number of control codes available. It may indicate a special, non-ASCII meaning for the number or numbers that follow.

You might have noticed that, though the FORMAT.COM file is over 11000 bytes (according to the directory), nowhere near 11000 characters were displayed. That's because a byte having a value of 26 was encountered.

The **type** command, finding the code for Control-Z, thought it reached the *end of file* (when it really hadn't). This is another important distinction between ASCII and binary files. Control-Z indicates end of file for ASCII files. For binary files, DOS uses the length indicated in the directory.

Other ASCII Characters

Reference manuals provided with the computer have a table that shows the ASCII values from 0 to 255. They show what keys can be used as "short cuts" and what characters are displayed on the screen.

Only codes 0 through 127 are widely accepted as ASCII standard. Codes from 128 to 255 are used for special symbols. Key in ALT 1 2 8, ALT 1 2 9, and ALT 1 3 0, and you'll get the idea. These are useful for non-English text.

Try ALT 1 7 6, ALT 1 7 7, and ALT 1 7 8. These, and other higher numbered codes, are used as graphics characters. They can dress up the display with lines, bars, and boxes.

While the codes from 128 to 255 are not universally accepted as ASCII standards, they are fairly standard among "IBM compatible" microcomputers like the Tandy MS-DOS models we are studying here.

What Is the Real Advantage of ASCII?

The ASCII standard provides the basis for international computer standards. Any computer or computer program which supports ASCII standards can read ASCII files from another computer, which may or may not be under the control of MS-DOS. This common basis permits moving data among the many different type files such as word processors, data base managers, spreadsheets, and BASIC programs. ASCII files can be easily and reliably sent from computer to computer via a cable or by radio link or satellite by adding modem converters at each end.

Using an example that's "close to home," this book was written on a number of different computers and word processing programs in different parts of the world. Most parts were written, tested, checked, and edited in the CompuSoft laboratories using various Tandy computers, with many versions of MS-DOS and several word processors. Other parts were written

on non MS-DOS Tandy portable computers and sent to CompuSoft via telephone line and via satellite from Europe. Some chapters were stored on diskette and sent back from Hawaii via overnight air express. The ASCII standard format was the bridge that connected these many "incompatible" computers and locations.

In each case, the text was translated into ASCII and dumped into a Tandy 3000 for assembly into chapters and final "touch-up." From the 3000, some text was sent via a short cable into a Linotype CRTronic® computerized digital typesetter where the keystrokes were converted into images on film. The rest was loaded into special publishing software for typesetting.

Therefore . . .

MS-DOS has many powerful and useful commands expressly for use with ASCII files. The upcoming chapters deal with commands that create, sort, filter, and manipulate ASCII files.

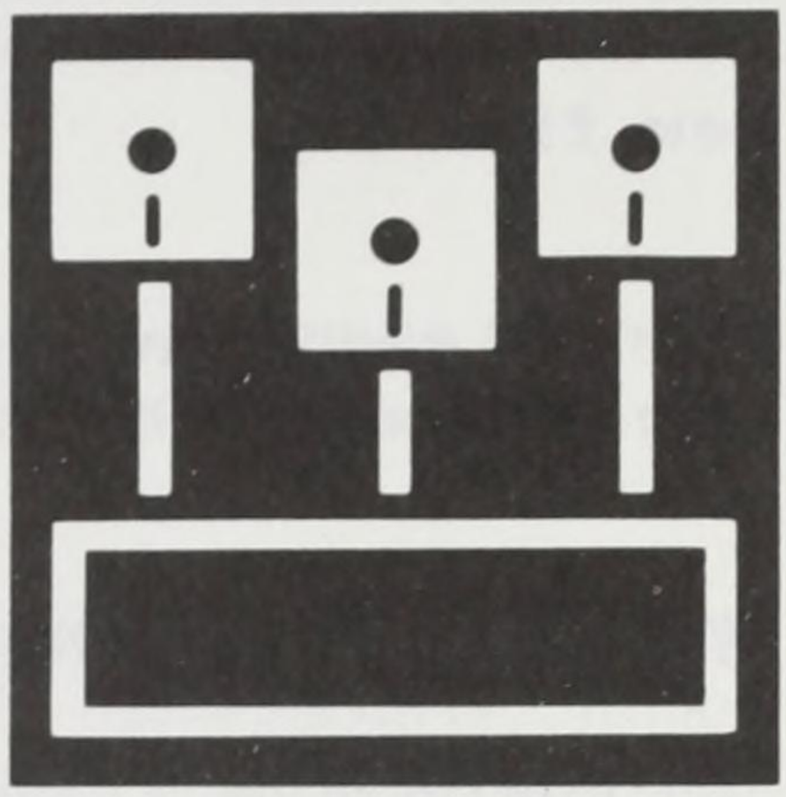
Chapter 8 Summary

It is important to know the types of files and their purposes. DOS expects certain kinds of files in specific situations.

An ASCII file is one that conforms to certain computer standards. Each byte represents a character or control code. ASCII files can be read by the **type** command. Binary files cannot.

Important control codes in ASCII files are ^M (carriage return), ^J (new line), ^I (tab), ^L (new page), and ^Z (end of file). Any byte value from 0 to 255 can be entered by pressing ALT and keying in the number.

CHAPTER 9



The DOS Editor

DOS contains a special program named EDLIN for creating and editing ASCII files. You can use it to create and edit lists, memos, notes, letters, computer programs, or any other ASCII file. EDLIN (pronounced Ed'-lin) is a "line oriented" text editor best suited for working with one line of characters at a time, rather than with paragraphs, pages, or chapters.

Before we start, we need to make room on our *system* disk. Type:

del basic?.* ENTER

> If you're working with a 1.2M or 1.44M *system* disk, you won't need to do this. You have plenty of room.

Text Editing Using EDLIN

Before using **edlin**, some users must copy EDLIN.COM from a Supplemental Programs disk to their *system* diskette. Take a directory of your *system* disk to see where you stand. We will create a small **edlin** "help file" just for practice and name it HELP.TXT. At the `A>` prompt, type:

edlin help.txt ENTER

DOS loads **edlin** from the *system* disk into memory and checks the disk for a file named HELP.TXT. Not finding it, **edlin** creates one. We see:

```
New file
*
```

The "star" is **edlin**'s command prompt. It serves the same purpose as the > prompt does at DOS level.

The Insert Text Command

The **edlin** command to **i**nsert text is:

i ENTER

Type it and see:

```
1:*
```

edlin assigns each line of text a reference number. To add words to the file, just type them, pressing ENTER at the end of each line. Type the following, *exactly* as shown including the "strange" phrasing and the "errors." Use BACKSPACE to correct your own typos:

```
1:*MS-DOS EDLIN Help File     ENTER
2:*ENTER
3:*The commands are:    ENTER
4:*TAB  I for insert a line of text    ENTER
5:*TAB [CTRL] [C] to end inserting     ENTER
6:*TAB  D to delete a line of text     ENTER
7:*TAB  X to make the computer explode     ENTER
8:*TAB  L to list lines of text     ENTER
9:*TAB  E to end or then an editing session     ENTER
10:*TAB  Q to abort and quit an editing session     ENTER
11:*CTRL C
*
```

Listing the File

To see what we've created, use the **l** (list) command. Type:

l ENTER

Hmmm. Line 7 looks a little suspicious. Is that a real **edlin** command? Let's double check that line by listing it. Type:

7,7l ENTER (That's 7,7 L.)

This approach to listing may appear a bit strange, backwards in fact, to most of us. Don't worry — it is. To list a specific line, or group of lines, **edlin** wants us to list those line numbers first, followed by the command **l**. Because we told **edlin** 7,7, it listed lines 7 through 7, which means *only* line 7. If we want to see the range of lines from 1 through 7, the command is:

1,7l ENTER

The , (comma) means "through." So "1,7" reads "lines 1 through 7."

How to Delete

To delete line 7, use the **d** command. Like **l**, the line number (or range of numbers) needs to come first. Type:

7d ENTER

And it's gone. List the text to verify:

l ENTER

Note how **edlin** renumbered the lines. Everything falls into place.

See the * by line 7? It means line 7 is the line we are currently working with. If the **i** command was used now (alone, without a line number), it would start inserting new line numbers at line 7. The "*" means, "This is the current line."

Appending to the EDLIN File

There's one more important command. Besides Control-C to end an inserting session, we can use Control-Z (the end of file). To insert after the last line of text (line 9), type:

10i ENTER

Type the following:

```
10:*TAB[CTRL][Z] (then [ENTER]) also ends    ENTER
```

To verify this text, on the next line press CTRL Z, then press ENTER. List again with **l**. It works!

Before we're done, let's insert a blank line between lines 3 and 4. Use the **i** command as follows:

4i ENTER

which means, "Insert a line before line 4." We see:

```
4:*
```

Whatever we type here will become the new line 4, and all the lines following line 4 will increase by one line number. Because we just want a blank line, press ENTER, and see:

```
5:*
```

Since we don't want a new line 5, press CTRL C to end the insert. List all the lines:

l ENTER

See the * by the 5? It means 5 is now the current line.

Our text should look like this:

```
1: MS-DOS EDLIN Help File
2:
3: The commands are:
4:
5:*  I for insert a line of text
6:    [CTRL][C] to end inserting
7:    D to delete a line of text
8:    L to list lines of text
9:    E to end or then an editing session
10:    Q to abort and quit an editing session
11:    [CTRL][Z] (then [ENTER]) also ends
```

Ending It All

One way to end is with **q** to quit, or abort the edit, losing all the changes we made. Type:

q ENTER

edlin wonders:

```
Abort edit (Y/N)?
```

Don't touch a thing! When you use **q** and press Y, the text file edits are lost. In this case, since we have not yet saved our file, the entire file would be lost. Everyone press N.

To exit **edlin** and save the file complete with all changes, use the **e** command. Type:

e ENTER

edlin saves the file to disk and returns us to the DOS system prompt.

Meanwhile, Back at DOS . . .

There are 2 ways we can examine the new file from DOS. First, pull a directory to see that the new file exists. Type:

dir help.txt ENTER

> Remember, when followed by a single file name, the **dir** command displays information only about that file.

To view the file, type:

copy help.txt con ENTER

or

type help.txt ENTER

And there it is! How about a *hard copy*? Send our help file to the printer with:

copy help.txt prn ENTER

Editing a File Using EDLIN

Oh, oh! It looks like we should change the wording in a few of those lines. Back to the trenches . . .

Load in **edlin** again and re-edit HELP.TXT:

edlin help.txt ENTER

MS-DOS loads **edlin** into memory and displays:

```
End of input file
*
```

List it with:

l ENTER

Our first major grammatical mistake appears in line 5. To edit that line, type **5** and press ENTER. We see:

```
5:*  I for insert a line of text
5:*
```

The cursor waits for us. The most obvious thing to do might be retype the entire line. But the only real change we need to make is to replace "insert" with "inserting." So try this: tap the --> (right-arrow) key 13 times.

Incredible! It's almost like **edlin** *copied* a character from the line above each time we pressed -->! That is exactly what happened.

edlin has a small memory that remembers the *last line* worked on, or in the case of changes, the line you are changing. The --> key copies characters from this memory, one character at a time. The F1 key does the same thing. Press F1 7 times. Now the screen looks like:

```
5:*   I for insert a line of text
5:*   I for insert a line
```

To backtrack, press the <-- (left-arrow) key. This is the same as pressing the BACKSPACE key. For now, press <-- (or BACKSPACE) 7 times to move back 7 characters in the *template*.

EDLIN Rapid Transit

Let's forget what we've done with line 5 so far. Press the ESC (escape) key. This is the "Eh, forget it!" key. It allows us to start over without erasing the **edlin** line, bringing back the original line from memory.

edlin displays a backslash (\) then gives us a fresh line. But is the memory intact? Press --> a few times to be sure.

There it is.

Press F3.

Whoa! There's the whole line. In **edlin**, F3 is the "display memory" key. It's equivalent to tapping the --> or F1 key until the end of the current line. Press <-- or BACKSPACE until your screen looks like this (make sure the cursor is in the space right after the "t" in "insert"):

```
5:*  I for insert a line of text
5:*  I for insert\
     I for insert
```

To insert "ing", press the INSERT or INS key. This allows inserting text until we press INSERT again. Type:

ing INSERT

Now press F3 to display the rest of the line. Press ENTER to lock the edits into memory, then **l** to list our new creation.

If you type characters without pressing INSERT, those characters *replace* text in the memory. Once the memory contents have been changed, the only way to retrieve the old one is ESC then F3.

Minor Alterations

Line 9 somehow looks mixed up. The "or then" can be deleted with no ill effects. To edit line 9, type 9 ENTER at the * prompt, and see:

```
9:*  E to end or then an editing session
9:*
```

We could press --> or F1 10 times, but let's use Edlin's "search to" key instead. Press F2, then O.

```
9:*  E  to end or then an editing session
9:*  E  t
```

F2 O tells Edlin to "Search until you find the first letter o." Press F2, then O again. And here we are. To delete the next 8 characters from the template, press the DELETE or DEL key 8 times. Press F3 to display the rest of the line.

```
9:*  E to end
9:*  E to end an editing session
```

To lock in the changes, press ENTER.

The Final Key

Let's add some lines to our help file which will describe the line editing keys. Everyone type:

12i ENTER

```
12:*ENTER
13:*Line Editing commands are:
14:*ENTER
15:*TAB  [F1] or [-->] to move one character right    ENTER
16:*TAB  [BACKSPACE] or [<--] to move one character left    ENTER
17:*TAB  [F3] to display all of the current line    ENTER
18:*TAB  [F2] to search (move to) a specific character    ENTER
19:*TAB  [F4] to delete up to a specific character    ENTER
20:*TAB  [F5] to perform acts of wizardry and magic    ENTER
21:*CTRL Z ENTER
```

Hmmm. Is line 20 really true? We'd better edit line 20. Type:

20 ENTER

Enter the following for line 20:

TAB **[F5] to replace the line in memory with the line edited**

Then press F5. **edlin** displays:

```
20:*  [F5] to perform acts of wizardry and magic
20:*  [F5] to replace the line in memory with the
line edited
```

Now press F3. There's our new line. Press ENTER to lock everything into memory.

We know F3 means, "Display the rest of the line." Well, F5 translates as, "Replace the line in memory with *this line* and start over." Once F5 is pressed, F3 will display the memory as F5 replaced it. F5 is a convenient way to "lock in" changes without pressing ENTER.

List the entire help file, then save it to disk. Try issuing both commands at once, type:

le ENTER

> **edlin** allows commands to be "stacked up," one after the other. Here we've put the **l**, list-text command, before the **e**, save to disk and exit.

The edited help file is now on the system disk. Type:

dir /p ENTER

There it is. Where did that second HELP file, the one with the BAK extension, come from? Whenever an *existing* file is loaded, worked on, and then saved, **edlin** automatically creates a second, or backup, copy. More about backup copies later. For now, delete that second copy to free up disk space. Type:

del help.bak ENTER

Once in DOS, make a hard copy by typing:

copy help.txt prn ENTER

Other Editing Commands

The *move* command lets you move a line or range of lines to another position in a file. For example, **5,7,3m** moves lines 5 through 7 to begin at line 3.

If more than one copy of a line(s) is needed, use Edlin's *copy* command. To copy lines 15 through 18 to begin at line 2, use **15,18,2c**.

To *search* for a specific word, such as "Tandy," located between lines 20 and 40, use **20,40sTandy**.

To *replace* "Tandy" with "Radio Shack," between lines 20 and 40, use **20,40rTandy^Radio Shack**.

Advanced EDLIN

Because **edlin** is a word processor of sorts, it has a number of progressively more esoteric editing features. Their mastery requires frequent usage, so they will be of interest only to advanced programmers. We refer those few readers who can benefit from knowledge of them to the DOS manual which came with your computer.

There are, however, additional **edlin** commands which are very important to all of us. They are the subject of the next chapter.

Chapter 9 Summary

edlin is DOS's text editor. It creates and edits ASCII files.

edlin newfile loads **edlin** and the file named NEWFILE. If a file named NEWFILE does not exist, **edlin** creates it.

edlin's command prompt is a "star" (*). The following commands may be entered at the * prompt:

- **i** inserts new lines into a file.
- **l** lists the file. **4,35l** lists lines 4-35.
- **d** deletes and renumbers the remaining lines. **5d** deletes line 5.
- **q** quits **edlin** without saving the changes made to the file.
- **e** exits **edlin** and saves the file and the changes to disk.

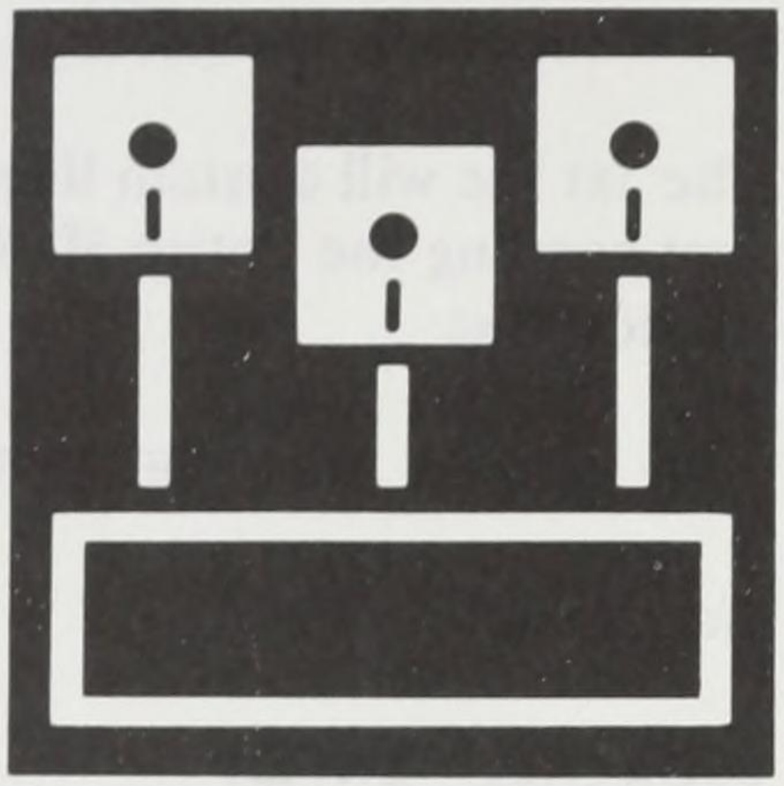
edlin displays a line number for each line to be inserted or changed. The last line worked on is saved and can be copied using the --> or F1. Other editing keys include:

BACKSPACE or <--	erases the last character typed.
ESC	voids the current changes made to the line.
F2 O	moves cursor to the first "o."
F3	displays all of the current line.

F4	deletes up to a specific character.
F5	replaces the line that was originally saved with the new one.
INSERT or INS	inserts into the saved line.
DELETE or DEL	deletes characters from the saved line.
ENTER	accepts the new line and displays the next line number for editing.
CTRL C	ends an inserting session.
CTRL Z	ends an **edlin** file and returns the * prompt.

Additional editing commands include **m** (*move*), **c** (*copy*), **s** (*search*), and **r** (*replace*).

CHAPTER 10



Advanced EDLIN

We've learned the basic features of **copy** and **edlin**, but there are several more powerful actions that **edlin** can perform on ASCII files.

Splitting a File

Why would we want to split a file? Here are some possibilities:

- You have a 260,000 byte file containing text for a company policy manual. Unfortunately, your word processing program is unable to handle a file that big. You want to break it into smaller files to change some policies.

- You have a big ASCII file on your hard drive and want to split it into several small files that can be stored on floppy diskettes.

- You just finished an online session with CompuServe. During the telephone hookup, you *downloaded* a file of stock market quotes plus a list of airline schedules — into the same file. You now want to split the file into two separate ones.

***edlin** can work with ASCII files of any size.*

We'll learn how to split a file using HELP.TXT from the last chapter. We'll split a copy of it into 3 new files, but leave the original file intact.

First we need to know about the pound sign (#). When used in place of a number, # means, "the next line beyond the last line in the file." So **#i** means, "insert a new line at the end of the file."

The 1st file will contain lines 1-7 and be named FILE1. We'll create it by first copying the entire HELP.TXT file to FILE1, then removing the unwanted text:

copy help.txt file1 ENTER

Now go into **edlin**:

edlin file1 ENTER

Delete all lines beyond 7:

8,#d ENTER

(**8,#d** says, "Delete everything from line 8 to the end of the file." That way we don't have to look up the last line number.)

List the file. Nothing should remain but lines 1-7.

Exit **edlin**:

e ENTER

The edited file is saved, and you are back at DOS. Display the file to check it:

type file1 ENTER

Use the same process to create the 2nd file.

copy help.txt file2 ENTER
edlin file2 ENTER
16,#d ENTER
1,7d ENTER
e ENTER

Now look at the result.

type file2 ENTER

Nothing should remain but the original lines 8-15, now renumbered as 1-8.

Finally, create FILE3:

copy help.txt file3 ENTER
edlin file3 ENTER
1,15d ENTER
l ENTER
e ENTER

Display the result — lines 16-20, renumbered as lines 1-5.

The job is complete. You copied one file and split it into 3 smaller files, and the original file was not altered.

The Transfer Command

The **edlin** command that allows merging of one ASCII file into another is called *transfer*.

To see how the **t** command works, type:

edlin letter.txt ENTER

Begin the letter by inserting **Dear Robert**: Then press ENTER and CTRL C. Here's the screen:

```
*i
   1:*Dear Robert:
   2:*^C
```

Now put DATADISK #1 in Drive A and type:

#t memo.txt ENTER

List the file in memory to see what you did:

```
1: Dear Robert:
2:*I know just enuf MS-DOS to be dangerous.
3: I'm storing this memo on diskette.
4: It will be there whenever I need it.
*
```

The **#t** *transfer* command allowed you to append MEMO.TXT to the end of the file that is now open in Edlin.

Add another line at the end of this file:

```
*#i
5: *This exercise is as exciting as a perilous journey.    ENTER
6: *ENTER
7: *CTRL C
```

Use the *transfer* command again to append text from FILE1, FILE2, and FILE3. With the *system* diskette in A, simply type:

#t file1 ENTER
#t file2 ENTER
#t file3 ENTER

The *transfer* command consists of the letter **t**, preceded by a line number reference and followed by a disk file name. We used the **#** to insert the text at the *end* of the file in memory. This is the most common way, but you can transfer to anywhere in the file if you put a line number before the **t**. If you omit a number or **#**, **edlin** inserts at the current line — the one flagged by an asterisk.

Now list the file. Since it will require more than one screen, use **edlin**'s *page* command. Type **1p** ENTER and the first page is displayed:

```
*1p
 1: Dear Robert:
 2: I know just enuf MS-DOS to be dangerous.
 3: I'm storing this memo on diskette.
 4: It will be there whenever I need it.
 5: This exercise is as exciting as a perilous journey
 6:
 7: MS-DOS EDLINE Help File
 8:
 9: The commands are:
10:
11:    I for inserting line of text
12:    [CTRL] [C] to end inserting
13:    D to delete a line of text
14:    L to list all lines of text
15:    E to end an editing session
16:    Q to abort and quit an editing session
17:    [CTRL] [Z] (then) [ENTER] also ends
18:
19: Line Editing commands are:
20:
21:    [F1] or [-->] to move one character right
22:    [BACKSPACE] or [<--] to move one character left
23:*   [F3] to display all of the current line
```

Press P ENTER for the next page:

```
24:    [F2] to search (move to) a specific character
25:    [F4] to delete up to a specific character
26:*   [F5] to replace the line in memory with the line edited
*
```

Now, use **e** ENTER to save the LETTER.TXT file and return to DOS:

Use the **type** command to display this letter you created:

type letter.txt ENTER

Send a copy to the printer with:

copy letter.txt prn ENTER

Editing Big Files

edlin can work with ASCII files of any size, but if the file is more than about 50,000 bytes, **edlin** must handle it in sections, working from beginning to end.

You are accustomed to one of two messages when starting **edlin**: `New file` and `End of input file`. When editing a large file, **edlin** displays neither upon startup. You only get the * prompt. This indicates **edlin** has loaded in as much as it can, but has not yet reached the end of the file.

Let's say, for example, you have a file with 2000 lines, and **edlin** isn't able to load them all. You can check the last lines in memory with:

ENTER
l ENTER

Suppose the display shows that the last line is 1099. To work on anything past line 1099, you must tell **edlin** to write some of the lines from the beginning of the file to disk using the *write lines* command. For example:

500w ENTER

writes the first 500 lines to disk, then deletes them from memory. Now there's enough space to bring in some (or all) of the lines beyond the (original) 1099th line using the *append lines* command:

a ENTER

Edlin loads what it can from the rest of the file. The new lines are added after whatever is still in memory. When **edlin** reaches the end of the original file, it displays `End of input file`. If you don't get this message, you can use **#** ENTER **l** ENTER to see how far it got. If necessary, use the **w** and **a** commands again to reach the part of the file you want to edit.

The *write* and *append* commands can be used with or without a number. Without numbers, **edlin** writes or appends until 75% of the memory space

is full. 25% is left free for insertions and replacements that may require space. Here are some examples:

100w	writes the first 100 lines from memory to disk.
100a	appends the next 100 lines from disk to memory.
w	writes lines from memory to disk until 25% of memory is free.
a	appends lines from disk to memory, until 75% of memory is full.

Combining Files

edlin's *transfer* command can combine files. The MS-DOS **copy** command can also combine files, as long as the *total* of their lengths is less than about 64,000 bytes. Since **copy** is an *internal* command, it is always available at the DOS system prompt.

Let's combine 3 files into one. With the *system* disk in Drive A, type:

copy file1+file2+file3 all3 ENTER

The "+" symbols separate the 3 files to be combined. A space precedes the destination file name.

The display lists each file as it is combined:

```
FILE1
FILE2
FILE3
        1 File(s) copied
```

The result is a new file called ALL3. Inspect it:

type all3 ENTER

What we took apart earlier in this chapter is now back together under a new file name.

Appending Files

The DOS **copy** command can also be used to *append* files. Instead of creating a new file to hold the result, the first file just gets longer.

For example, suppose you want to add a second copy of FILE3 to the end of the ALL3 file:

copy all3+file3 ENTER

This differs from "combining" because no new destination file is created. The destination file is the first one listed in the command.

Display ALL3 to see what you have:

type all3 ENTER

Combining and Appending Binary Files

Using + in a **copy** command is most useful with ASCII files. Here's what goes on:

When combining, DOS creates an "empty" destination file. Then, one by one, each source file is copied into it until its Control-Z is found. After the last one, DOS writes a new Control-Z to mark the end of the combined file it has created.

When appending, DOS looks for the Control-Z that marks the end of the first file, adds the second file at that point, and moves the Control-Z to the end.

Knowing this, you'll understand why binary files require special treatment when appending or combining. Binary files don't use a Control-Z to mark the end. They may have Control-Z's anywhere. To avoid confusing DOS, use **/b** after the file names when combining or appending binary files. When ASCII files are to be combined with binary files, use **/a** to identify the ASCII files.

Here's what a mixed binary and ASCII combining command might look like:

copy mainprog.bin/b+screen1.bin/b+helpfile.txt/a mainprog.com/b

The result is a new file called MAINPROG.COM.

Fortunately for most of us, there is rarely a reason to combine binary files. It's not always as simple and reliable as shown here. But if you have a reason, it is important to know how the files are organized.

Chapter 10 Summary

edlin can split a large file into smaller files or extract unwanted data from a file. Copy the file, then use **edlin** to delete lines from the copy.

In **edlin**, a pound sign (#) used in place of a number means, "the next line past the end of the file."

edlin's *transfer* command merges lines from other files into the file you are editing. **24t b:checklst.txt** inserts the file CHECKLST.TXT from Drive B at line 24 of the file being edited.

Page breaks can be inserted into **edlin** files. 26i ENTER CTRL L ENTER CTRL C forces a *form feed* just before line 26.

edlin's *page* command is used to view files longer than one screenful. **1p** displays the first "page" of an **edlin** file.

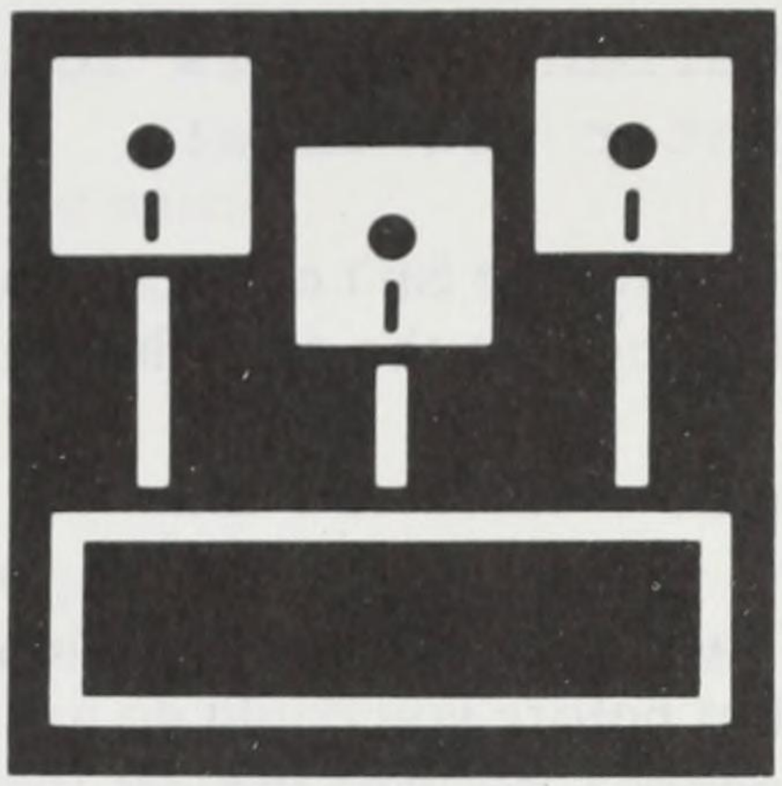
edlin can edit files larger than the available memory. Work on the file from beginning to end, one section at a time. The *write* command frees memory by writing lines to disk. The *append* command loads additional lines into the memory that has been freed. **100w 100a** writes the first 100 lines from memory to disk and loads 100 more lines for editing.

Another way to combine files is with *copy-combine* and *copy-append* at the MS-DOS system prompt. Use **copy** with file names and plus (+) symbols. **copy myfile.txt+yourfile.txt ourfile.txt** combines two files into a third file. **copy yrdata+modata** lengthens the YRDATA file by appending a copy of the MODATA file.

copy's *combine* and *append* options are most useful for ASCII files. When combining or appending binary files, use /**b** after binary file names and /**a** after ASCII file names.

CHAPTER 11

Redirecting Input and Output



DOS allows you to easily redirect the computer's flow of information. What is normally entered from the keyboard, for example, can be supplied by a file. What is sent to the screen can be sent to a printer or to a file or out over a telephone line. In this chapter we will begin studying how to change the flow of data traffic.

Automatic Answers

What's the simplest thing the computer asks you to do?

```
Strike a key when ready.
```

Too demanding? OK, let's create a file that does the job. At the `A>` prompt, type:

copy con enter ENTER
ENTER
CTRL Z ENTER

The file named ENTER now contains the code for a tap on the ENTER key. We'll use it in a second, but first, type:

time ENTER

The computer responds with:

```
Current time is 10:17:01.70
Enter new time:
```

If that time isn't correct, change it, then press ENTER. Now let's try again, linking it to the new file named ENTER:

time < enter ENTER

The time was displayed and you were offered the opportunity to change it, but before you could do anything, the `A>` prompt returned. What happened? We used the less-than symbol (<) to obtain *input* from the ENTER file. That input was the pressing of the ENTER key which caused DOS to accept the `Current time`.

Making a Current Date File

Let's take this idea a little further. Suppose we want a file that contains the current time. Try this (watch the spacing):

time < enter > curtime ENTER

Where did the `Current time is` message go? It's in a new file called CURTIME because the > sent the *output* there. Prove it:

type curtime ENTER

There it is:

```
Current time is 10:21:00.03
Enter new time:
```

The same thing can work with **date:**

date < enter > curdate ENTER

Now a file named CURDATE contains the date (and the update question).

Try this again with **time**, using >> instead of >:

time < enter >> curtime ENTER

Look at the CURTIME file. It now has two time entries:

```
Current time is 10:21:00.03
Current time is 10:21:58.85
```

The double greater-than sign redirects output, just like >, but it *adds to the end* of the specified file. Try it again, and see that the file contains three entries.

Putting the current date and time in a file really does have useful applications.

Sending Date and Time to the Printer

Date and time can be sent to a printer instead of a file. Turn on your printer, and try these commands:

date < enter > prn ENTER
time < enter > prn ENTER

The date and time were printed on paper. Consider putting these two commands into a batch file called TSTAMP.BAT. Then type **tstamp** whenever you want to "time stamp" a printout. (Much more about batch files in Part 4.)

The Standard Device

DOS 2.x users will notice a funny thing when they redirect the I/O with the **date** and **time** commands.

Whether or not a message is redirected depends on how the program is written. In this case, the programmer of COMMAND.COM used *standard console output* for the first message and *direct console output* for the second. Standard console output sends the message to the screen, or wherever output has been redirected. Direct console output sends a message to the screen, *even* if you've redirected output elsewhere.

The same applies to input. The programmer can request input from the *standard device* (allowing redirection) or directly from a specific device, such as the keyboard.

Redirecting

We saw how powerful **edlin** can be as a file editor. With redirection of I/O we can make it do some rather sophisticated operations automatically.

First, let's create a file to edit:

```
copy con testfile
this is line 1    ENTER
this is line 2    ENTER
this is line 3    ENTER
this is line 4    ENTER
this is line 5    ENTER
CTRL  Z   ENTER
```

Now make a file that contains "key-presses" recognized by **edlin**:

```
copy con keyfile    ENTER
3d   ENTER
1d   ENTER
e   ENTER
CTRL  Z  ENTER
```

KEYFILE contains the **edlin** commands that delete lines 3 and 1 of the file (yet to be specified), and then exit it. Try it:

```
edlin testfile < keyfile    ENTER
```

DOS called up **edlin**, loaded TESTFILE, deleted lines 3 and 1, and exited. All the keystrokes were executed automatically! Enter **type testfile** to see what happened.

```
this is line 2
this is line 4
this is line 5
```

Sure enough, lines 1 and 3 are gone.

NUL Output

To eliminate the screen reporting of the **edlin** actions, let's redirect that output to NUL.

Try the **edlin** example again with TESTFILE. Since **edlin** created a safety backup, we can restore the original with:

copy testfile.bak testfile ENTER

Use **type testfile** to check for all 5 lines. Then:

edlin testfile < keyfile > nul ENTER

This time, no messages are displayed since they were sent to the never-never land called NUL. Enter **type testfile**, and you'll see that lines 1 and 3 are gone, as before.

Redirecting the Directory

We can redirect a directory listing to the printer instead of the screen. Type:

dir > prn ENTER

We can even send it to a file. Try:

dir > dirfile.txt ENTER

Now there is a copy of the directory in DIRFILE.TXT.

Type:

chkdsk >> dirfile.txt ENTER

This appended the information from a **chkdsk** to the DIRFILE.TXT file.

Now let's type DIRFILE.TXT, but instead of sending it to the screen, we can redirect it to the printer. Try:

type dirfile.txt > prn ENTER

Now we have a printout of DIRFILE.TXT.

Delete ENTER, CURDATE, CURTIME, TESTFILE, TESTFILE.BAK, KEYFILE, and DIRFILE.TXT if they are still on your *system* disk.

Chapter 11 Summary

To redirect input, create a batch file that contains the needed keystrokes and use < in your commands. **edlin myfile < autokeys** loads MYFILE and executes the commands in AUTOKEYS.

To redirect output, use the > symbol. **chkdsk > statfile** executes CHKDSK.COM and sends all screen output to STATFILE.

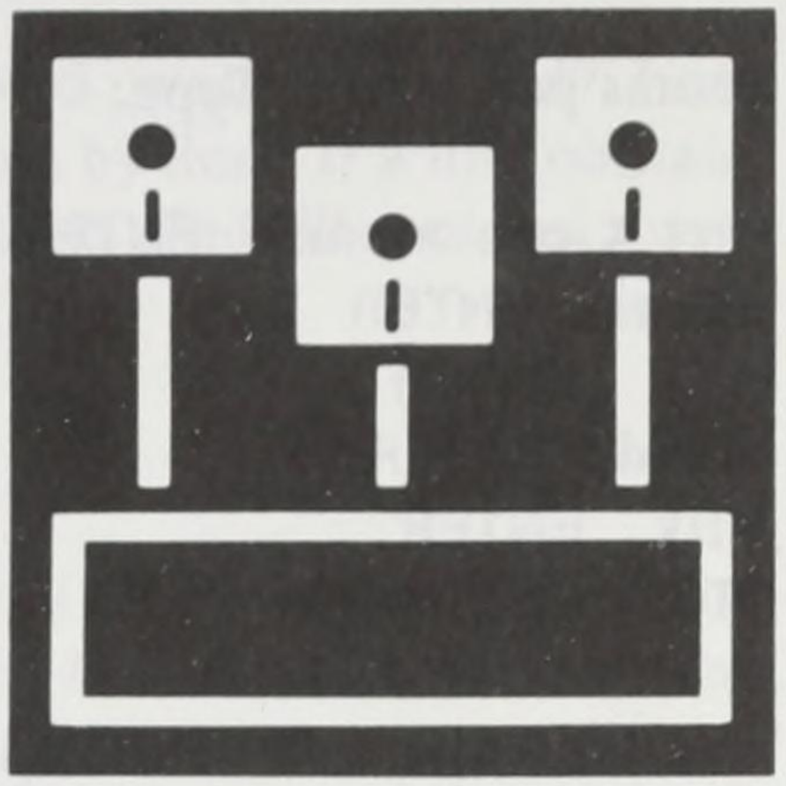
When redirecting output, use >> if you want the output to be added to the end of the file you specify.

Output may also be redirected to devices. Use **> prn** to send output to the printer. Use **> nul** to suppress the output.

Some programs and commands lead themselves to redirecting I/O better than others. Only output that goes to the *standard output device* and input that comes from the *standard input device* may be redirected.

CHAPTER 12

Understanding Filters and Pipes



A *filter* is a command that accepts data, changes it in some way, then outputs it. Three filter commands are included on the DOS diskette. They are **sort**, **find**, and **more**.

How a Filter Works

First, let's look at something that behaves like a filter. Type:

```
copy con con  ENTER
butter  ENTER
milk  ENTER
bread  ENTER
jelly  ENTER
CTRL  Z  ENTER
```

After the CTRL Z, MS-DOS redisplays what you typed:

```
butter
milk
bread
jelly
        1 File(s) copied
```

Input came *from* the console and output went *to* the console. The data was not altered in any way before it was output.

Seems pointless? Type:

sort < con > con ENTER
butter ENTER
milk ENTER
bread ENTER
jelly ENTER
CTRL Z ENTER

This time, when DOS redisplays what was typed, it's in *alphabetical* order:

```
bread
butter
jelly
milk
```

All filters assume the console for input and output, unless told otherwise. You could have just typed **sort**. Try it:

sort ENTER

Only a flashing cursor, since SORT.EXE is waiting for input from the console. Type **butter, milk, bread**, and **jelly** again, and press CTRL Z and ENTER.

The FIND and MORE Filters

The other two filters, **find** and **more**, work like **sort**, but they process one line at a time. Try this:

find "bread" ENTER

The system waits for your input. Type:

butter ENTER
milk ENTER
bread ENTER

jelly ENTER
CTRL Z ENTER

Did you notice that after you typed **bread**, `bread` was redisplayed? Look at the screen. The **find** filter accepts input, line by line. If a line contains the search string, it outputs the line. In this case, **find** filtered out everything but `bread`. Now try the /**v** option:

find /v "bread" ENTER

The /**v** switch allows each line *not containing the search string* to pass from input to output. Type the four words, and press CTRL Z ENTER. This time, each line is echoed *except* `bread`.

The **more** filter also processes one line at a time. Each line is output immediately. After every 23rd line, `--More--` is displayed, and the system waits for you to press a key. **more** is designed for use with programs and commands that supply a lot of screen output. It allows you to read the screen before information rolls off the top, similar to a pause. To watch **more** in action, type:

more ENTER

The system stops to wait for input from the console. Type:

xxx ENTER

and `xxx` repeats. Type:

yyy ENTER

Same thing. **more** passes input directly to output. In this case, the console provides both the input and output, so you get a "double vision" effect. Put in 21 additional "dummy" lines. Each one passes through and repeats just as lines `xxx` and `yyy` did. Then `--More--` is displayed. Press a key, and you can continue for 23 more lines. Press CTRL Z ENTER when you've seen enough.

What's the Point?

There is little practical reason to use **sort, find**, and **more** like this. We used the console for input and output only as a convenient way of seeing what they do.

With **more** fresh in our minds, let's use it as intended. To see the effect, we'll need an ASCII file that is over 23 lines long.

more < all3 ENTER

more used this way is like the **type** command, but scrolling stops for viewing after each screenful.

Notice the format. The < tells DOS to give **more** its input from the ALL3 file, rather than the console. You are redirecting input.

Piping

The output of one command can be "piped" to the input of another. Piping may be used to connect commands with filters. Try these examples:

dir | sort ENTER

The | character connects **dir** to **sort**. DOS "saves up" all the information that would ordinarily be displayed by **dir** and "feeds" it to **sort**. SORT.EXE puts the directory in alphabetical order and displays it.

dir | sort /r/+15 ENTER

sorts the directory in reverse order, based on the 15th column. Since the *file sizes* start in the 15th column, the result is a directory with the biggest files listed first. Use this one when you need to decide what files to erase for more room on the diskette.

dir | find "COM" ENTER

lists the COM files.

dir | find /v "COM" ENTER

displays the directory, *excluding* the COM files.

dir | find "(Type in today's date.)" ENTER

Only files created today are listed.

chkdsk | find "bytes available" ENTER

DOS runs CHKDSK.COM, but only the "bytes available" line makes it through the filter.

Multiple Filters

With a little thought, you can use more than one pipe and filter in a command to find or do exactly what you want. Consider this:

dir | find /v "COM" | find " 9-16-99"

DOS first locates all the directory entries. It then filters out the COM files. Finally it displays the remaining directory entries dated 9-16-99.

Take it a few steps further:

dir | find /v "COM" | find " 9-16-99" | sort | more

All directory entries for 9-16-99, except COM files, are sorted by file name. Then **more** displays the result, stopping every 23rd line.

It gets detailed, doesn't it? The best approach is to write and save your own batch files for the **sort** and **find** commands you use most often. You'll learn how to work with batch files later in this book.

FIND with Multiple Files

The **find** filter has another option. It can accept a list of files for searching:

find "delete" file2 file3 ENTER

```
---------- file2
```

and

```
---------- file3
```

are displayed, with all the lines containing "delete" listed below each heading.

Wildcard file names are *not* permitted with the **find** command, but you can work around the limitation with **for**. Suppose you want to find *all* the lines containing "MS-DOS" in all your TXT files. Here's how:

for %f in (*.txt) do find "MS-DOS" %f ENTER

%PIPE1 and %PIPE2

%PIPE1 and %PIPE2 serve as temporary storage files while piping. DOS normally erases or renames them after the operation is complete.

If a piping or redirection command fails to execute properly, %PIPE1 and %PIPE2 may still be there. Delete them.

Chapter 12 Summary

SORT.EXE, FIND.EXE, and MORE.EXE are the *filters* provided with MS-DOS. They accept input from the console and send output to the console, unless you specify otherwise.

sort accepts data from a file or command and puts it in order. **sort < names.txt > prn** sends data from NAMES.TXT to the printer in alphabetical order. **dir *.com | sort | more** displays a directory of all COM files in alphabetical order, stopping after every 23rd line.

sort has two optional switches. **/r** causes a reverse-order sort. /+ causes sorting to be done on a character other than the first character of each line. **sort /r/+24 < agefile.txt > agefile.sor** sorts AGEFILE.TXT in descending sequence and puts the result in AGEFILE.SOR. The data at the 24th column of each line determines the order.

The **find** filter may be used to search single or multiple files. **find "CA" <address.txt** searches for "CA" in ADDRESS.TXT. **find "Elvis"**

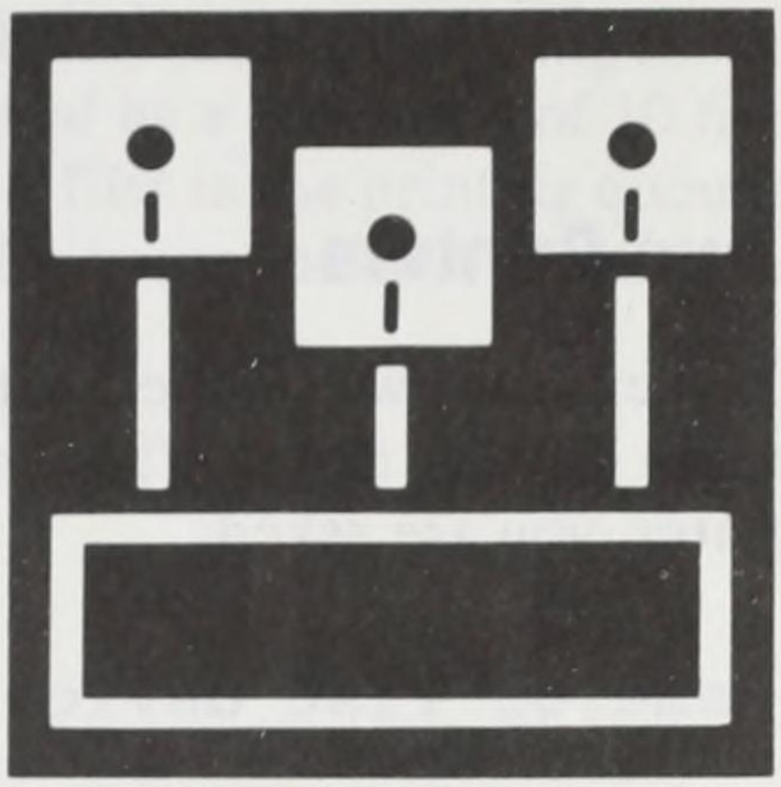
albums.txt singles.txt > > playlist.txt finds all lines containing "Elvis" in ALBUMS.TXT and SINGLES.TXT and adds them to PLAYLIST.TXT.

find has three optional switches. /**v** outputs all lines *not* containing the search string. /**c** outputs a count of the lines containing a match. /**n** displays a line number next to each line output. (See Volume 1 for an explanation of /**c** and /**n**.)

more displays a file and halts the scrolling after every 23rd line. Press a key, and the next 23 lines are displayed. **more < todo.txt** displays the contents of TODO.TXT 23 lines at a time. Output from commands may be piped to **more**. For example, **dir | more** is similar to **dir/p**.

The | (pipe) character is used to pipe the output of one command to the input of another. **dir | sort** pipes the output of a directory listing through the **sort** filter, displaying a directory in alphabetical order.

CHAPTER 13



The PRINT Command

We have studied several ways to control the flow of data. Now it's time to take a hard look at PRINT.COM.

The **print** command allows us to send data files directly to a printer. While they're being printed, you are free to use the computer for other work. It's almost like having two computers in one!

Suppose you are entering data from payroll time cards. The marketing director walks in. He wants the latest price list and sales forecast. You return to DOS and issue a command:

print pricelst.txt forecast.txt ENTER

As soon as PRICELST.TXT begins printing, you restart the payroll program and continue entering time cards. FORECAST.TXT is "in the queue" and will be printed after the price list.

The **print** command has several important options:

- To add a file to a print queue which already has a file in it, type **print** and put **/p** after the file name. Up to 10 files may be scheduled for printing at once.

- To cancel a file waiting in a queue, type **print** and put **/c** after the file name.

- To see what is in the print queue, simply type **print**.
- To terminate printing, type **print /t**.

Two Printers

If there are two printers attached to the system, you may be able to use them both at once. The first time you use **print** after booting the computer, you are asked:

```
Name of list device [PRN:]
```

If you want all printing to default to printer #1, just press ENTER. If there's a second printer, enter **lpt2:**. The **print** command will then send all its output to printer #2. Printer #1 remains available for other work.

The ASCII Catch

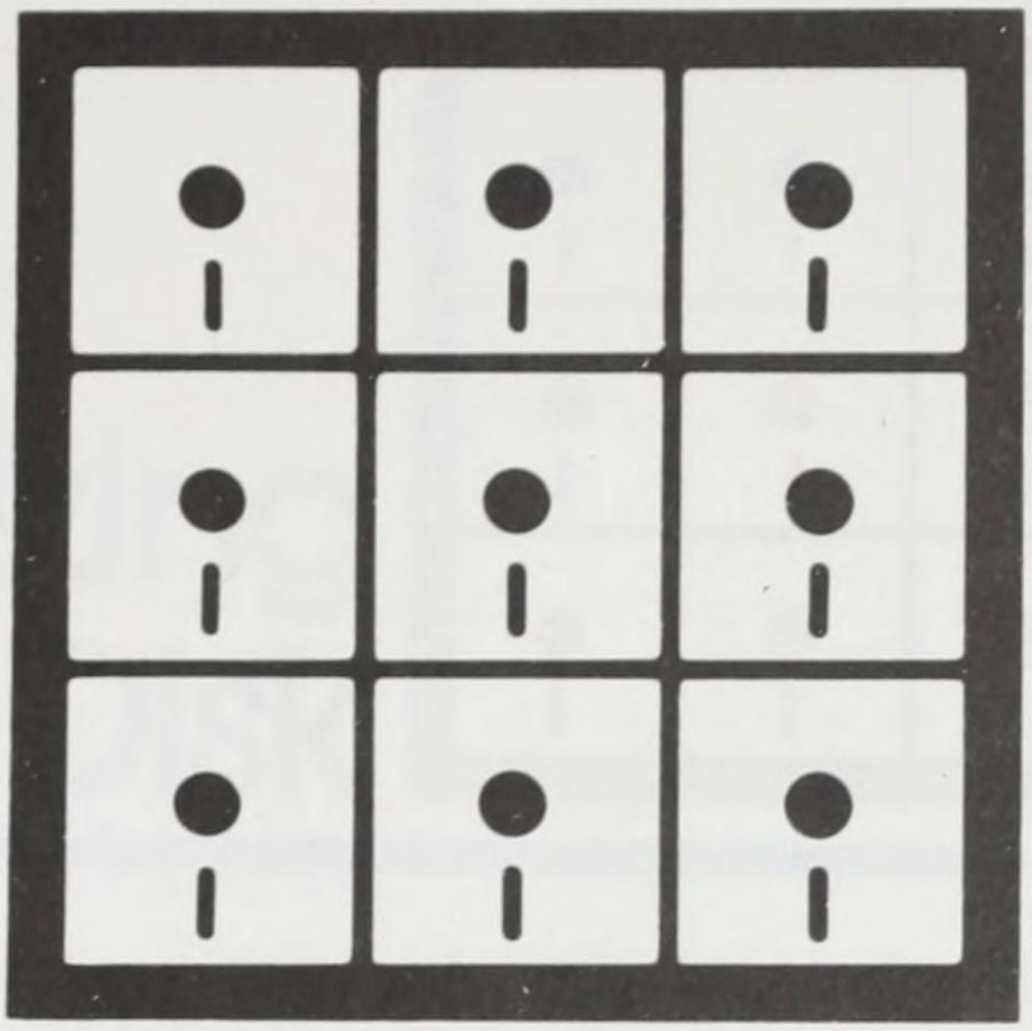
To use **print** effectively, planning is necessary. To be printed, the files must be in ASCII format. Unfortunately, most application programs store their files using binary format to save disk space. There are 3 ways to solve the problem:

- Look for an "ASCII save" option in the program. Some word processors let you put **/a** or **,a** after the file name, causing it to be saved in ASCII. Others let you "unformat."
- Some programs allow you to select the output device. You may be able to respond with a disk file name that **print** can use.
- Some programs have the option to display printouts on the screen. Try using > to redirect that output to a disk file, then print that file. (See Chapter 12.)

Chapter 13 Summary

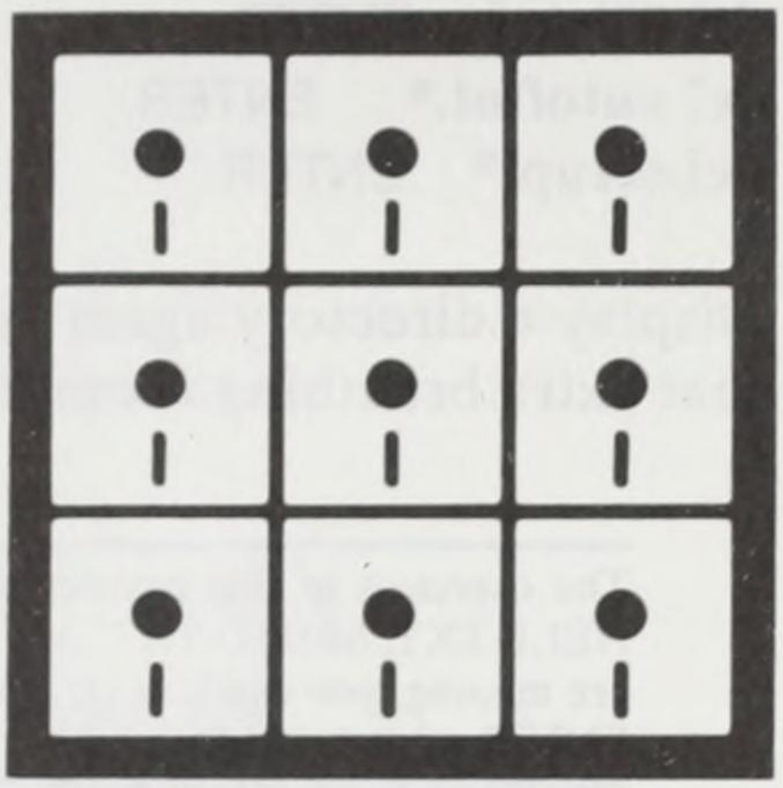
The **print** command allows you to print ASCII files while performing other non-printing tasks. Simply enter **print** followed by a maximum of 10 file names. Use /**p** and /**c** to add or cancel individual files in the printing queue. **print** by itself displays on the screen what's in the queue. **print**/**t** terminates the printing.

PART 3



The path to organization

CHAPTER 14



Organizing the Disk

The directory listing is already so long it's hard to make sense of. Almost like a file drawer without dividers. DOS has the perfect solution.

Breathing Room

Insert the *system* diskette in Drive A and set the system prompt to A>
Type:

dir /p ENTER

(Remember the /**p** option "pauses" the directory listing so we can view a screenful at a time.) We want to delete and organize these files to make at least 150,000 bytes available on disk for the exercises which follow.

Readers who were able to make a 1.2M or 1.44M *system* diskette earlier needn't bother deleting any DOS files.

At this time we need to delete several files to free up additional diskette space. Here are some files you won't need for a while. Since you can re-copy them from the original MS-DOS *master* disk later, go ahead and delete those which are part of your DOS:

del mode.* ENTER
del link.* ENTER
del fdisk.* ENTER
del autofmt.* ENTER
del setup.* ENTER
del fmat2000.* ENTER
del basic?.* ENTER
del test.* ENTER
del mouse.* ENTER

Display a directory again and look at the free bytes. Feels good to have that extra breathing room, doesn't it?

The exercises in this chapter use files we created earlier. Verify that LETTER.TXT, HELP.TXT, MEMO.TXT, ALL3, FILE1, FILE2, and FILE3 are on your diskette. If any are missing, you can quickly make "dummies" that will suffice. Example: **copy con file1** ENTER **This is FILE1** ENTER CTRL Z ENTER

Making a Subdirectory

To organize the files on this diskette, let's put our practice files in a section of their own. We'll create a special subdirectory area on the disk and name it EXAMPLES. Type:

mkdir examples ENTER

Display the directory, and see this new entry:

```
EXAMPLES    <DIR>     9-15-xx  11:07a
```

<DIR> makes it stand out from the other files because it's a special file. It is a *subdirectory*, a directory within a directory. The **mkdir** command MaKes a DIRectory. Right now, the EXAMPLES subdirectory is empty, so let's copy all of our practice files into it.

copy *.txt examples ENTER

That command copied MEMO.TXT, HELP.TXT, and LETTER.TXT into the EXAMPLES subdirectory. Next:

copy file? examples ENTER
copy all3 examples ENTER

Displaying a Subdirectory

To see what we've done, type:

dir examples ENTER

The display shows:

```
Volume in drive A has no label
Directory of A:\EXAMPLES

.            <DIR>      9-15-xx  11:07a
..           <DIR>      9-15-xx  11:07a
MEMO     TXT     116    9-15-xx   8:32a
LETTER   TXT     754    9-15-xx   9:35a
ALL3             804    9-15-xx   9:50a
HELP     TXT     569    9-15-xx  10:03a
FILE1            138    9-15-xx  11:22a
FILE2            197    9-15-xx  11:25a
FILE3            236    9-15-xx  11:31a
      9 Files(s)     xxxxxx bytes free
```

Copies of all the practice files now in EXAMPLES are shown. Try a regular directory listing:

dir ENTER

The practice files are still there. Copying them into a subdirectory didn't "move" them, it duplicated them. Let's delete the original copies of the practice files.

del file? ENTER
del all3 ENTER
del *.txt ENTER

Use **dir** to see that they are gone. Once again, take a directory of the subdirectory named EXAMPLES:

dir examples ENTER

The copies are still there.

Backslash and Pathnames

Try this:

type letter.txt ENTER

DOS responds with `File not found`. That's because the LETTER.TXT file is no longer in the main directory. You have to tell DOS where to find it. Here's how:

type examples\letter.txt ENTER

> On the Tandy 1000s, the backslash is on the 7 key on the numeric keypad. On the 3000s/4000s, it's on the upper-right-hand side next to the <-- key.

In DOS, a backslash specifies a *path*. EXAMPLES\LETTER.TXT is a *pathname*. A pathname is like a file name, but it also tells what *subdirectory* to look in.

Try it on several other files:

type examples\file1 ENTER
type examples\all3 ENTER

Changing the Directory

Typing **examples** for each file name in the EXAMPLES subdirectory is a bit inconvenient. How about an easier way? Type:

chdir examples ENTER

The **chdir** command CHanges the *current* DIRectory. What does that mean? You have moved your point of perspective and are now viewing the diskette from the EXAMPLES subdirectory. To check the view, type **dir** ENTER.

Once "in" a subdirectory, you can work with any of its files without having to specify a pathname. Prove it yourself:

type memo.txt ENTER
type help.txt ENTER
type all3 ENTER

It's as if the files in the EXAMPLES subdirectory were the only ones on the disk.

Now type:

**dir ** ENTER

See a list of all the files *other than* those in the EXAMPLES subdirectory.

Using the backslash by itself as a pathname requests a view of the *main* directory, called the *root directory*. It is the root because all subdirectories you create "grow" out of it, like a tree and its branches. In fact, the whole concept of subdirectories is called *tree-structured*. You'll see why as we continue.

How do we get out of the EXAMPLES subdirectory and back to the root? Easy:

**chdir ** ENTER

Go ahead and do it. Use **dir** to see where you are.

Lost Horizon?

So far we have two directories, the root directory and a subdirectory named EXAMPLES. Typing the **chdir** command without a pathname always reports where you are. Try it:

chdir ENTER

The response is `A:\`. You are in the root directory of the disk in Drive A.

Now pop back into EXAMPLES. Type:

chdir examples ENTER

Display the directory if you wish. Now:

chdir ENTER

DOS responds with `A:\EXAMPLES`. You are back in the EXAMPLES subdirectory of the disk in Drive A.

Making More Subdirectories

Let's make more subdirectories to store the other files and simplify the root directory. First type:

**chdir ** ENTER

to return to the root. One subdirectory can nicely hold all the MS-DOS COM and EXE files (with the exception of COMMAND.COM which *must* remain in the root). Call this new subdirectory DOSCMDS. Type:

md doscmds ENTER

md is an abbreviation of **mkdir**. Now we'll move all the COM files into the DOSCMDS directory. Since we don't want to move COMMAND.COM, use this trick. First:

rename command.com command.xxx ENTER

Now type:

copy *.com doscmds ENTER

That put a copy of every COM file in the DOSCMDS directory, except COMMAND.COM, which we deliberately renamed. Next delete the COM originals from the root directory to free up that space. Type:

del *.com ENTER

Finally, rename COMMAND.XXX back to COMMAND.COM:

rename command.xxx *.com ENTER

Do you see how the trick works? When you want to copy most but not all files with a particular extension, rename the files you don't want to copy. Then copy the rest using * and the extension. Finally, use **rename** to restore the names of the files not copied. The same idea works with **delete** and other commands requiring file names as parameters.

Let's also move the EXE files into the DOSCMDS directory. Use these commands:

copy *.exe doscmds ENTER
del *.exe ENTER

And display a directory of the DOSCMDS subdirectory:

dir doscmds ENTER

You should see all the COM and EXE files (except COMMAND.COM). Display the root directory to see what is left:

dir ENTER

On the 1000 TX using DOS version 3.x, it looks something like this:

```
Volume in drive A has no label
Directory of A:\

COMMAND  COM     23612  6-17-xx  1:00p
ANSI     SYS      4963  6-17-xx  1:00p
DOSCMDS      <DIR>      6-17-xx 11:30a
EXAMPLES     <DIR>      6-17-xx 11:08a
KEYCNVRT SYS       202  6-17-xx  1:00p
LPDRVR   SYS      3452  6-17-xx  1:00p
SPOOLER  SYS      3982  6-17-xx  1:00p
VDISK    SYS      2753  6-17-xx  1:00p
DRIVER   SYS      1102  6-17-xx  1:00p
MLPART   SYS      2535  6-17-xx  1:00p
      10 File(s)     xxxxxx bytes free
```

You've probably noticed that DOS doesn't keep the directory in any particular order. Why, for example, aren't EXAMPLES and DOSCMDS the last entries? Answer: New files in the directory take the place of the files previously deleted. Only after all previously-deleted positions are taken do new entries go at the end.

The final step in this directory organization is to move the remaining files from the root to another subdirectory. Since these are "miscellaneous" files that come with DOS, we'll name the new subdirectory DOSMISC. Type:

md dosmisc ENTER

and copy everything remaining:

copy *.* dosmisc ENTER

EXAMPLES <DIR>, DOSCMDS <DIR>, and DOSMISC <DIR> were not copied. They are *directories*, not ordinary files, so **copy** ignores them. Only the files in the *current directory* (in this case, the root) are copied.

Delete all remaining files from the root directory:

del *.* ENTER

DOS asks `Are you sure (Y/N)?` You are asked to confirm whenever you use *.* with **delete**. Knowing that all files have been copied to subdirectories, you can press Y. Now read the root directory.

One little detail remains. DOS needs to be able to find COMMAND.COM. If it is not in the root directory, DOS can't find it. Since you are currently in the root, use this command to copy it back:

copy dosmisc\command.com ENTER

Study this **copy** command. The *source* is DOSMISC\COMMAND.COM, meaning COMMAND.COM in the DOSMISC subdirectory. The *target* is omitted. When the *target*, or destination, is omitted, DOS assumes the same filename, in the *current* directory, on the *current* drive.

There are now two COMMAND.COM files on the disk, distinguished by their pathnames. The one in the root is COMMAND.COM. The other is DOSMISC\COMMAND.COM. Delete COMMAND.COM from the DOSMISC subdirectory by using the pathname:

del dosmisc\command.com ENTER

We deliberately avoided using the "rename" trick this time to show you another way to accomplish the same result.

The View from the Root

We did a lot of reorganizing without looking at our work. Since we are still in the root directory, let's see what's here. Type:

dir ENTER

Ah! The ultimate in organization (at our level of sophistication). All the files we started with are on the disk, but at the root we only see the mandatory COMMAND.COM, and the names of the subdirectories that hold the files:

```
Volume in drive A has no label
Directory of A:\

COMMAND  COM    23612  6-17-xx    1:00p
DOSMISC     <DIR>      9-15-xx   11:24a
DOSCMDS     <DIR>      6-17-xx   11:30a
EXAMPLES    <DIR>      6-17-xx   11:08a
      4 File(s)     xxxxxx bytes free
```

Try **dir *.*, dir **, and **dir *.***. They all give the same display. **dir *.*** requests all files in the current directory, which happens to be the root. **dir ** is an explicit request for a directory of the root. **dir *.*** is an even more specific request for a list of all files in the root. (For this purpose, subdirectory names are considered to be files.)

Now try:

dir *. ENTER

Only the names of the subdirectories are listed because the directory names, DOSMISC, EXAMPLES, and DOSCMDS have no extensions. It is legal to give a directory name an extension, but it's not common.

Wildcards and Subdirectories

Just for practice, try:

dir doscmds*.* ENTER

dir examples*.txt ENTER

dir dosmisc\ansi.sys ENTER

Exploring the Subdirectories

Now let's go into one of the subdirectories. Just as **md** is short for **mkdir, cd** is an abbreviation for the **chdir** command. Type:

cd doscmds ENTER

Now use **dir** or **dir *.*** and see a listing of the files in the DOSCMDS subdirectory. How about a directory of EXAMPLES. Try:

dir examples ENTER

It didn't work! You are on the DOSCMDS branch, and EXAMPLES is not one of its subdirectories. It is necessary to search via the root:

dir \examples ENTER

This way, the directory of the EXAMPLES subdirectory is listed, but you remain in DOSCMDS. Verify your location by typing **cd** ENTER. `A:\DOSCMDS` is displayed.

When looking for a file or directory, DOS starts wherever you are, unless you tell it otherwise. The \ is what told it otherwise. Backslash as the first character in a pathname means "start from the root."

You can use a little longer pathname if you want to look at specific files in the EXAMPLES directory, while still in the DOSCMDS directory. Here's an example:

dir \examples*.txt ENTER

Now go into the EXAMPLES directory itself. Try:

cd examples ENTER

That command doesn't work. As with the **dir** command, you must tell DOS the path to follow. The correct command requires a backslash:

cd \examples ENTER

Display the directory. Are you there? Good.

That's enough for now. Let's end this very important chapter by returning to the root.

**cd ** ENTER

No Old, Bold Pilots

At this point in your work, it's a *very* good idea to make a safety backup of your reorganized *system* diskette. You have a lot of time invested, and we have a lot more changes to make, complete with numerous opportunities to make bad mistakes.

Write protect this diskette, and label it "Thru Chap 14." Since **diskcopy** is now in a subdirectory, **cd doscmds** then **diskcopy**. Use the fresh copy as we proceed to the next chapter.

There are old pilots, and there are bold pilots. There are no old, bold pilots. Make safety backups often.

Chapter 14 Summary

DOS lets you organize files on a disk by creating *subdirectories*. A subdirectory is a directory within a directory. Subdirectories "branch out" from the *root directory*, hence the term *tree-structured directories.*

The **mkdir**, or **md**, command makes a subdirectory. The rules for naming directories are the same as those for naming files. You may use up to 8 characters plus an optional extension. **mkdir myfiles**, or **md myfiles**, makes a directory called MYFILES.

Pathnames tell DOS how to find the file you want. A pathname most often consists of a subdirectory name followed by a backslash and a file name. **myfiles\calendar.txt** specifies the CALENDAR.TXT file in the MYFILES subdirectory.

Backslash (\) by itself is a special pathname. It indicates the root directory. **dir** \ displays the root directory.

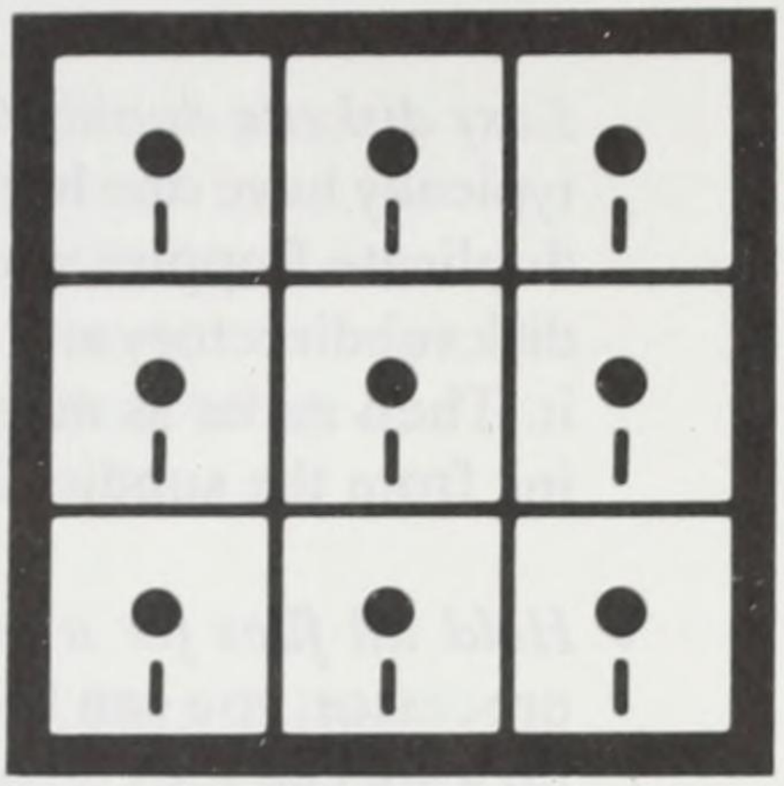
Pathnames always start from your current position in the "tree," unless you tell DOS to start from the root. **dir \myfiles** tells DOS to start at the root, search for a subdirectory called MYFILES, and then display its contents.

chdir, or **cd**, used by itself displays the name of the current directory. Used with a pathname, it moves you into another directory. **chdir \myfiles**, or **cd \myfiles**, makes MYFILES the current directory. Once in a directory, you may refer to the files it contains without specifying a path.

COMMAND.COM must be handled with special care when organizing directories. MS-DOS expects to find it in the root directory.

Wild cards can be used to move files between directories. **copy *.com \doscmds** copies the COM files in the current directory to a directory named DOSCMDS. An easy way to exclude a file from a wild card move is to rename that file's extension before the copy, then change it back when the copy is completed.

CHAPTER 15



Paths and Shortcuts

In the last chapter, we saw that establishing subdirectories is a good way to organize files on a disk. *Tree-structured directories* are especially important on a hard drive, where there may be hundreds or even thousands of files. Here are a few ways to cluster files.

- *Group files of a particular type:* We did this by putting the MS-DOS COM and EXE files in one subdirectory and our practice files in another.

- *Make an area for files "owned" by a particular employee, family member, company, or division:* You don't have to decide how much disk space to allocate. The subdirectories automatically "expand" and "contract" according to the need.

- *A quick backup area:* Make one subdirectory that serves as a backup area for another subdirectory. There's no problem with duplicate file names as long as they are in separate directories. Just **copy *.*** from one subdirectory to the other. This is faster and more convenient than copying to another diskette. Some users do a subdirectory backup every few hours and a complete diskette backup at the end of the day.

- *Multiple diskettes on a hard drive:* Suppose you have a set of 20 diskettes containing a collection of programs. You want to put all the programs on a hard drive but don't want to lose track of

what programs came from which diskette. Just create separate subdirectories for each diskette.

- *Easy diskette duplication from a hard drive:* Hard drive systems typically have one hard drive plus one floppy disk drive. You can duplicate floppies with **diskcopy**, but it's faster to create a hard disk subdirectory and copy all files from the original diskette into it. Then make as many floppy disk copies as you want by copying from the subdirectory to new diskettes.

- *Hold all files for a particular application:* If you have a word processor, you can keep all word processing files separate or set up a special area to hold all *DeskMate* files.

- *Overcome file limitations:* The root directory on the Tandy 1000's *diskette* is limited to 112 files, including the hidden ones. On a *hard drive* it is usually limited to 512 files. With subdirectories you can have as many files as you want. DOS can usually find a file faster when searching a small subdirectory than when searching a large root directory.

A Subdirectory for Backup

Since we'll be doing some practice reorganization of the EXAMPLES subdirectory, we can try one of these ideas right now. At the end of the session we want to put it back the way it was, so how about a quick backup?

With the *system* diskette in Drive A, and the `A>` prompt showing, type **cd** ENTER. You should see `A:\`. If not, change to the root directory with:

**cd ** ENTER

Make a new subdirectory called EXAMPLES.BAK:

md examples.bak ENTER

Copy all files from the EXAMPLES subdirectory to the EXAMPLES.BAK subdirectory.

copy examples*.* examples.bak*.* ENTER

or:

copy examples examples.bak ENTER

Each pathname is listed as the files are copied:

```
EXAMPLES\MEMO.TXT
EXAMPLES\LETTER.TXT
   .
   .
   .
   etc.
```

Compare the EXAMPLES and EXAMPLES.BAK subdirectories. They should be identical.

Dot and Dot Dot

No, this isn't a new version of the Morse Code. They are special symbols used to refer to files in the directory and subdirectory. You may have noticed them in the last chapter when we displayed the EXAMPLES subdirectory.

Go into the EXAMPLES subdirectory:

cd \examples

```
Directory of  A:\EXAMPLES

.        <DIR>      9-15-xx  11:07a
..       <DIR>      9-15-xx  11:07a
```

Both of these entries have a `<DIR>` extension, which means they refer to directories, just as `EXAMPLES <DIR>` did in the root, but dots are shown in the left column. To see what they do, try this:

dir . ENTER

You see the EXAMPLES directory again. Now try:

dir .. ENTER

The root directory is displayed. In this case, **dir ..** is the same as **dir **.

A single dot means *current directory*. A double dot means *parent directory*. The parent directory is the directory to which the current directory is a subdirectory. The dots are called *anonymous directory names* because they can be used in place of the actual directory names.

In this case, the parent of the EXAMPLES subdirectory is the root. Two dots stand for root when there are 2 levels of directories. Subdirectories more than 2 levels deep are our next subject.

Multi-Level Directories

Suppose we want to organize the EXAMPLES subdirectory so files are classified by the chapter (of this book) in which they were created. Let's do that now.

First check the current directory:

cd ENTER

It should be `A:\EXAMPLES`. Now make 6 subdirectories. (Remember, you can use F3 and BACKSPACE after the first command to simplify typing.)

md chap5 ENTER
md chap6 ENTER
md chap7 ENTER
md chap8 ENTER
md chap9 ENTER
md chap10 ENTER

Use **dir** to check your work.

MEMO.TXT was created in Chapter 6, so move it into the CHAP6 subdirectory:

copy memo.txt chap6 ENTER
del memo.txt ENTER

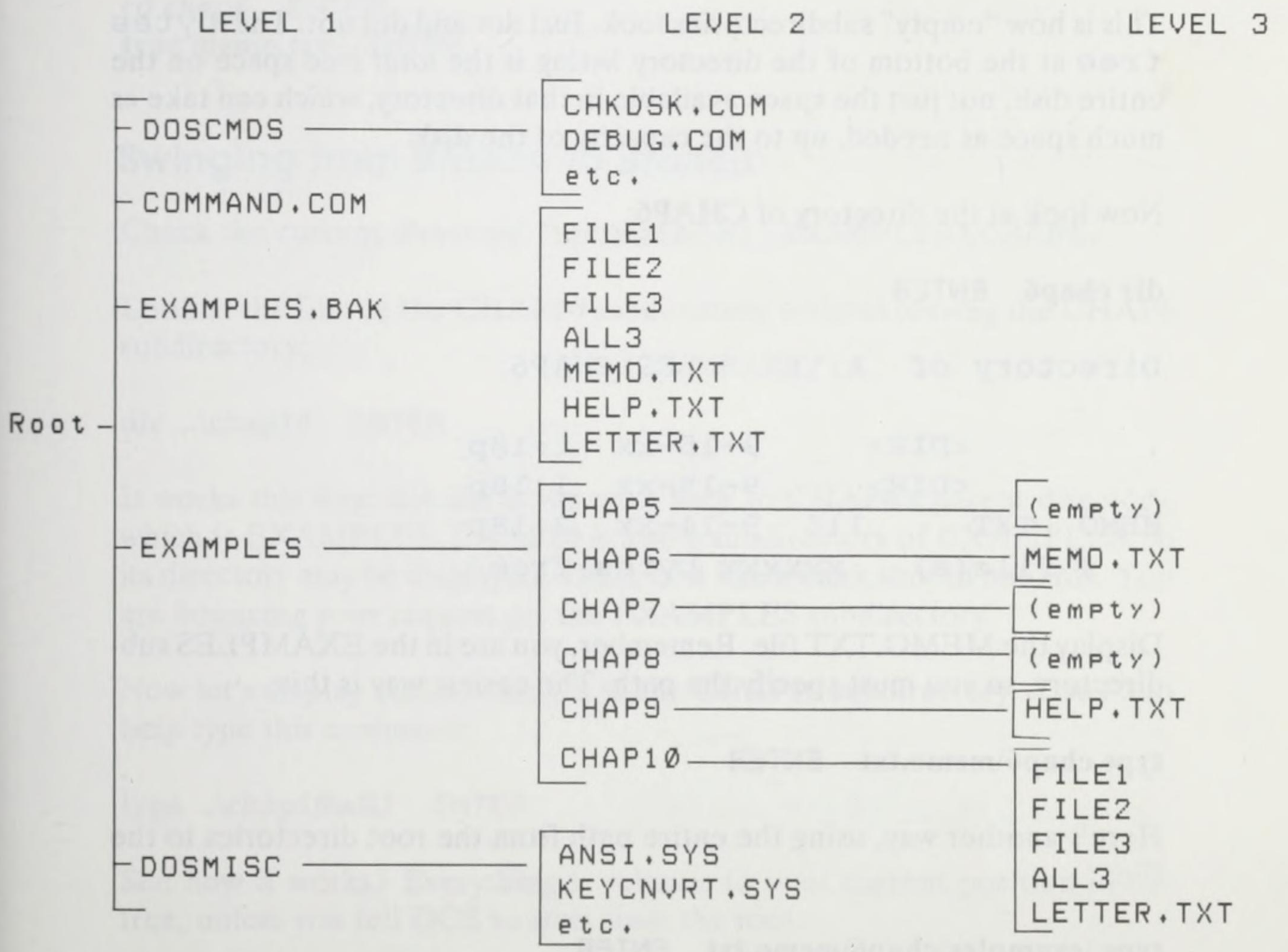
HELP.TXT was created in Chapter 9. Put it in the CHAP9 subdirectory:

copy help.txt chap9 ENTER
del help.txt ENTER

The other files were created in Chapter 10:

copy *.* chap10 ENTER
del *.* ENTER Y ENTER

The "tree" has 3 levels, and looks something like this:



Turn the book sideways and squint. Now do you see the tree?

Moving Around the Tree

Display a directory listing of the CHAP5 subdirectory:

dir chap5 ENTER

```
Directory of  A:\EXAMPLES\CHAP5

.          <DIR>      9-15-xx  1:18p
..         <DIR>      9-15-xx  1:18p
   2 File(s)    xxxxxx bytes free
```

This is how "empty" subdirectories look. Just dot and dot dot. The `bytes free` at the bottom of the directory listing is the *total* free space on the entire disk, not just the space available in that directory, which can take as much space as needed, up to the capacity of the disk.

Now look at the directory of CHAP6:

dir chap6 ENTER

```
Directory of  A:\EXAMPLES\CHAP6

.          <DIR>      9-15-xx  1:18p
..         <DIR>      9-15-xx  1:18p
MEMO  TXT      116  9-14-xx  2:18p
   3 File(s)    xxxxxx bytes free
```

Display the MEMO.TXT file. Remember, you are in the EXAMPLES subdirectory, so you must specify the path. The easiest way is this:

type chap6\memo.txt ENTER

Here's another way, using the entire path from the root directories to the file:

type \examples\chap6\memo.txt ENTER

Do the same thing with the anonymous directory names:

type .\chap6\memo.txt ENTER

and:

type ..\examples\chap6\memo.txt ENTER

The first one says, "Start from the current directory, find the CHAP6 subdirectory, and type MEMO.TXT." The second one says, "Start from the parent of the current directory, find the CHAP6 subdirectory within the EXAMPLES subdirectory, and type MEMO.TXT."

Perhaps the most obvious way to display the file is to go right into the EXAMPLES\CHAP6 subdirectory.

cd chap6 ENTER
type memo.txt ENTER

Swinging from Branch to Branch

Check the current directory. It should be `A:\EXAMPLES\CHAP6`.

Look at the files in the CHAP10 subdirectory without leaving the CHAP6 subdirectory:

dir ..\chap10 ENTER

It works this way: dot dot is the path back to CHAP6's parent directory, which is EXAMPLES. CHAP10 is also a subdirectory of EXAMPLES, so its directory may be displayed. Think of it like a bank shot in billiards. You are bouncing your request off the EXAMPLES subdirectory.

Now let's display ALL3, which is in the CHAP10 subdirectory. Use F3 to help type this command:

type ..\chap10\all3 ENTER

See how it works? Everything is relative to your current position in the tree, unless you tell DOS to start from the root:

dir \examples\chap10 ENTER

type \examples\chap10\all3 ENTER

We want to see the DOSCMDS subdirectory without leaving the CHAP6 subdirectory. One way to get there is by following the branches. Look at the tree diagram, and you'll see this means going down two levels and up one.

dir ..\..\doscmds ENTER

DOS interprets this as, "Go to the parent directory of the current directory's parent and list the files in DOSCMDS."

Try this directory request for FORMAT.COM:

dir ..\..\doscmds\format.com ENTER

Just for practice, copy FORMAT.COM into the CHAP6 subdirectory:

copy ..\..\doscmds\format.com ENTER

The *target* directory doesn't need to be specified. Because you are already there, it is the default. Use **dir** to see what happened.

There it is! Delete FORMAT.COM from the current directory.

While staying in the CHAP6 subdirectory, copy FORMAT.COM into the EXAMPLES\CHAP7 subdirectory. Take your pick of two ways to do it. First by using anonymous directory names:

copy ..\..\doscmds\format.com ..\chap7 ENTER

Or second, by spelling out the paths from the root:

copy \doscmds\format.com \examples\chap7 ENTER

Move now into the CHAP7 subdirectory:

cd ..\chap7 ENTER

Use **dir**. Sure enough, FORMAT.COM is here.

Let's delete it, but via the root. You pick a way to get there:

cd .. ENTER
cd .. ENTER
or **cd ..\..** ENTER
or **cd ** ENTER

Then:

del examples\chap7\format.com ENTER

If you didn't get `Invalid directory` or `Files not found`, it worked!

Running Programs in Subdirectories

To run a program in a subdirectory, we must tell DOS where it is. Try **chkdsk** ENTER. It doesn't work because CHKDSK.COM isn't in the current directory.

cd \doscmds ENTER
chkdsk ENTER

and it works.

By the way, notice **chkdsk**'s new statistic. `Bytes in directories` refers to the "overhead" used for the subdirectories. The space used by the files they hold is included in the "user files" statistic, just below it.

The PATH Command

The **path** command is a convenient tool for working with subdirectories. At the `A>` prompt type:

path ENTER

The response is `No path`. Now type:

path \doscmds ENTER

Once again, type **path** ENTER and the response is:

```
PATH=\DOSCMDS
```

What have you done? Type:

**cd ** ENTER
chkdsk ENTER

Now, even though you are in the root directory, CHKDSK.COM works. The path command tells DOS where to look for a file if it is not found in the current directory. For more proof, **cd** to **\examples\chap10**. Execute CHKDSK.COM again:

chkdsk ENTER

It works! DOS looked first in the CHAP6 subdirectory. Not finding CHKDSK.COM there, it checked the current *path* setting, which told it to look in the DOSCMDS subdirectory.

Alternative Paths

More than one path can be specified with the **path** command. Try:

path \;\doscmds ENTER

Notice the semicolon separating the two paths. DOS understands this to mean, "First check the current directory, then the root directory, then look in the DOSCMDS directory."

Drive letters may also be included in a path specification. Look at this example:

path a:\;a:\doscmds;a:\dosmisc;b:

DOS first checks the current directory, then the root of Drive A, then DOSCMDS on Drive A, then DOSMISC on Drive A, and then the root directory of Drive B.

Finally, you must know how to cancel the path setting:

path ; ENTER

Now, when you request a program, DOS will again search only the current directory.

Subdirectories on Different Drives

To specify a file on a specific drive, put the drive letter and colon before the pathname. **c:\wp\myresume.txt** refers to a MYRESUME.TXT file in the WP subdirectory on Drive C.

DOS remembers the current directory for each of the system's drives. The current directory on a particular drive is called that drive's *home directory*. Knowing this, you can simplify **copy** commands. Use **cd** to set the current directory on each drive, then you can copy between those subdirectories by just specifying the file names.

The TREE Command

Despite subdirectories' obvious virtues, it's possible to forget where you stored a particular file or program. TREE.COM is a special program that displays the way you've organized the disk. To see how it works, type:

cd \doscmds ENTER
tree ENTER

The complete list of all subdirectories and paths is displayed. In most DOS versions, it's so long that a printout would be better. Put the printer online and try it again:

CTRL P or PRINT (causes output to go to the printer.)
tree ENTER
CTRL P or PRINT (stops output to the printer.)

A special **tree** command option makes it even more useful, especially if you are looking for a "lost" file. Type:

tree /f ENTER

The **/f** option gives the same listing as before, but adds all the file names in each subdirectory. It's a master directory of your disk, one which you may wish to print and hang nearby.

tree can also display the tree structure on another drive. To view Drive B you would use:

tree b: ENTER or **tree /f b:** ENTER

Removing Directories

It's easy to remove subdirectories when they are no longer needed. There is only one rule to remember. A subdirectory must be empty before it can be removed.

Since we no longer need EXAMPLES.BAK, we will now remove it. If for some reason, you wish to keep the file, make your own backup now on a separate disk.

Let's dismantle the CHAP5 through CHAP10 subdirectories. First, go to their parent directory:

cd \examples ENTER

Then use the **rmdir** command for the CHAP5 subdirectory. Type:

rmdir chap5 ENTER or **rd chap5** ENTER

rd is shorter, so we'll use it from here on. Type:

rd chap6 ENTER

Oops! DOS says we can't:

```
Invalid path, not directory,
or directory not empty
```

Type:

dir chap6 ENTER

It has a file in it. Delete that file with:

del chap6*.* ENTER or **del chap6** ENTER

DOS asks if you are sure. Answer Y. (We still have copies of all these files in EXAMPLES.BAK, remember?)

Now remove the CHAP6 subdirectory:

rd chap6 ENTER

Do the same for CHAP7 through CHAP10. Display the EXAMPLES subdirectory. It should be empty.

Now copy the files from EXAMPLES.BAK to EXAMPLES:

copy \examples.bak ENTER

And remove the EXAMPLES.BAK subdirectory. It's served our demonstration purpose:

**cd ** ENTER
del examples.bak ENTER Y ENTER
rd examples.bak ENTER

Everything's back the way it was at the start of the chapter! (Well — almost.)

dir ENTER

Retrieving Files Deleted Earlier

We deleted files earlier to make work space on the diskette. Now it's time to copy some of them back to our working disk from the original MS-DOS/BASIC master, or a copy, and store them in the DOSCMDS subdirectory. Specifically, we need the files named:

```
MODE.*
LINK.*
BASIC?.*
MOUSE.*
```

Some have the extension COM and others EXE or SYS, varying with the version of DOS. By using the * we get around that problem.

But something as simple as copying a specific file from one diskette to another can get tricky. Without getting too esoteric, here is an easy way to do it.

With Only 1 Disk Drive

This is easy, since the computer makes the single drive serve as both Drive A and Drive B. To retrieve the MODE.* file, simply:

copy a:mode.* b: ENTER

and follow the instructions on the screen.

With 2 Identical Disk Drives

Again, no problem. Use the same command as above, placing the original or *source* diskette in A, and the one we have been working with, the *target*, in Drive B.

With 2 Different Disk Drives

Trouble! You can't copy directly from the *source* diskette to the *target* diskette since the diskettes are the same size, but the drive sizes are different. One drive is probably for 3-1/2" diskettes, and the other one for 5-1/4". And you can't fool the computer to use one drive as both A and B since there really are 2 drives, and they are named A and B.

The simplest way around this problem (until we learn to use the "virtual disk") is to copy the files needed from the *source* diskette in A to a diskette of whatever size in drive B. Then copy the files from B back to the *target* in Drive A.

Testing, Testing

To make sure they made it:

dir doscmds ENTER

Chapter 15 Summary

Subdirectories always contain two special files, **dot** (.) and **dot dot** (..). They are used by MS-DOS to keep track of its location in the tree. **Dot** refers to the current directory. **Dot dot** refers to the current directory's parent directory.

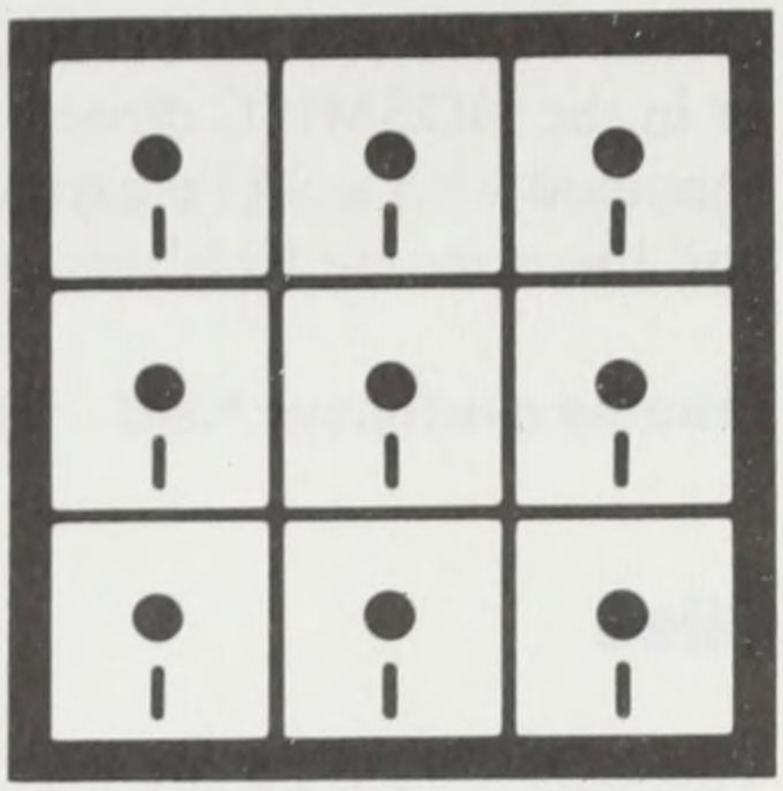
To run a COM, EXE, or BAT program, you must be in the directory that contains it. The **path** command overcomes this restriction. **path \;\utility** tells MS-DOS to search the root directory, then the UTILITY subdirectory if a program you request is not found in the current directory.

Remove a path specification with **path ;** ENTER or by rebooting.

You may have as many levels of subdirectories as you wish. The only limitation is available disk space. TREE.COM is a utility program that displays a disk's tree structure. Run it by typing **tree** or **tree/f**. The /**f** option lists all files in all subdirectories.

To remove a subdirectory, delete all the files it contains, then go to its parent and use **rd files.xyz** to remove the subdirectory named FILES.XYZ.

CHAPTER 16



Controlling DOS with CONFIG.SYS

A stereo system has controls for volume, bass and treble, and a knob for balancing the sound between the left and right speakers.

DOS also has a provision for "customizing," but instead of twisting knobs, you leave messages in a special file named CONFIG.SYS. Each time the computer is turned on, or booted, DOS reads this file and "sets the controls."

What's in CONFIG.SYS?

A CONFIG.SYS file may contain one command or many. If there is no CONFIG.SYS file in the root directory, DOS simply uses its own default settings.

Here's an example of what a CONFIG.SYS file might contain:

```
files=30
buffers=22
break=off
device=mouse.sys
device=ansi.sys
```

We'll learn what each of these commands does.

Just in Case

Check to see if a CONFIG.SYS file is already present, either at the root or in the DOSMISC directory. (We didn't put it there.) If so, it should be renamed with a SAFe extension so it can be used again when this chapter and the next are finished.

rename config.sys *.saf ENTER

Files

The CONFIG.SYS file may include a **files** specification.

files=30

To understand its purpose, think of the **copy** command. When copying a file from one place to another, two files are open at the same time: the *source* file and the *target* file. DOS keeps track of its place in both files by using *file pointers*. The **files** entry in CONFIG.SYS tells how many *file pointers* to reserve room for.

DOS automatically sets aside enough space for handling its mundane tasks, like copying one file to another, but complex programs may require several, even dozens of disk files to be open at the same time. Devices such as printers also count as files that require pointers.

The recommended minimum setting is **files=10**. You'll know that's not enough if a program sends you a message like `Too many files`, or `Not enough file handles`

If you get the `Too many files` message when using the BASIC language, check the **/f** parameter of the BASIC startup command. For example, if the BASIC program uses 5 files, you need to enter BASIC from MS-DOS with (at least) **basic /f:5**. If changing the **/f** parameter doesn't solve the problem, put a **files=** statement (with a higher number) in the CONFIG.SYS file.

Trying It Out

To get a feel for different CONFIG.SYS settings, create the following at the root:

copy con config.sys ENTER
files=10 ENTER
CTRL Z ENTER

The result is a CONFIG.SYS file containing **files=10**. Press CTRL ALT DELETE to reboot the system. A change in the CONFIG.SYS file *must* be followed by rebooting, or it will have no effect.

At the A> prompt try:

dir ENTER
chkdsk ENTER

Oops. CHKDSK.COM is not in the root directory. We must set a path to the DOS directories each time we reboot. Type:

path \doscmds; \dosmisc ENTER
chkdsk ENTER

Make a note of the bytes free Now change CONFIG.SYS to **files=11**.

copy con config.sys ENTER
files=11 ENTER
CTRL Z ENTER

Reboot again, reset the path, and **chkdsk**. The bytes free should be less this time to reflect the space reserved for an additional file pointer. You'll want to use this test with the other CONFIG.SYS entries we discuss. It's a way of determining the memory "cost" for each.

Buffers

A buffer is a storage area in memory. When a program tells DOS to put data into a file, the data goes first to a buffer. DOS may decide not to write it to a disk right away, just in case the program sends along more data to go to the same or an adjacent location in the disk file.

A buffer holds a block of (usually) 512 adjacent bytes. If you don't specify the number of buffers in a CONFIG.SYS file, DOS automatically reserves 2. For most applications, **buffers=10** is a good setting.

The BREAK Command Option

Pressing CTRL C cancels a program or command before it has finished its work. Sometimes Control-C doesn't take effect immediately, especially when a program is doing a time-consuming series of computations or disk accesses. That's because DOS normally checks for Control-C only when it is printing or displaying or waiting for input from the keyboard. This is the case when **break** is **off**.

When **break** is **on**, the ability to abort a program is enhanced. DOS checks to see if you've pressed CTRL C before *every* operation, not just before console and printout operations. This can be valuable when troubleshooting.

However, **break on** has drawbacks. The system will run a bit slower since doing all that Control-C checking takes time. Second, it may be risky. Canceling a program in the middle of a series of disk operations could leave something undone. Unless you are able to deal with the consequences, it's best to leave **break off**. CONFIG.SYS may contain:

break=on or **break=off**

You may also activate the break directly from the DOS system prompt:

break on ENTER or **break off** ENTER

Installable Devices

The **device** entry is a little different. It provides a way to add "modules" that give the computer added capabilities. For each such entry, a *pathname* tells where to find a disk file, which, when booting, DOS loads and "attaches" to itself. These optional modules are called *device drivers*.

Suppose, for example, you purchased a Logi-Mouse (Radio Shack Cat. No. 26-1199) for your computer. With certain application programs, it moves a pointer on the video display screen as you roll a palm-sized *mouse* around the desktop.

On the diskette supplied with the mouse is a file called MOUSE.SYS. It is the *device driver*. MOUSE.SYS contains program logic which DOS "installs" in memory. To install MOUSE.SYS, you must insert this message in the CONFIG.SYS file:

device=mouse.sys

Then you must copy the MOUSE.SYS file onto the *system* disk. DOS will now look for the file called MOUSE.SYS each time the computer is booted.

Device drivers can be stored in subdirectories. Suppose, for example, you want to keep MOUSE.SYS in the subdirectory called DOSMISC, where we put the other SYS files. Just add the pathname to the CONFIG.SYS message:

device=dosmisc\mouse.sys

(Since different mice versions use different driver schemes, check the manual that came with yours.)

You may find other device drivers on the DOS diskette(s). They usually have the SYS extension.

Special Drivers

Device drivers sometimes add new ways to control devices which DOS already controls. ANSI.SYS and LPDRVR.SYS are examples.

DOS already knows how to display data on the video screen and how to accept data from the keyboard. ANSI.SYS adds ways to control cursor position, text and background color, and the meanings of keys on the keyboard.

You can also control most printer text and control codes. Unfortunately, control codes other than line feed, carriage return, and form feed are not always compatible from one printer make and model to another. LPDRVR.SYS solves the problem as it applies to Tandy printers by providing a standardized way of giving commands to set number of characters per line, horizontal and vertical tabs, and page length.

ANSI.SYS is installed by putting **device=ansi.sys** in the CONFIG.SYS file. LPDRVR.SYS is installed by using **device=lpdrvr.sys**. If ANSI.SYS or LPDRVR.SYS are not placed in the root directory of the *system* disk, a pathname must be provided, as above. For example:

device=dosmisc\ansi.sys

We will learn how to use ANSI.SYS in Chapter 24.

Chapter 16 Summary

One of the first things DOS does is check the root directory of the *system* disk for a file called CONFIG.SYS. If this file is present, DOS reads it for instructions about Control-C checking, amount of memory to allocate for file handling, and special device drivers to load.

A **files** entry, if present, tells how many *file handles* to reserve space for to keep track of file input and output. **files=10** allows DOS to work with 10 files at once.

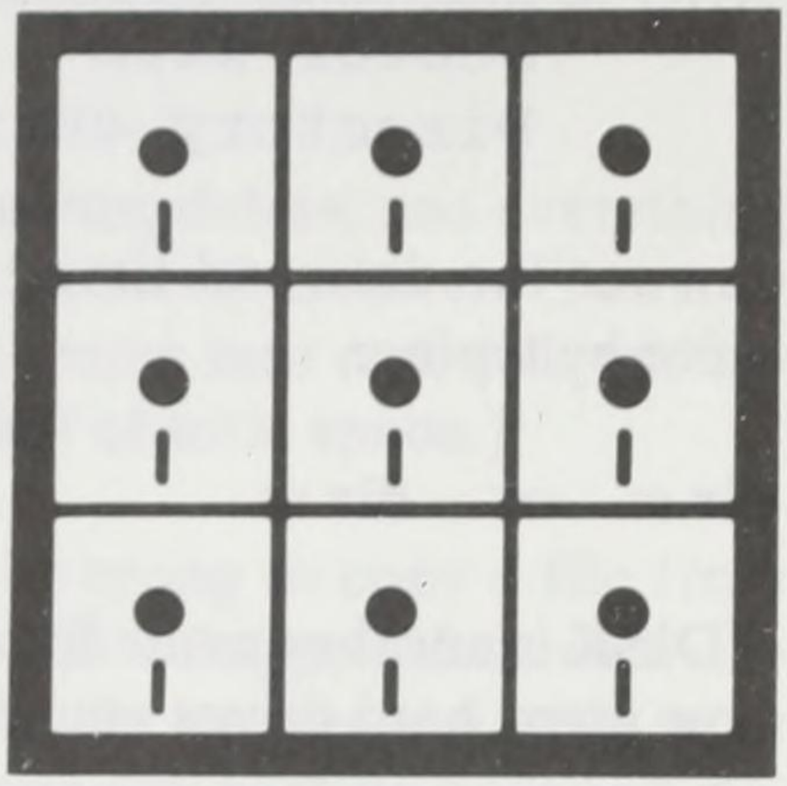
A **buffers** entry may be included to change the number of 512-byte storage locations in memory for handling the transfer of data to and from disk files. **buffer=12** sets aside 12 buffers.

A **break=on**, if present, causes DOS to check for Control-C before *each* operation it performs. Otherwise, Control-C checking is only done before keyboard, printer, display, and auxiliary input/output operations.

device entries in CONFIG.SYS specify disk files to be loaded into memory and included as part of DOS. **device=mouse.sys** tells MS-DOS to include the program logic from a file named MOUSE.SYS (stored at the root) for controlling the Logi-Mouse.

CHAPTER 17

Virtual Disk and Spooler



Virtual Disk

The moving parts are the slowest components of any computer system, and the only components subject to wear and tear. Imagine having a disk drive with no moving parts.

That's what DOS "imagines" when you install a *virtual disk*. VDISK.SYS is a DOS program that sets aside memory in the computer and uses it as if it were an extra disk drive, operating dozens of times faster than a "real" disk drive. A virtual disk can make operating easier and much faster, particularly if numerous disk *reads* and *writes* are involved.

Installing VDISK.SYS

Installing a virtual disk is easy. Just add an entry to the CONFIG.SYS file, then reboot.

With VDISK.SYS in the DOSMISC directory, at the root type:

copy con config.sys ENTER
device=dosmisc\vdisk.sys ENTER
CTRL Z ENTER

Reboot with CTRL ALT DELETE and see:

```
Tandy VDISK version 03.00 ram disk drive C
      Disk buffer       64 KB
      Sector size      128
      Directory entry   64
```

Answer the date and time requests (if necessary), then move to the new drive by typing:

dir c: or **dir d:**

VDISK is another name for *ram disk*. The drive letter assigned depends on how many hard drives you already have.

The capacity of this virtual disk is 64KB, about 1/11th that of a standard 3-1/2" diskette. Its high speed can make good use of this small amount of reserved memory.

Using the Virtual Disk

Don't let the silence spook you. The ram disk works like any other newly formatted disk. All virtual disk commands work like they do on a real disk. The only exceptions are **format** and **diskcopy**, which don't apply. VDISK can even have subdirectories.

Let's copy a "real" program to ram drive C or drive D, if you have a hard disk, and try it. How about **chkdsk**?

copy a:\doscmds\chkdsk.com ENTER
dir ENTER
chkdsk ENTER

It checks its own space blazingly fast. The `0 bytes in 1 hidden files` refers to the new volume label which was automatically assigned: VDISK. That label requires one space in the directory, but no bytes.

Next, have it **chkdsk** Drive A.

chkdsk a: ENTER

Very slow, by comparison, even allowing for A being 11 times as large. Most of the time wasted is waiting for the mechanical movement in Drive A.

Try other programs and commands. **copy, rename, delete,** and everything else work as usual — but anything involving disk access is faster. You may want to load one of your own application programs into ram disk and try it. (Remember, the ram disk has less than 64KB of total space.)

Think back several chapters to the problem of trying to copy a file from one diskette to another, where the computer has 2 floppy drives with different capacities. Can you see how easy it would be to copy each file from Drive A to the virtual disk, then from the virtual disk back to a second diskette in Drive A?

It works well, except for one little problem. Some files, like the BASIC program file are larger than the 64K available in the virtual disk. But we can solve that easily enough in just a minute.

Now the Bad News

Everything on a virtual disk is erased when the computer is turned off. Therefore, before the end of a session, it's important to **copy** any files you want to keep from a virtual disk onto diskettes or a hard drive.

Some users prefer to use a virtual disk for programs only. That way there's no danger of lost *data* due to forgetfulness or a power outage.

A Tandy standby power supply can eliminate the second possibility.

VDISK Options

There can be more than one virtual disk in memory. Just put a separate entry for each one in the CONFIG.SYS file. DOS will assign drive letters in sequence.

For specialized uses, the capacity, sector size, and directory limits can be changed by specifying parameters with **device=vdisk.sys** in CONFIG.SYS.

- Capacity is expressed in kilobytes. (1 kilobyte = 1024 bytes). The range is from zero up to the amount of available memory. (Use **chkdsk** to see the bytes free.) If you request too much, VDISK will set aside as much as it can, leaving about 64K free for running programs.

- Sector size may be 15, 128, 256, or 512 bytes. The 256 and 512 sizes mean greater speed, but less efficiency when there are multiple files on the ram disk. A regular diskette uses sectors of 512 bytes in clusters of two, so even a 1-byte file requires 1024 bytes of storage (1023 bytes are wasted). The ram disk uses 128-byte sectors in clusters of one, so at most, 127 bytes are wasted per file.

- The ram disk economizes on space by limiting the directory to 64 entries (compared to 112 entries for a regular diskette or 512 on a hard drive). The directory limit may range from 2 to 512. If you plan to use only a few files, specify a small directory. (Each directory entry takes 32 bytes. VDISK will round your request upward according to how many fit in the sector size selected.) Remember, VDISK uses 1 file for the volume label.

The following CONFIG.SYS line requests a virtual disk of 100KB, a sector size of 512 bytes, and a directory limit of 10 files. Change your CONFIG.SYS file and try it.

device= \dosmisc\vdisk.sys 100 512 10

Now there should be plenty of space in the *ram disk* to hold BASIC.

If your computer has *extended memory*, you may earmark part of it for virtual disk storage by including **/e** as in:

device= \dosmisc\vdisk.sys 100 512 10 /e

This way, all the conventional memory remains available for executing programs. If you can spare the memory, consider leaving a "permanent" **vdisk** entry in the CONFIG file on the disk you normally use for booting. Meanwhile, EVERYBODY return to A:, **del config.sys**, and reboot.

HDRIVE.SYS (For 3000s and 4000s only)

This driver is needed in the CONFIG.SYS file if you're using a hard disk type other than those supplied by Tandy and listed in your DOS manual's hard disk table. Only if you can't find your drive type in that table should you use HDRIVE.SYS, as in:

device=\dosmisc\hdrive.sys

Remember to use the SETUP program each time you add or subtract a hard drive from the computer system. After running SETUP, run the FORMAT HARD DISK utility on the Utilities Diskette to format the non-standard hard drive. See the factory MS-DOS manual for more information.

SPOOLER.SYS

Check to be sure both SPOOLER.SYS and SPOOLER.COM are on your *system* diskette in the DOSMISC subdirectory. If they're not, copy them there, from the Supplemental Programs Diskette to the *system* diskette, if necessary.

The SPOOLER.SYS device driver installs a memory resident printer buffer at a special location in memory, allowing your printer to print "in the background." This means you won't need to wait for printing to finish before using the computer for something else.

The SPOOLER is installed by adding the following line to the CONFIG.SYS file:

device=spooler.sys */printer /buffer /memory*

Note the 3 options included with the SPOOLER. They are:

printer	is the number of the printer port to spool to. Use 1 for LPT1 or 2 for LPT2. The default is 1.

buffer	is the size of the spooler's buffer in kilobytes. The *buffer* size must be three digits, so place a 0 in front as needed. For example, if you want a 64K buffer, use 064. If no number is given, SPOOLER creates a 20K buffer.
memory	If your computer is equipped with extended memory, you can utilize it with the /**e** option.

To install a 20K printer spooler for LPT1 (your first, or only printer port), everybody place this line in a new CONFIG.SYS file:

device=dosmisc\spooler.sys

and reboot.

Other examples:

To install a 64K printer spooler for LPT1, use:

device=dosmisc\spooler.sys /1 /064

To install a 20K spooler for LPT1 in extended memory:

device=dosmisc\spooler.sys /e

SPOOLER.COM

SPOOLER.SYS is the device driver, installed by CONFIG.SYS. After the SPOOLER is installed, it becomes part of DOS. The SPOOLER.COM program which came with DOS is used to further control the spooler.

Type:

doscmds\spooler/g ENTER

The **spooler** command options are single character switches that control the spooler:

/p — causes the spooler to pause. It is similar to pressing the on-/off-line switch on the printer. Printing restarts automatically when the printer buffer is full. It can also be restarted with a second **spooler /p** command.

/s — turns the spooler off. Text sent to the printer after the **spooler /s** command is not spooled, but goes directly to the printer. Specifying **spooler /s** again restarts spooler.

/c — clears all data from the spooler buffer.

/g — gets the status of the spooler, returning its size, whether or not it is paused, and the percentage of the spooler that's full.

Testing the Spooler

With your printer either off, or off-line, type:

dir > prn ENTER

This command sent a copy of the diskette's directory into the spooler and to the printer, which is off.

Now check the status of the spooler by:

doscmds\spooler/g ENTER

It should report that the **Storage used** is several percent. The directory is stuck in the spooler since the printer can't accept it.

Now clear the spooler buffer with:

doscmds\spooler/c ENTER

and check it again:

doscmds\spooler/g ENTER

It's empty, its contents having been erased.

Experiment also with the /**s** and /**p** options until you are comfortable with the spooler's commands.

Finally, turn your printer on and on-line and print out the root directory. Then print out some subdirectories.

The spooler's real advantage shows up when printing long documents that tie up the entire computer. If you wish to pursue spooling further (but not as part of this book), use the TEXT processor in *DeskMate* or another word processor and try printing through a spooler. With a little practice, you might find the spooler to be something you can't do without. (Do not goof up your current system diskette, as we have a long way to go before completing Advanced MS-DOS.)

Delete the CONFIG.SYS file before continuing to the next chapter.

Chapter 17 Summary

A *virtual disk* is an imaginary disk drive created in memory, providing much greater speed than a "real" disk drive.

For each virtual disk to be established, include a **device=vdisk.sys** entry in the CONFIG.SYS file. Optional parameters specify the capacity in kilobytes, the sector size, and the directory limit. With no parameters, the defaults are 64, 128, and 64, respectively. /**e** places the virtual disk in extended memory.

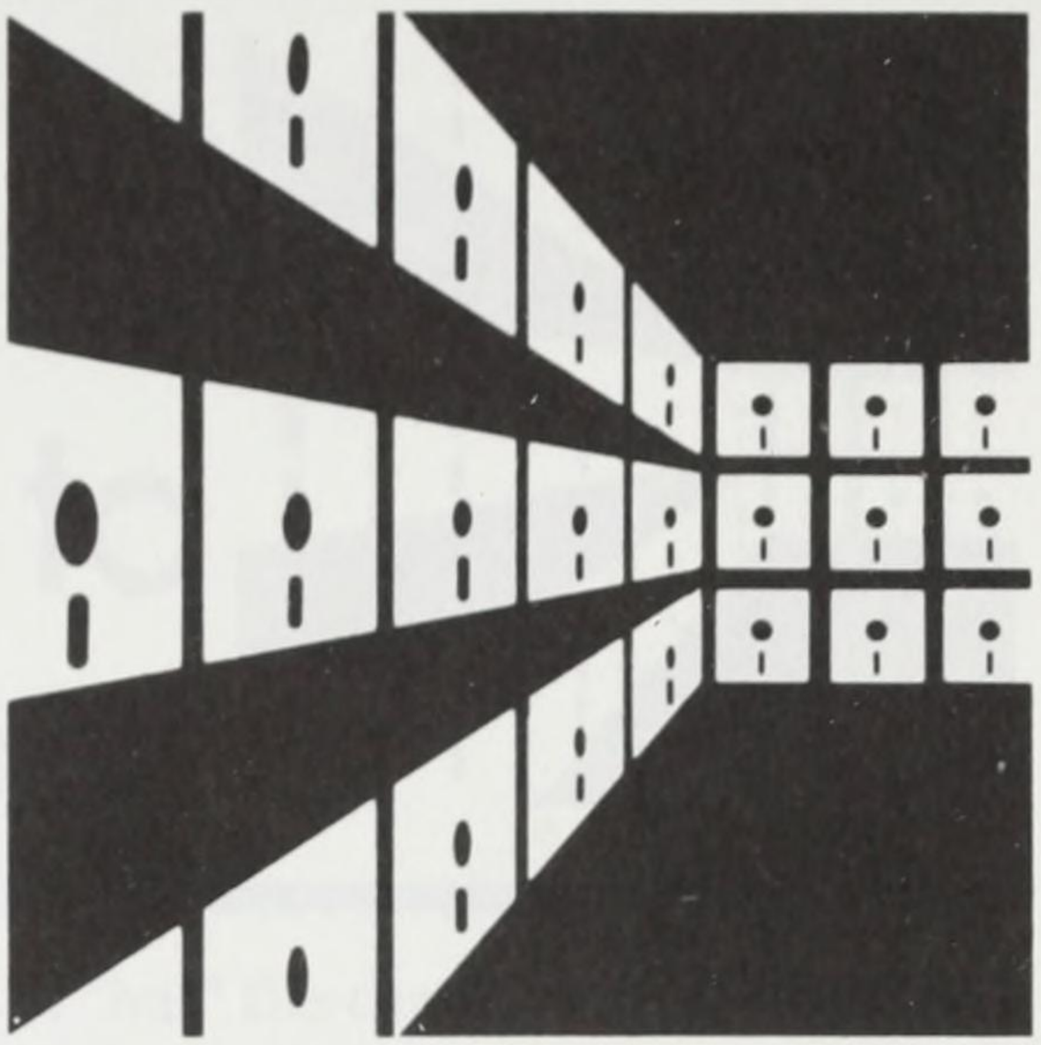
After booting, DOS assigns a drive letter to each virtual disk. Data remains in a virtual disk only while computer power is on. Rebooting clears and recreates it.

Use **copy** to transfer files to and from real disk drives. To regain use of the memory previously reserved for a virtual disk, remove the **device=vdisk.sys** entry from the CONFIG.SYS file and reboot.

HDRIVE.SYS is used in the CONFIG.SYS file if your hard drive is not a Tandy model or one mentioned in your DOS manual's hard drive table.

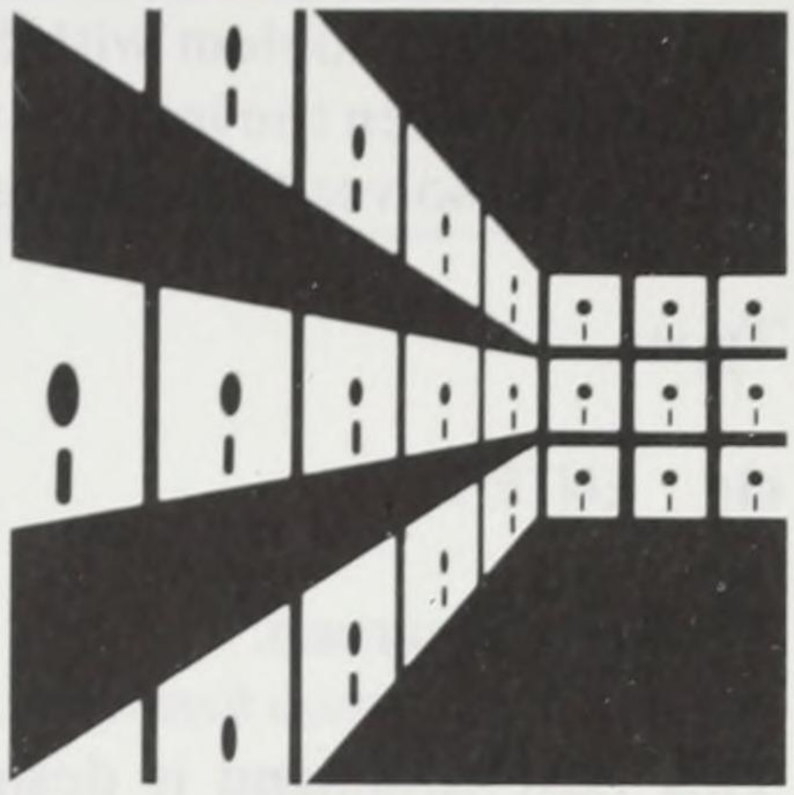
A *spooler* is a part of memory reserved for use as a printer buffer. It stores information to be printed, and lets you do other things with the computer while the printing takes place .

PART 4



Batch files

CHAPTER 18



Intro to Batch Files

A batch file is a "to do" list for DOS. A "bat" file can remember an almost endless list of commands for DOS to execute, and you can create an almost unlimited number of bat files to meet your own specialized needs. The best way to learn about batch files is to write some.

Put the *system* disk in Drive A, and set the system prompt to `A>` in the root directory.

Since this is a short file, instead of using **edlin**, we'll write it by copying from the console.

copy con subdir.bat ENTER
dir doscmds /w ENTER
dir dosmisc /w ENTER
dir examples /w ENTER
CTRL Z ENTER

SUBDIR.BAT is a batch file named SUBDIR which contains **dir** commands to display 3 directories you created earlier. Try it:

subdir ENTER

Think of SUBDIR as a new "command" you created for DOS. It combines 3 commands into one. "Customized commands" can make life much easier!

CLS and PAUSE

There is one problem with SUBDIR.BAT. The first directory rolled off the screen when the last one was displayed. How about an "enhanced" version which solves that problem?

Type:

cls ENTER

to clear the screen.

This next command is designed especially for batch files. At the A> prompt, type:

pause ENTER

DOS responds with:

```
Strike a key when ready . . .
```

Press a key and the A> prompt returns. That's not too exciting right now, but **pause** is just what we need for this SUBDIR batch file.

Modify SUBDIR.BAT so it contains the following:

```
cls
dir doscmds /w
pause
cls
dir dosmisc /w
pause
cls
dir examples /w
```

Use **edlin** to make the changes, or if you are a fast typist, just retype the entire file from the console.

Remember, **edlin** is in the DOSCMDS subdirectory. You want the batch file to end up in the root, so your command sequence is:

path doscmds ENTER
edlin subdir.bat ENTER

Ready to try it? Type:

subdir ENTER

The screen clears before each directory, and the **pause** command halts execution so we can read the individual listings.

ECHO and REM

The **echo** command "echoes" messages to the screen. Try this:

echo Hello there! ENTER

DOS responds with `Hello there!` Just like a real echo.

echo can also suppress messages.

Run **subdir** again and watch carefully. Notice how each command is displayed as it is executed.

```
A>cls
```

executes so fast that it is not seen, but when it comes to:

```
A>pause
```

execution pauses. It is neater (and less confusing) if SUBDIR.BAT does its job without showing all the commands and system prompts.

To hide batch commands from view as they are executing, simply include **echo off** in the batch file.

In case you ever need it, **echo on** cancels the last **echo off**. Without a parameter, **echo** simply displays the status: `ECHO is on` or `ECHO is off`.

The **rem** command is for remarks. It is a way of making notes to yourself about the purpose and workings of a batch file. If an **echo off** command precedes the remark lines, they are not displayed while the batch file is executing.

To see how **echo** and **rem** are used, edit SUBDIR.BAT so it looks like this:

```
echo off
rem Our purpose is to learn about batch commands.
rem It assumes you are in the root directory and
rem DOSCMDS, DOSMISC, and EXAMPLES are present.
cls
echo There are 3 subdirectories on this disk.
pause
cls
dir doscmds /w
pause
cls
dir dosmisc /w
pause
cls
dir examples /w
```

The **echo off** command suppresses display of all other commands in the batch. The remarks explain. The `echo There are 3...` line adds a message as the batch file executes.

Try it. Type:

subdir ENTER

Control-C and Batch Files

Execution of a batch file can be terminated by pressing CTRL C. Type:

subdir ENTER

again, and before it ends, press CTRL C. DOS displays:

```
Terminate batch job (Y/N)?
```

If you type Y, the batch operation is terminated. Typing N allows the batch operation to resume.

Replaceable Parameters

Replaceable parameters can make batch files even more flexible. To create a one-line batch file called D.BAT, type:

copy con d.bat ENTER
dir %1 ENTER
CTRL Z ENTER

Type:

d ENTER

and see a directory. Now:

d /w ENTER

to see a wide directory listing.

The replaceable parameter is **%1**. It tells DOS to put whatever you type after the **d** wherever it finds **%1** in the batch file. When you typed **d /w**, it replaced the **%1** with **/w** to make **dir /w**. Try a few other parameters:

d command.com ENTER

d doscmds ENTER

d doscmds*.com ENTER

Very elementary, but demonstrates the principle. Now try:

d doscmds*.com /w ENTER

D.BAT ignored the /w, a second parameter. Let's make a batch file that allows more than one parameter.

copy con hello.bat ENTER
echo off ENTER
echo Hello %1 from %2. ENTER
echo How do you like living in %2? ENTER
echo %2 is my favorite town! ENTER
CTRL Z ENTER

At the A> prompt, type:

hello Joe Boise ENTER

Here's what we get:

```
A>echo off
Hello Joe from Boise.
How do you like living in Boise?
Boise is my favorite town!

A>
```

Test it with a few other replacements. See how it works?

A batch file may contain up to 9 replaceable parameters: **%1** through **%9**. **%0** is special. It is reserved for the name of the batch file. To see how it works:

copy con name.bat ENTER
echo off ENTER
echo This batch file is called %0 ENTER
CTRL Z ENTER

name ENTER

DOS responds with `This batch file is called name.`

The use of batch files opens up all kinds of possibilities. In the next chapter, we will practice more commands that make them especially powerful.

Chapter 18 Summary

Batch files execute MS-DOS commands automatically. Just enter the batch file name.

The **pause** command makes the system wait until a key is pressed.

The **cls** command clears the screen.

Use **echo** for displaying messages to the operator.

Remarks may be included in batch files by using **rem**.

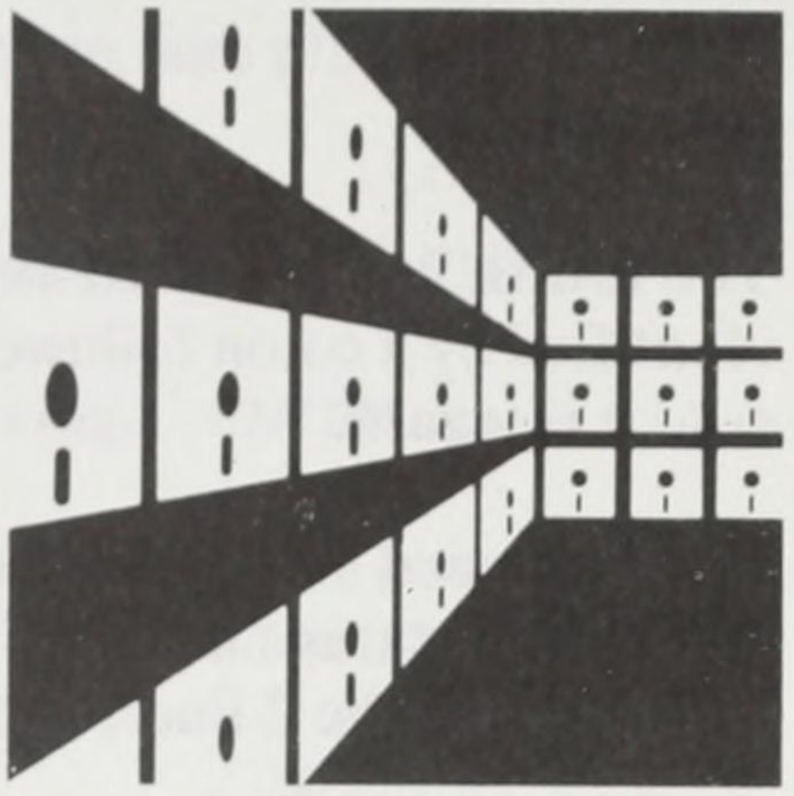
CTRL C terminates batch file execution.

Normally, all commands in a batch file are displayed, just as if you were typing them at the keyboard. The **echo off** command suppresses them.

Replaceable parameters make batch files more flexible. Wherever DOS finds **%1** in a batch file, it substitutes the first parameter the operator included when typing the batch file name. Up to 9 parameters can be handled this way. **%2** is the second, **%3** is the third, and so forth. **%0** is reserved to hold the name of the batch file.

CHAPTER 19

Making Batch Files Smarter



Our batch files so far have been just lists of commands for DOS to execute, one after another, from beginning to end.

In this chapter we work with commands that alter the flow. We'll make DOS use some logic and jump from one place in the batch file to another. We'll also see how batch files can make decisions. The new commands make it possible to write rather sophisticated programs — even if you aren't a programmer.

GOTO

The **goto** command tells DOS to jump from one place to another. It is most useful in batch files which do the same thing over and over.

Create the following as STICKERS.BAT, substituting your own name and mailing address in lines 5 through 7 and pressing ENTER at the end of each line:

```
echo off
echo Turn on your printer and press [CTRL] [P]. Then...
pause
:again
echo Oswald J. Hammerfest
echo 3340 Western Avenue
```

echo Omaha, Nebraska 68132
echo .
echo .
goto again

The fourth line is known as a *label*. A label is a destination address. It is identified by a colon followed by a word, which is the label's name. In this case it is **:again**.

Since we need 2 blank lines after the address and just typing "echo" would give us the status instead of a blank line, we have a dot following "echo." This gives us the 2 lines, and the dot is small and insignificant.

The final line is a **goto** command. It causes execution to jump back to **:again**. The effect is to endlessly repeat lines 5 through 9 until you press CTRL C or the RESET button. This is known as a continuous loop.

Try STICKERS.BAT, with or without a printer. Pressing CTRL P sends output to the printer as well as the screen. With continuous gummed labels in the printer, you can create a file of return address stickers.

IF EXIST

The **if** command makes it possible to program automatic decision-making into batch files. You can make a batch file do different things, depending on the conditions it encounters.

We'll use **if exist** first. Try it at the system prompt. Type:

if exist command.com echo Yes it is here. ENTER

DOS checks the diskette, finds a file named COMMAND.COM, and responds with `Yes it is here`.

The decision line has two parts: a *condition* and an *action* to be taken if the condition is true. The *condition* is: **if exist command.com**, which in plain English means "if a file named COMMAND.COM exists." The *action* is: **echo Yes it is here**. Try this test on a file that doesn't exist:

if exist comet.com echo Yes it is here. ENTER

The A> prompt returns without any action.

You can insert **not** after the **if** to make the action take place *if* the condition is *not* true. Try these:

if not exist command.com echo command.com is missing. ENTER

if not exist comet.com echo comet.com is missing. ENTER

In the first case, COMMAND.COM is present, so no message is displayed. In the second case, a file named COMET.COM does not exist, so DOS reports `comet.com is missing.`

IF EQUAL

Let's try another variation of the **if** statement:

if apples==oranges echo yes ENTER

No response. The item on the left of the double equal sign, apples, is not the same as the item on the right, oranges. (Batch files use == to mean =.)

Now try:

if oranges==oranges echo yes ENTER

```
yes
```

How about:

if Oranges==oranges echo yes ENTER

No response. Since the first Oranges is capitalized, but the second is not, they are considered *unequal.*

if not apples==oranges echo They're not equal! ENTER

DOS reports `They're not equal!`

The real value of **if ==** is when it's used with replaceable parameters, as you'll see in the next chapter.

IF ERRORLEVEL

Another **if** statement is available, but unless you are working with programs that have been designed a certain way, you won't have an occasion to use it.

When writing COM or EXE programs, the programmer has the option of supplying an error code with his *terminate* instruction so a batch file can take appropriate action. An error code of 0 means the program ran without problems. Error codes of 1 and higher can be used to indicate special conditions, such as "disk full," "file not found," or "insufficient memory."

Most of the COM and EXE programs on the MS-DOS/BASIC disk do not return error codes, so we can't use **if errorlevel** with them. The following example is, however, worth reading.

ONEDISK.EXE (Tandy 1000 Using 2.11 Only)

Early versions of MS-DOS for the Tandy 1000 included a program called ONEDISK.EXE. Its sole purpose was to check the disk drives. If only one drive was present, it returned an error code of 1. If two or more drives were installed, it returned an error code of 0. This is how it worked. When the user typed:

onedisk ENTER

the system prompt returned. Nothing appeared to happen until the user typed:

if errorlevel 1 echo You have just one drive ENTER

If there was just one drive, the message was displayed. If there were 2 drives, it wasn't.

The number after **if errorlevel** can range from 0 to 255. It causes an action to be taken if the error code is that number or higher. Here's an example of some logic that could be included in a batch file:

```
echo off
onedisk
if errorlevel 1 goto single
echo Please insert your data diskette in Drive B
goto end
:single
echo Please insert your data diskette in Drive A
:end
```

The message tells the operator to put the disk in Drive B if 2 drives are present, or in Drive A if only 1 is present.

Chapter 19 Summary

The **goto** command causes DOS to jump from one point to another in the batch file it is executing. The destination for the **goto** is given by specifying a *label*. A label may be any string of up to 8 characters, preceded by a colon. **goto loop** causes DOS to look for a destination line named `:loop` in the batch file. Execution continues there.

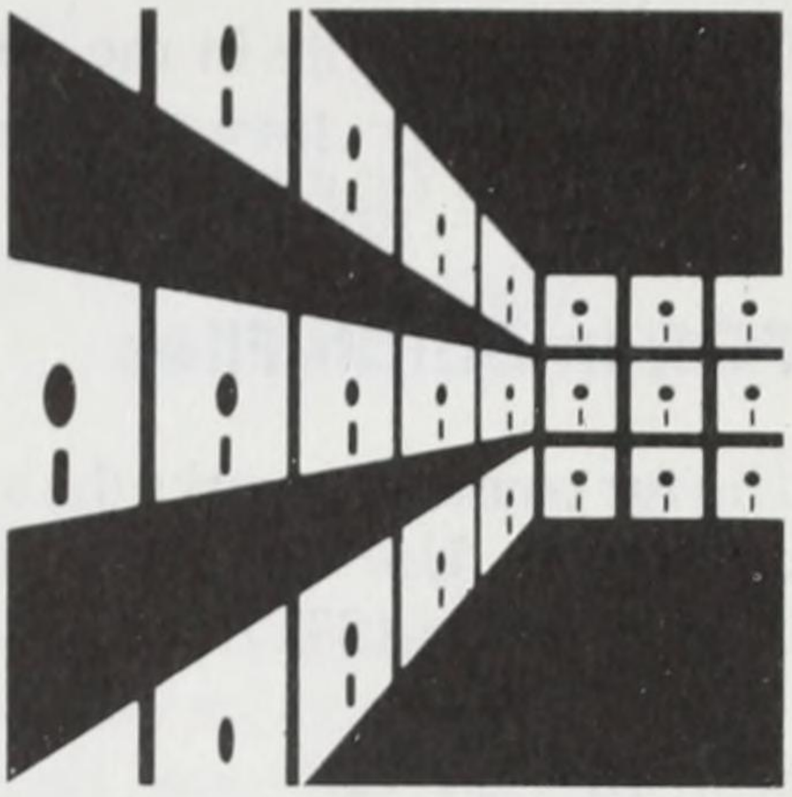
The **if** command causes DOS to consider a condition and to take an action if the condition is true. Three types of **if** commands are available: **if exist**, **if ==**, and **if errorlevel**. You may place **not** after the **if** to make the action take place if the condition *is not* true.

if exist is used with a file name. If the file is present in the current directory, the condition is true and the action is taken.

if == can compare one string to another. This is most common with replaceable parameters.

if errorlevel causes an action to be taken if the preceding COM or EXE program returned an error code.

CHAPTER 20



Repetitive Tasks in Batch Files

The **for** command allows you to say *for* everything in a list, *do* this. For example:

for %f in (ready set go) do echo %f ENTER

Study this one carefully. You supplied a list of 3 items: **ready**, **set**, and **go**. One by one, DOS replaced **%f** with each item and executed the command to the right of **do**.

Suppose you want to see the 3 subdirectories on your DOS diskette. Type:

for %f in (doscmds dosmisc examples) do dir %f ENTER

Each directory is displayed. The items between the parentheses are called the *set*. The set can be just about anything: file names, commands, paths, drive letters, or words to be filled in. You may use spaces, commas, or semi-colons to separate the items.

The **for** command is even more powerful when a wild card is used for the set. Try this:

for %f in (*.bat) do type %f ENTER

Each of the batch files is typed to the screen.

for %f in (*.*) do pause %f ENTER

DOS lists each file in the directory. After each name, you are asked to `Strike any key when ready`.

FOR in Batch Files

The **for** command works the same way when placed in a batch file, with one difference. Use a double percent sign with **f** instead of one. Try this two-liner. Call it LISTDIR.BAT.

echo off ENTER
for %%f in (*.*) do echo %%f ENTER

Type:

listdir ENTER

The list looks something like this:

```
COMMAND.COM
STICKERS.BAT
LISTDIR.BAT
.
.
.
```

etc.

A new kind of **dir** command! Only the file names are listed, one after another.

Here's another batch file. Call it BYTYPE.BAT. Press ENTER at the end of each line.

```
echo off
echo ***** COM Files *****
if not exist *.com echo None present.
for %%f in (*.com) do echo %%f
echo ***** EXE Files *****
if not exist *.exe echo None present.
for %%f in (*.exe) do echo %%f
```

```
echo ***** BAT Files *****
if not exist *.bat echo None present.
for %%f in (*.bat) do echo %%f
```

As you can see, BYTYPE.BAT lists the COM, EXE and BAT files in your current directory. The **if not** commands display `None present`.

SHIFT

Suppose you want a special version of BYTYPE.BAT that allows specifying the kinds of files to list. You could use the replaceable parameters, with **%1** indicating the first file type, **%2** for the second, and so forth, but a better solution is available.

The **shift** command slides all replaceable parameters down one position. The **%1** parameter becomes **%0**, **%2** becomes **%1**, **%3** becomes **%2**, and so forth. One purpose of **shift** is to allow more than 9 parameters. More commonly, **shift** is used for repeating the same operation for each parameter entered.

Create this file as PICKDIR.BAT, pressing ENTER at the end of each line:

```
echo off
if not x==%1x goto again
echo Parameters are required!
echo Examples: pickdir com exe bat
echo Try again please.
goto end
:again
echo ***** %1 Files *****
if not exist *.%1 echo None present.
for %%f in (*.%1) do echo %%f
shift
if x==%1x goto end
goto again
:end
```

Notice the **shift** command on the 11th line. It will cause **%1** to contain the next parameter each time the program jumps back to **:again**.

To run PICKDIR.BAT, use file extensions as your parameters. Try this first:

pickdir com exe bat ENTER

The display should be the same as that for BYTYPE.BAT. First the COM files are listed, then the EXE files, and finally the BAT files. Type:

pickdir bat dat ENTER

Only the files with BAT and DAT extensions are displayed. Now try:

pickdir ENTER

PICKDIR.BAT requires at least one parameter. We programmed a special trap for this. The display shows:

```
Parameters are required!
Examples:  pickdir com exe bat
Try again please.
```

The second line did the job:

if not x==%1x goto again

Since no parameter was entered, **%1** was nothing. Line 2 became:

if not x==x goto again

so control went to line 3. The **x==%1x** is just a trick to check for an empty parameter. The **x** was arbitrary. It avoided the possibility of having a blank on either side of the equal signs, which would confuse DOS.

You can see a similar trick in line 12:

if x==%1x goto end

This time it checks to see if any parameters remain after having shifted them.

Chapter 20 Summary

The **for** command allows operations on each item in a set. **for %f in (format.com diskcopy.com chkdsk.com) do copy %f b:** copies the three files to Drive B.

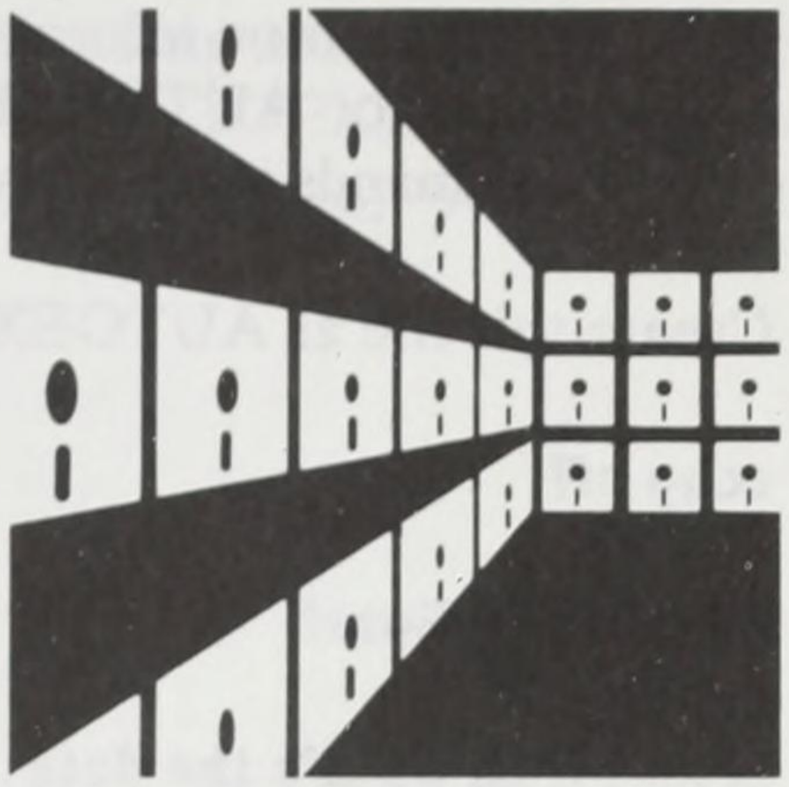
You may use wild cards to specify the set. **for %f in (*.txt) do type %f** types all TXT files.

In batch files, use **%%f** to represent each item in the set. Example: **for %%f in (A: B:) do dir %%f** displays a directory of drives A and B. **for %%f in (beg end) do erase *.%%f** erases all files having BEG or END as their extension.

The **shift** command moves all replaceable parameters down one position. **%1** becomes **%0**, **%2** becomes **%1**, and so forth. This makes it possible to use more than 9 parameters in a batch file.

CHAPTER 21

The AUTOEXEC.BAT File



As we have seen, MS-DOS files and programs can be placed in separate subdirectories where they are independent of each other, yet on the same disk. Only 3 DOS programs must be in the root directory:

IBMBIO.COM or IO.SYS
IBMDOS.COM or MSDOS.SYS

and

COMMAND.COM

IBMBIO.COM and IBMDOS.COM are invisible files. They do not appear in a directory listing.

When the computer boots up, DOS asks for the date and time, then displays the `A>` prompt. At least that's how it *usually* works.

Knowing something about batch files, we can make DOS perform other tasks. Perhaps you want it to say "Hello Boss!" each time it is booted. Maybe you want it to set the path, check the disk, and list the files in the directory. We can make it do all these things, and more.

The secret is a *very special* batch file with the reserved name AUTOEXEC.BAT. On each boot, the root directory is checked and specific action taken. The order of activity is:

- Load COMMAND.COM into memory.
- Look for CONFIG.SYS. If it exists, load the device drivers and configuration information into memory.
- Look for AUTOEXEC.BAT. If it exists, DOS executes the commands it contains.

Create this file as AUTOEXEC.BAT.

```
echo off
cls
echo Hello Boss!
echo .
echo First, what's the date ...
date
echo .
time
path=doscmds
```

Check the file with:

type autoexec.bat ENTER

and reboot. Note that just as with the CONFIG.SYS file, rebooting is required after any change in order for the file to be effective.

This AUTOEXEC.BAT file turned off the echoing of commands making for a neater display. It cleared the screen and said "Hello Boss!" Then it prompted you for the date and time. Finally it set the path so all files in the DOSCMDS directory were available from the root. Since the AUTOEXEC.BAT file is AUTOmatically EXECuted each time the computer is booted, you no longer have to be concerned about the path.

If your computer has an internal clock, the date and time commands can be removed. Replace the AUTOEXEC.BAT file with this simpler one, adding **date** and **time** if needed:

```
echo off
path=doscmds
dir
```

Reboot.

At Last, a Road Map

We have struggled through subdirectories, rechecking with **cd** to see where on the diskette we were. Now that we have an AUTOEXEC.BAT file, a very nice feature can be used. Add this simple line, anywhere in your AUTOEXEC.BAT file:

prompt=pg

and reboot.

```
A:\>
```

Wow! Look at that fancy prompt. What does the backslash mean?

It means we are in the root directory. **prompt=pg** set the prompt to display the drive name, current directory, and a > symbol followed by a space. The prompt and its many other options will be explained in detail in upcoming chapters.

Meanwhile, type:

cd doscmds ENTER

and look at the prompt. It tells exactly where you are in the disk's tree structure. Type **dir** to be sure it's true.

Try:

cd \dosmisc ENTER

and, after oogling the screen some more,

cd \examples ENTER

What a great invention!

Now type **path** and see what happens.

The path was automatically set by the AUTOEXEC.BAT file. This is almost too good to be true. But it is true. And we've only just begun.

The Batch Wrap Up

As a wrap up to this section on BAT files, create a new subdirectory off the root named BATCH:

md batch ENTER

Copy all the batch files from the root to this new subdirectory. Then delete all the batch files from the root, but copy the AUTOEXEC.BAT file back, since it must remain there. Just like what you did with the COMMAND.COM file a long time ago.

If you are getting at all low on diskette space, do not copy any BAK files to the BATCH subdirectory. Just delete them.

When you are done, the root directory should contain only the COMMAND.COM and AUTOEXEC.BAT files, plus the four subdirectories.

A Path More Elegant?

But what about the path? We have only one path setting but 4 different subdirectories. How about a path to each subdirectory?

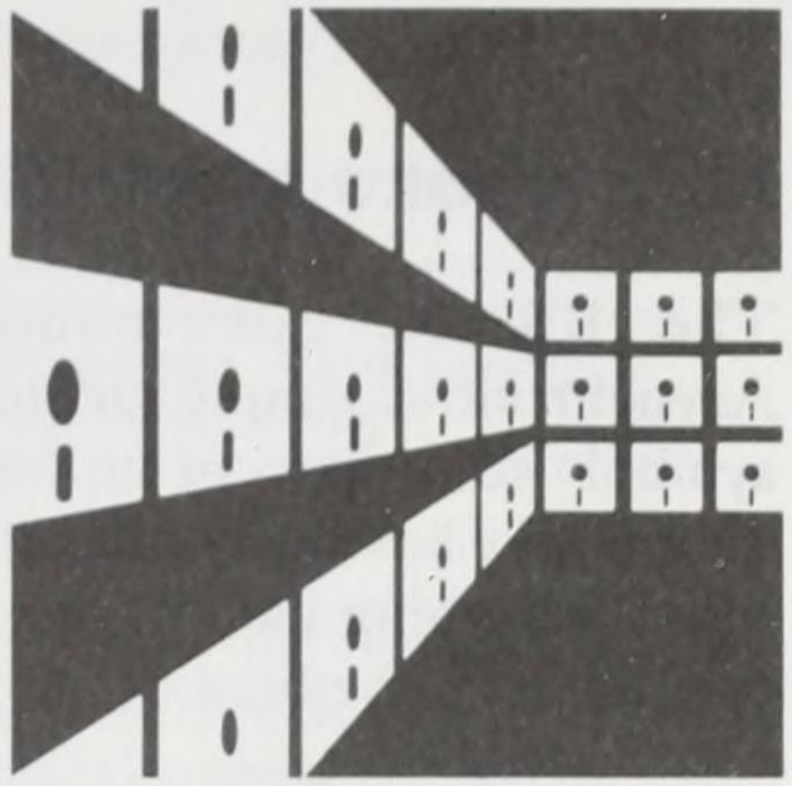
That's pretty good thinking, but think a little harder. How many of those subdirectories contain commands that we need or want to access frequently, like **format**, **copy**, **chkdsk**, etc. Right. The important commands are all residing in DOSCMDS, so we are in good shape, for now. Before this safari through the computer jungle is over, however, we will have a very elegant **path** command.

Chapter 21 Summary

AUTOEXEC.BAT is a special and unique batch file. It is executed automatically each time the computer is booted. We can customize it by installing many helpful commands which squeeze more power out of the computer in less time and with less work.

CHAPTER 22

DOS and Communi-cations



How about an "armchair" chapter? The subject is communications.

Your computer can be connected to other computers and to a variety of accessories. Because of technical differences among them, some "personalizing" of the connections are usually necessary. Knowing the fundamentals of communications will help you interconnect successfully.

The RS-232

Early in the book we learned of the auxiliary device, AUX. What does an "aux" look like? It is all or part of a circuit board inside the computer. The end of the *RS-232 board* can be seen as a socket with either 25 or 9 holes or pins on the back panel. A cable may connect it to a *modem* (the device that makes telephone hookups possible), a serial printer, a scientific instrument, or some other accessory.

"RS-232" is a standard for interconnecting computer equipment that was agreed upon years ago by the Electronic Industries Association. As a standard, RS-232 dictates that certain wires in the cable will carry certain types of signals. The RS-232 board sends data, bit by bit, via the proper wires in the cable, or (if you are inputting from a device) it reads signals from the cable and feeds them to the computer.

With the proper RS-232 boards installed, Tandy computers can utilize two or more RS-232 *ports*, addressing them as COM1 and COM2.

Asynchronous and Serial

The RS-232 port handles *asynchronous serial communications.* Asynchronous simply means "not synchronized." Letters, numbers, and symbols come into or go out of the computer one after another without necessarily having a uniform time interval before and after each one, much like a marching band going single file through a hallway without a drummer to keep the beat.

> The opposite is *synchronous communications*. The devices sending and receiving synchronous data are, in effect, both watching the same clock. With each "tick," a certain amount of data is placed on the communications line by the sender and retrieved from the line by the receiver.

Serial means "one bit after another." Each character is represented by a series of (usually) 8 bits, which in turn are each represented by "on" or "off" voltage levels. At the other end of the cable, another RS-232 board reassembles these bits into the original characters.

> The opposite of serial data transmission is *parallel*. Instead of breaking each character into 8 bits and passing them one after another along the same wire, the 8 bits are put onto 8 separate wires, and they are sent all at once.

If all the bits go down a single wire, why does an RS-232 cable have up to 25 wires? Actually, only a few of them are used. For a little more background, let's consider what some of them do.

The Serial Printer Connection

When connecting to a serial printer, as few as three of the wires in the cable may do the job. One transmits the data from the computer to the printer. A second wire is the "busy" indicator, used by the printer to tell the computer to "wait a second while I catch up." The third wire acts as a common or ground.

An External Modem

When the external device is a modem, more coordinating must be done, so more wires are used.

Modem is short for Modulator DEModulator. It is a "box" that changes electronic pulses from the computer into tones that can be sent over a telephone line. Converting data to tones is called *MOdulating*. Converting tones back to data is *DEModulating*.

In this setting, one wire in the RS-232 cable handles the data being sent. Another handles the data being received. Other wires make sure both sides of the connection are ready. Signals are sent across them to indicate that certain events have taken place. Some of them are devoted to handling messages between the modem and the computer.

DSR - Data Set Ready	indicates the remote computer is ready to transmit.
CTS - Clear To Send	indicates the remote computer is ready to receive.
DCD - Data Carrier Detect	indicates that a signal is coming in over the phone line adequate to meet quality standards.
RI - Ring Indicator	indicates that the local phone is ringing.

Other lines pass signals from the local computer to the modem:

DTR - Data Terminal Ready	indicates the local computer is ready to receive data.
RTS - Request To Send	indicates the local computer is ready to send data.

Internal Modems

Another way of handling phone communications is with an *internal* modem, such as the 1200/300 Baud Modem Board for the Tandy 1000 (Radio Shack Cat. No. 25-1013B). Instead of being in a separate box connected by a cable, the modem plugs in directly inside the computer. Two modular telephone jacks are exposed through the back panel of the computer. One jack connects to the telephone wall outlet. A telephone plugs into the other.

All the needed RS-232 circuitry is built into an internal modem. As far as MS-DOS is concerned, it is simply serial port COM1 or COM2.

When using two RS-232 boards, or an RS-232 board and an internal modem, you must specify which is to be COM1 and which is to be COM2 by setting a switch, or "jumper," on each. The instructions that come with them tell the where and how.

Computer to Computer

You can connect two computers together, provided each has an RS-232 port. It can be done without modems if the distance between them is less than a mile or so. No phone lines are involved. To do so, you need a special adapter called a *null modem*. (Radio Shack Cat. No. 26-1496.)

A null modem is a device that "switches" certain wires in the cable going from one computer's RS-232 port to the other's, usually #2 and #3, and sometimes makes additional changes. The "transmitted data line" from each computer goes to the other computer's "received data line." Think of it this way. When you talk into a telephone, what goes into your mouthpiece goes into the other person's earpiece. What goes into his mouthpiece goes into your earpiece. The idea's the same.

The Baud Rate

Look at this sentence:

```
What's up Doc?
```

To transmit it, your computer will send the "W" first, then the "h," the "a," and the "t." For the "W" it sends a pattern of 8 bits representing the ASCII "W":

```
01010111  -->
```

The bits are sent one after another, starting with the *least significant bit* (the one on the right):

```
01010111  -->
```

The RS-232 standard specifies two different voltage levels to represent the ones and zeros. When sent over the phone lines, a modem translates these voltages to different pitches of sound. At the other end of the phone line, another modem converts the sounds back into voltage levels, and the RS-232 circuit converts them back into a string of bits representing a "W."

To send the lower case "h," the computer uses another series of 8 bits:

```
01101000  -->
```

Communicating this way with a stream of bits raises a question. How does the device at the other end know when one cluster of 8 bits ends and another starts?

The solution is timing. If both devices can agree on the number of bits to be transmitted per second, the interchange becomes possible.

Bits per second is called the *baud rate*; 300 baud means data is sent *at the rate* of 300 bits per second.

Actually, there's a technical difference between baud rate and bits per second (bps). Baud is the rate at which the modem modulates its tone per second. Bits per second is the speed of data transfer. (Some modems are capable of encoding two data bits per baud.) In practice, the terms are loosely equated. For our purposes, at the DOS level and in communications programs, baud rate refers to bits per second.

The baud rate used is related to the ability of the RS-232 board, which, in turn, is related to its cost. Higher speeds cost more money. It can send and receive characters only at factory-specified baud rates. It is up to you (or the communication program you are running) to select from the rates available.

Typical rates for microcomputer applications are 300, 1200, 2400, and 9600 baud. 300, 1200, and 2400 baud are common for transmitting data over phone lines. The higher speeds are used more for communication to local devices, such as printers and computer networks. The RS-232 board may also handle certain other standard but less common rates as well: 110, 150, 600, and 4800 baud.

Start and Stop Bits

How does the receiving computer or device know when the first bit of a character has arrived? While waiting for the first character, it "hears" a continuous tone (representing a voltage level). The clock in its RS-232 board chops this tone into bits, the rate determined by where its baud rate has been set. In effect, it is receiving a series of one-bits:

```
11111111111111111111  -->
```

At 300 baud, each of these one-bits lasts one 300th of a second. They are called *mark bits*. The communications line is idle. Before the first bit of the first character, a zero bit is sent for one 300th of a second:

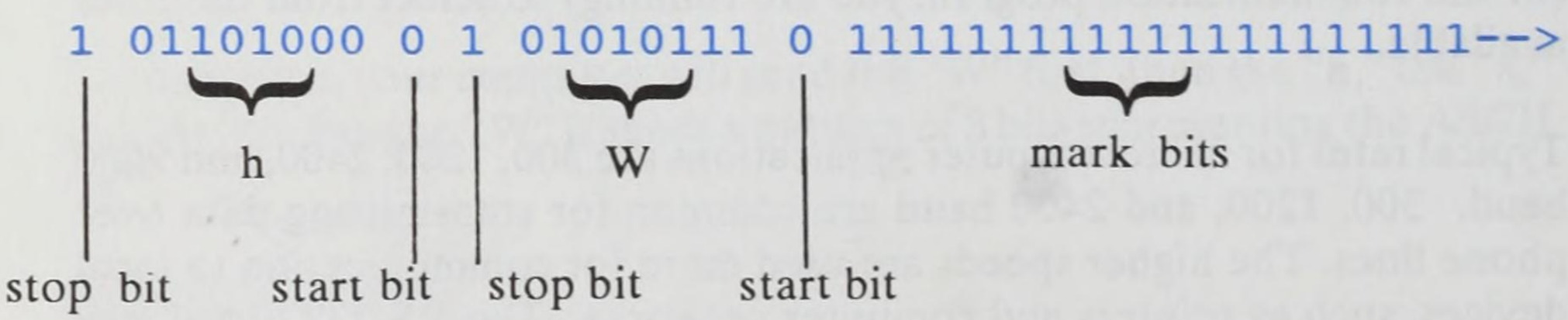
```
011111111111111111111  -->
```

This is the *start bit*. It tells the receiving device to get ready because the next 8 bits will be a character, in this case a "W."

```
01010111  -->
```

After the 8th bit of the character, a one-bit is sent, the *stop bit*. The start bit and stop bit "frame" the character. If a zero bit, rather than a one-bit, is received after the 8th bit of the character, the receiving device or computer knows that a transmission error occurred. This is called a *frame error*.

After the stop bit, more one-bits can be sent. The receiving device ignores them until it receives a zero-bit to signify the start of the next character. Thus, the bits sent for the "Wh" of "What's up Doc?" might look like this:



This particular scheme, or *protocol*, is designated "300 baud, 8 data bits, 1 stop bit, and no parity." A protocol is simply a scheme specifying the baud rate and how the data flow is to be organized.

Parity Bits

Just one incorrect bit can make a message meaningless, so error checking is critical. You've already seen one way that errors can be sensed. A start bit of zero and a stop bit of one must frame the 8 bits of the character. That is no assurance, however, that the 8 bits representing the actual character are correct.

For better error checking, it is common to send an additional bit with each character, called the *parity bit*. The communication software on the sending and receiving ends can be told to use *even* or *odd* parity.

For *even parity*, all the one-bits in the character are added together. If the result is an *odd* number, a one-bit is sent just after the character's data bits to make the total of the one-bits an even number. "W" has 5 one-bits. Count them:

```
01010111
```

For even parity, it is, therefore, sent as:

```
101010111  -->
```

Now there are 6 one-bits, and 6 is an *even* number. The receiving computer can also add and check the 9th bit to confirm that it is a 1.

Note: This method of error checking isn't perfect, but for the parity bit to be wrong, there would have to be 2 offsetting errors in the 8 bit byte. The odds against that happening are high.

The "t" in "What" already has an *even* number of one-bits:

```
01110100
```

so the 9th bit is sent as 0 to keep the total *even*:

```
001110100  -->
```

Odd parity works the same way, except a zero or one-bit follows each character to make an *odd* number of one-bits:

"W" is 001010111 -->
"h" is 001101000 -->
"t" is 101110100 -->

In computer-to-computer hookups, both sides must decide upon the same baud rate, number of data bits, and even, odd, or no parity before communication can take place.

Transmission to a serial printer isn't usually quite so critical. Some printers have switches that can select the type of parity checking to be used. Others don't accept parity bits; they just check for framing errors.

Stop Bits

At least one stop bit follows the parity bit (or if no parity is being used, the last data bit of each character). In setting up the communications protocol, you can usually specify either one or two stop bits.

One stop bit is normally sufficient. Sometimes two stop bits are used as a way of slowing down the data transfer just a little so the receiving device can keep up.

Data Bits

Eliminate the mark bits, start bits, parity bits, and stop bits, and what's left are *data bits*. Data bits represent the information being sent.

Communications can be conducted with 8 data bits per character or 7 data bits per character. A character is one byte and each byte is 8 bits, but 7 bits per character are sufficient to handle the "standard" 128 ASCII letters, digits, symbols, and control codes. ASCII text files can be sent with 7 bits per character.

If you are sending binary data such as program files, compressed numerical data, or graphics, 8 bits per character are required to use all 256 ASCII characters.

Copying to the AUX Device

Now that you are an expert in RS-232 boards, baud rates, data bits, parity, and stop bits, you are ready to explore the features that MS-DOS provides for communications.

At "gut" level, DOS only knows how to do two things with the communications ports. It can do aux *input* and aux *output*.

When DOS receives a command for auxiliary input, it waits for a character to come through COM1 or COM2, whichever is specified. When a character arrives, it is passed to the program that requested it. When a program gives DOS the command for auxiliary output, it responds by sending a single character out the specified communications port.

The **copy** command is the most direct way to put DOS communications to work. If you wish, fire up your system and try it. Although nothing is hooked to a port (if you have one), we can "fake" it. To get the idea, try some output. At the system prompt, type:

copy con aux ENTER
Send this out the communications port. ENTER
CTRL Z ENTER

DOS responds with `1 File(s) copied` or...

```
Write fault error writing device AUX
Abort, Retry, Ignore,
```

...to which you should answer A for abort.

How about some input? Type:

copy aux con ENTER

Unless something is coming into the communications port, you get:

```
Read fault error reading device AUX
Abort, Retry, Ignore?
```

Press A again for abort. In some cases, this command causes the system to "lock up," waiting for data. If this happens, press CTRL ALT DELETE to reboot the system.

Communications Mode

Were you impressed? Probably not. What's needed is something at the other end of the communications line and an agreement on the protocol.

The **mode** command allows you to set the baud rate, parity, number of data bits, and number of stop bits.

- The *baud rate* may be 110, 150, 300, 600, 1200, 2400, 4800, or 9600.
- The *parity* can be N, O, or E. N stands for none, O for odd, and E for even.
- The number of *data bits* can be 7 or 8.
- The number of *stop bits* can be 1 or 2.

To set the protocol for communications port 1 to 300 baud, no parity, 8 data bits, and 1 stop bit, use:

mode com1:300,n,8,1 ENTER

That sets the protocol.

Sending a File to Another Computer

mode works okay to **send** data to a printer or other device that doesn't have to talk back. But, the key to good communications between computers is having good communications programs in both computers. If you try to simply copy *to* the AUX port in one computer and *from* the AUX port in the other, it's often hard to coordinate the transfer. One computer tries to send before the other is ready to receive.

A communications, or *terminal*, program makes the job easier. To illustrate, let's see how an ASCII file can be sent from the Tandy 1000 TX

to the Radio Shack Model 100 Portable Computer, which has a built-in terminal program.

The first step is the physical connection. A cable goes from the RS-232 socket on the back of the Tandy 1000 TX to a null modem. The null modem plugs into the back of the Model 100.

Next, the communications protocols must be set the same on both computers. On the Model 100, select the TELCOM program, then type:

stat 38n1e ENTER

On the Tandy 1000 TX, type:

mode com1:300,n,8,1 ENTER

Now they're both set to 300 baud, no parity, 8 data bits, and 1 stop bit.

Press F4 on the Model 100 to go into terminal mode, then F2 to *download* data from another computer. The Model 100 asks what it should name the data to be received. Since this is just for practice, type **TEST** as the file name, and press ENTER.

The Model 100 is now ready to receive. Let's try sending it a batch file. Batch files are in ASCII, so they're appropriate for this demonstration.

Your MS-DOS command is:

copy autoexec.bat aux ENTER

As the file transfers, you can see it being displayed on the Model 100's screen. When the transfer is complete, the Tandy 1000 TX screen says:

```
1 File(s) copied
```

Press F2 on the Model 100 to end the download. Press F8 Y ENTER and F8 again. You'll notice a new file in the Model 100's directory called TEST.DO. Got it! *QSL*, as a ham radio operator would say.

Receiving Files

By far, the best way to *upload* into our DOS machines is to invest in a good terminal program. It allows communicating directly from computer to computer, or over the phone lines, and has provisions for setting the baud rate, so you don't need to use the DOS **mode** command. The Tandy *DeskMate* communication software works great!

Part of every good terminal program is an *upload* and *download* feature. Uploading means sending a file from your computer "up" to another. Downloading means receiving a file from another computer "down" to yours.

The CTTY Command

In most environments, input comes from the console. The **ctty** command allows you to set up a remote console, making it possible for someone in another room to operate the computer.

ctty aux gives input and output control to the device connected to communications port 1. That device returns control to the console when **ctty con** is typed.

With a Model 100 as a remote terminal and your Tandy MS-DOS computer as the *host*, you can give it a try.

Make sure the Model 100 is connected just as it was in the last example and that the communications parameters are the same. Go back into the TELCOM program. Now press F4 on the Model 100 to go into terminal mode.

Back on the Tandy MS-DOS computer, type:

ctty aux ENTER

On the Model 100, press the ENTER key and see the system prompt. (Press F4 for *full duplex* if you see doubled characters on your screen.) Now you can execute just about any MS-DOS command from the remote keyboard. Try **dir**, for example, to display a directory.

To return control to the main console on the Model 100, type:

ctty con ENTER

Again, a better way to do this is with *Professional DeskMate*'s HOST feature, or other good terminal software, instead of using DOS.

Using a Serial Printer

If you use a serial printer instead of parallel, the following will assign LPT1 to COM1 and set the baud rate to 9600, no parity, 8 data bits, and 1 stop bit.

```
mode com1:96n81
mode lpt1=com1
```

Chapter 22 Summary

The *RS-232 board* allows communications between your computer and the outside world. It handles *asynchronous serial communications*.

The RS-232 port may be connected to serial printers, external modems (for telephone communications), and other devices. For direct connection to another computer, a *null modem* is necessary.

For successful communications, both devices must agree on a *protocol*. The factors involved are *baud rate, parity, data bits*, and *stop bits*.

The *baud rate* (in practice) refers to the number of bits per second. A baud rate of 300 bits per second corresponds roughly to 30 characters per second.

Parity is a method of checking for communications errors. The types of parity are *odd, even*, and *none*.

The number of *data bits* is the number of bits per byte. For sending text and ASCII files, 7 data bits may be used. For sending binary files, 8 data bits are usually required.

Stop bits separate each character when sent over the communications line. Normally 1, but sometimes 2 stop bits are used.

A communications port may be the target for **copy**. Example: **copy myfile aux** sends the file MYFILE to the aux device.

The **mode** command can set the communications protocol. **mode com1:300,e,7,1** initializes the RS-232 board for 300 baud, even parity, 7 bits per character, with 1 stop bit.

The **ctty** command allows use of a remote console. **ctty aux** tells DOS to use the communications port for console input and output. **ctty con**, at the remote console, returns control to the "real" console.

PART 5

Personalizing your system

CHAPTER 23

Customizing the Prompt

They're not called "personal computers" for nothing. Do you want "Hi Boss!" in place of the usual system prompt? Or maybe you have a serious reason for using a special prompt, as shown in the next several chapters.

The PROMPT Command

We have seen the system prompt as `A>`, `B>`, or `C>`, depending on the current drive. Let's customize it. Try this:

prompt Now what? ENTER

MS-DOS responds with:

```
Now what?
```

The new prompt will let you display a directory, clear the screen, use **chkdsk**, or do anything else DOS can do, but instead of `A>`,`B>`, or `C>`, it prompts `Now what?`

Did you like `A>` better? How about a compromise. Type:

prompt Current drive = $n: Now what? ENTER

$n in a **prompt** command represents the current drive letter. The prompt becomes:

```
Current drive = A: Now what?
```

More options in a minute, but first we'd better learn how to return the prompt to "normal." Just type:

prompt ENTER

Ahh . . . that's better, but not as good as when we began this chapter.

You can make the prompt say almost anything, but certain characters have special meanings when preceded by a dollar sign:

$t	displays the time
$d	displays the date
$p	displays the current path
$v	displays the MS-DOS version
$n	displays the current drive name

To avoid confusing MS-DOS, several characters have codes to represent them when included in prompting strings:

$$	displays the $ character
$q	displays an equals sign
$g	displays the greater-than symbol
$l	displays the less-than symbol
$b	displays the pipe symbol
$h	executes a backspace
$_	executes a carriage return and line feed
$e	executes the escape (ASCII 27) code

Test a few:

prompt Date: $d Time: t_ng ENTER

Now the prompt looks like this:

```
Date: Sat 12-14-19xx Time: 10:15:52.73
A>
```

The date replaced **$d** and the time replaced **$t**. The dollar sign and underline cause a carriage return and line feed. The second line of the prompt is just like the default. **ng** displayed the current drive letter and the greater-than symbol.

Special effects are also possible.

prompt thhhhh$h ng CTRL G ENTER

You see hour and minute of time, followed by the current drive letter:

```
10:16 A>
```

$t displayed the time. The six **$h** codes caused quick backspaces over the seconds and hundredths. **ng** did the rest, except for Control-G which "rang the bell."

Another Idea for PROMPT

Here's an idea which can be elaborated on:

prompt Don't touch! I'll be right back. ENTER

Keep this one in mind the next time you must leave the room for a few minutes!

When in Doubt

You can experiment endlessly, but when in doubt, my favorite remains this useful, but uncluttered, prompt. It should already be in your AUTOEXEC.BAT file:

prompt=np$g

This prompt displays the drive, current directory with its path, and >.

Chapter 23 Summary

The **prompt** command customizes the system prompt. **prompt What's new?** replaces the default prompt (A>, etc.) with What's new?

Certain characters preceded by a $ have special meanings in a prompt string. **prompt $d** displays the current date.

CHAPTER 24

An ANSI.SYS Primer

When to Get ANSI

DOS has only limited capability when it comes to displaying text on the screen. The dialogue between you and DOS is on a line by line basis. When you reach the bottom of the screen, everything *scrolls* upward to make space for the next message or command.

DOS has no provisions for changing the screen color, highlighting words, or using blinking or reverse video. The keyboard arrow keys can't be used to move the cursor around the screen because DOS is unable to return to a prior line. Besides sending line feeds, carriage returns, and backspaces, about all DOS can do is display one character after another.

You have seen programs that do much more with the screen. They display colorful graphics, place messages here and there, and move the cursor to various locations. Strictly speaking, these programs are not *MS-DOS compatible*. True compatibility means you can take a program from one MS-DOS computer and run it on any other make or model of computer that uses MS-DOS.

IBM Compatible

More often, programs that do fancy screen work are *IBM compatible*. To move the cursor, change the color, or clear the screen, they bypass MS-DOS and give commands directly to the BIOS.

The BIOS on IBM compatible computers is designed to accept the same machine-language commands as the IBM PC. All modern Tandy MS-DOS computers are IBM compatible.

Programs sometimes bypass the BIOS and manipulate the hardware directly. For example, instead of giving the BIOS a scroll command, a program can actually move the information in video memory to make the lines on the screen roll upward.

ANSI Compatible

There's another level of standardization more widely accepted than either MS-DOS or IBM compatibility. The American National Standards Institute has developed a scheme of codes for controlling console input and output. Programs that conform are called *ANSI compatible*.

Under the ANSI system, positioning the cursor, setting text colors, and other functions are performed by sending a series of ASCII characters. For example, **escape** (ASCII 27), followed by **[31m** means, "Change the foreground color to red." **Escape** followed by **[1m** means, "Turn bold on." Subsequent characters will be highlighted as they are typed. **Escape** plus **[5;8f** means, "Move the cursor to line 5, row 8" on the screen.

These ANSI commands all start with 2 characters, **$e** (the code for **escape**) and **[**. When displaying data, a system that is ANSI compatible "watches" for these 2 characters because the characters that follow them give information about what is actually to be done.

Many *ANSI escape sequences* have been agreed upon. Besides controlling the video display, they can also redefine keys on the keyboard. In screen handling, they provide much more power than DOS alone, but not quite as much flexibility and speed as can be achieved by going directly to the BIOS. Many program developers have chosen to make their programs ANSI compatible in order to reach a larger market.

Installing ANSI.SYS

To make your system ANSI compatible, you must install ANSI.SYS. With ANSI.SYS in the DOSMISC directory, add this line to your CONFIG.SYS file:

device=dosmisc\ansi.sys

DOS will install ANSI.SYS each time you boot. Press CTRL ALT DELETE to install it now.

ANSI.SYS enlarges DOS by about 5000 bytes, occupying that much more memory. If you are short on memory, you may need to remove ANSI.SYS when running programs that need the extra space.

Setting the Display Color

In the next few sections, we will refer to display *colors*, but that doesn't mean you are left out if you have a monochrome monitor. On some monitors shades of gray represent different colors. Other monitors can't display shades, but you will be able to use highlighting, reverse video, blinking, and in certain cases, underlining.

The most difficult part about entering ANSI escape sequences from the keyboard is that DOS intercepts the escape key. When you press ESC, DOS displays a backslash and assumes you want to cancel the line. Fortunately, there's another way to send it.

When **$e** is used with **prompt**, it produces an ASCII 27, the code for *escape*. Type:

prompt $e[7m ENTER

then:

prompt ENTER

Now press ENTER a few times, or request a directory. If everything went right, all new text on the screen is in reverse video, black on white. The sequence **$e[7m** is for reverse video. Change it back to normal:

prompt $e[0m ENTER
prompt ENTER

The coding **$e[0m** is for white on black. Try:

prompt $e[1m$nge[0m ENTER

Now the system prompt is highlighted, and everything else is normal. (Press a few keys to check it out.)

Study the **prompt** command and you'll see how it was done. The highlighting was turned on by **$e[1m**, the drive name and greater-than symbol displayed by **ng**, and the text color changed back to normal by **$e[0m**.

Color Escape Sequences

Here are the escape sequences for setting colors on the display. Don't forget to precede and follow them with **prompt**:

$e[0m	Normal white on black
$e[1m	Bold or high intensity
$e[4m	Underline (if available)
$e[5m	Blink
$e[7m	Reverse Video
$e[8m	Concealed (invisible)
$e[30m	Black foreground
$e[31m	Red foreground
$e[32m	Green foreground
$e[33m	Yellow foreground
$e[34m	Blue foreground
$e[35m	Magenta foreground
$e[36m	Cyan foreground
$e[37m	White foreground
$e[40m	Black background
$e[41m	Red background
$e[42m	Green background
$e[43m	Yellow background
$e[44m	Blue background
$e[45m	Magenta background
$e[46m	Cyan background
$e[47m	White background

The codes can be combined. Simply put one number after another, separated by a semicolon. **$e[34;42m**, for example, displays blue characters on a green background. **$e[1;33;41m** displays bright (high intensity) yellow characters on a red background.

Setting Colors with Batch Files

The escape sequences are difficult to remember. You might want to make special batch files to set your favorite colors. This one changes the color to white on blue whenever you type **blue**:

copy con blue.bat ENTER
prompt $e[44;37m ENTER
prompt ENTER
CTRL Z ENTER

Note that there is no **echo off** statement in this batch file. If **echo** was **off**, the prompt wouldn't be displayed, and the ANSI driver wouldn't get the message to change the color.

Making Color Files

The **prompt** command works well for setting colors. An alternative method is to create files that contain the escape sequences, then execute them with the **type** command. An easy way is by using the BASIC programming language. Give it a try.

If you deleted BASIC, put your *system master* containing BASIC in Drive A. Now everyone, type:

basic ENTER

If you took your *system* disk out, put it back in Drive A. At the `Ok` prompt, type:

open "o",1,"reverse" ENTER
print #1,chr$(27);"[7m" ENTER
close ENTER
system ENTER

This sequence of commands created a file called REVERSE. Now whenever you want reverse video simply:

type reverse ENTER

Create a similar file called NORMAL. To do it in BASIC, enter the same commands, replacing the word **reverse** with **normal** and **[7m** with **[0m**. (To do it with a BAT file, use codes 40 and 37.)

Redefining Keys

The ANSI driver also lets you redefine keys on the keyboard. For example, F1 can display your name and F2 can type a password. You can even make the A key into a B key if you want!

The escape sequence for redefining keys starts with **$e[** as usual. Then you add the code for the key you want to redefine, plus a semicolon, then the replacement string. The escape sequence ends with a small **p**. It's a bit complicated, so let's try one. The code for F1 is **0;59**. Let's make F1 the *name* key. Carefully type the following, inserting your own name between the double quotes:

prompt $e[0;59;"Your name**"p** ENTER
prompt ENTER

Press F1. Now it's *really* a personal computer!

With a slight change, you can put an automatic ENTER at the end of the name. Do it by inserting **;13** before the **p**:

prompt $e[0;59;"Your name**";13p** ENTER
prompt ENTER

Key redefinitions can be clustered and saved in batch files. Here's one that customizes ALT F1 through ALT F4 to execute automatic **dir, copy, erase,** and **type** commands.

copy con newkeys.bat ENTER
prompt=$e[0;104;"dir "p ENTER
prompt=$e[0;105;"copy "p ENTER
prompt=$e[0;106;"erase "p ENTER

prompt=$e[0;107;"type "p ENTER
prompt ENTER
CTRL Z ENTER

Just type **newkeys** ENTER, and these *soft keys* are redefined to display these specific tasks. Press ALT F1 ENTER to execute the **dir** command.

You can add other redefinitions if you want, but you need to know the key codes. They are found in the factory MS-DOS manual. The F-keys and arrow keys have what are called "extended codes" that consist of a 0 and another number. When using them in ANSI escape sequences, a semicolon serves as a separator.

The other keyboard keys are represented by their ASCII code. "A," for example, is 65 (see the table in Appendix C).

Other ANSI Escape Sequences

Other ANSI codes are available, but it takes some programming knowledge to use them. A few are listed here, in case you're interested. The # signs are to be replaced with the indicated parameters.

$e[#;#H	moves cursor to line #, column #
$e[#A	moves cursor up # lines
$e[#B	moves cursor down # lines
$e[#C	moves cursor forward # columns
$e[#D	moves cursor backwards # columns
$e[2J	clears the screen
$e[k	erases to the end of the line
$e[s	saves current cursor position
$e[u	goes back to the "saved" cursor position

Reboot when done.

Chapter 24 Summary

ANSI.SYS is a device drive that extends the video display and keyboard handling capabilities of DOS. It is installed by including **device=ansi.sys** in the CONFIG.SYS file.

The main advantage of ANSI.SYS is that it provides screen and keyboard handling compatibility for many makes and models of computers. Common operations, such as cursor movement, color changing, and keyboard redefinitions, can be handled by sending standardized sequences of characters to the video display.

The **prompt** command followed by ANSI escape sequences performs these operations. **prompt $e[7m** ENTER **prompt** ENTER sets the screen to display in reverse.

CHAPTER 25

Controlling the Computer's Mode

The **mode** command is a multi-purpose program for customizing your system to specific requirements. Control of the video display, line printer, and serial communications are within the scope of **mode**.

Since **mode** is an *external* command, MODE.* (or use the MODE program) must be in the current directory on the current drive, or the path must be set so DOS can find it.

Video Display Width

> Because of the wide variety of monitors and video display boards, not all commands work with all computers.

Let's start with the easiest settings. Type:

mode 40 ENTER

The screen clears, and the system prompt appears in the upper-left corner, twice as "fat." Display a directory and see that 40 characters, instead of 80, are displayed on each line. If you think this is easier on the eyes, use **mode 40** from now on. Otherwise, change it back:

mode 80 ENTER

40 and 80 are the only width options. The output from most programs supplied with DOS is designed so the width setting doesn't matter, but a 40-column display is often confusing.

> Try **mode 40** ENTER, then **dir /w** ENTER. That's really confusing!

Other programs may set the width to 80 or 40 columns without your having any say in the matter. When you exit them, the width is usually restored according to your original mode setting.

A Little Demonstration

For a demonstration of the difference in color modes, disable color, if necessary, with:

mode bw ENTER or **mode bw80** ENTER

and go into BASIC:

> If you don't have BASIC on your *system* disk, use the *system master* as we did in the last chapter.

basic ENTER

Below BASIC's `Ok` prompt, type:

for x=1 to 7 : color x,0 : print "color";x : next ENTER

In each of 7 different colors (or shades of gray) you see:

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

Return to DOS with:

system ENTER

That's the first half of the demo. Remember what you saw. Now do it with color enabled:

mode color ENTER or **mode co80** ENTER

Re-enter BASIC and type the same command line. If you have a composite video monitor, you'll notice a difference in the way the colors are displayed. They probably aren't as sharp. Return to DOS.

Remember this **mode** option next time you have difficulty reading text on a monochrome or color screen. But be aware that **mode** won't always help because many COM and EXE programs set the mode themselves, overriding anything you do at the DOS level.

In other cases **mode** may be useful after exiting a program. Some programs change the size or flashing rate of the cursor and do not restore it upon return to DOS. Use **mode bw 80** to restore the video display and cursor to normal.

Centering the Display (For the 1000 Series)

Type:

mode 40,r,t ENTER

The display shows:

```
012345678901234567890123456789012345678 9
```

```
Do you see the leftmost 0? (Y/N)
```

Just for practice, press N for no. The entire display shifts right one position. Press N again if you want. It shifts again. Now press Y and the DOS system prompt returns.

The **t** in the **mode** command requests the "test pattern." The **r** requested a shift to the right. Try the opposite adjustment:

mode 40,l,t ENTER

to move the screen's contents to the left.

Centering 80-column displays can be done with **mode 80,l,t** and **mode 80,r,t**.

Special Tandy 1000 Modes

Some Tandy 1000s have two additional **mode** options. When using a television set for a monitor, use:

mode tv ENTER

This sets the width to 40 and makes other changes for a clearer picture.

You can also substitute colors:

mode colormap blue,green ENTER

Everything normally blue will display in green.

The available colors are: black, blue, green, cyan, red, magenta, yellow, gray, dark gray, light blue, light green, light cyan, light red, light magenta, light yellow, and white. To restore the colors to normal, type:

mode colormap ENTER

Printer Mode

This one is from the "be prepared" department. You may run a program that makes a printout, only to find that it is double spaced when it should be single spaced, or it is printing without advancing the paper.

The printer must make two movements for each new line on a printout. To move the print head back to the left-hand column, it must execute a *carriage return*. To move the paper up to print on the next line, it must execute a *line feed*.

Here's the controversy: should the computer send the printer one signal or two? One point of view is that an ASCII 13 should be sufficient to request a carriage return *and* line feed. The other opinion is that the computer should send ASCII 13 for carriage return, then ASCII 10 for a line feed.

Most modern printers can behave either way. They have switches to enable or disable automatic line feeds after carriage returns. The problem usually occurs when using the same printer with different computers, or with one program that sends line feeds and another that doesn't.

The **mode** command has two line feed options:

mode lfoff ENTER

causes line feeds to be suppressed when preceded by carriage returns. Use it if you are getting double spacing, but only want single. If the computer responds with:

```
Loadable printer driver required
```

Type:

lf ENTER

and then continue with the **mode** command.

The opposite command is:

mode lfon ENTER

Use this when the printer is not receiving line feeds.

Either option can be included in AUTOEXEC.BAT to set the printer mode each time you boot up. If you have a program that needs a line feed setting different from other programs, include the **mode** command in a batch file that runs the program.

Communications Mode

mode has another purpose. Setting the *protocol* for the RS-232 serial communications board, as explained in Chapter 22.

Chapter 25 Summary

The **mode** command has several purposes. To change the video display width to 40 characters per line, use **mode 40** ENTER. To return to 80 characters across, use **mode 80**.

You can also request **mode color** or **mode bw**. Depending on the video display and the type of work you are doing, one setting may be easier on the eyes.

On some Tandy 1000s, MODE accepts **tv** in place of **bw** or **color** when a television is being use as a monitor. A **colormap** option allows color substitutions.

mode lfoff or **mode lfon** can be used to correct problems with vertical spacing on a line printer.

Customizing Drives and Files

A Little Protection with ATTRIB

Using the **attrib** command with its **r** option adds a little protection to certain files, making them "read only". For example, pretend our MEMO.TXT file is very important. By using ATTRIB, you can make this file read only; information in the file can only be read by a program or by the DOS **type** command. You cannot **delete** the file or change it in any way. For example, typing:

attrib +r \examples\memo.txt ENTER

sets the file MEMO.TXT to read-only status. The **type** command can still be used, but if you try to delete the file:

delete \examples\memo.txt ENTER

DOS responds:

```
Access denied
```

To remove the read-only status, allowing the file to be erased or changed, use:

attrib -r \examples\memo.txt ENTER

Wild cards can also be used with the **attrib** command. To set read-only status for every TXT file in the EXAMPLES directory, you would type:

attrib +r *.txt

Likewise, to remove read-only status, use:

attrib -r *.txt

A second **attrib** option, **a**, controls a file's attribute, or modification, bit. This bit is turned on each time a file is modified. The **backup** and **restore** commands use this bit to tell if a file has been modified since the last back-up. Turning the bit on or turning it off affects how a program or file is backed up. Use:

attrib +a \examples\memo.txt

to turn the bit on (meaning the file has been modified since the last back-up). Or use:

attrib -a \examples\memo.txt

to turn the bit off. DOS will not backup the file once **-a** is used, and only modified files are specified to be backed up.

attrib's **a** option should only be used when selectively backing up files from a hard drive. Otherwise, there's no reason to change attribute bits.

All DOS 2.11 users can go on to the next chapter.

Switching Drive Designations

assign is an *external* DOS command which "switches" the disk drive letters in any filename. For example:

assign b=c

reads roughly, "Convert all references to Drive B to read Drive C." Inside the computer, DOS searches for filenames specified Drive B and changes the drive to C. For example, the file named:

b:scores

is changed to:

c:scores

The reason for the **assign** command dates back to the IBM PC/XT. It came with only Drive A and hard Drive C. At the time, many software packages insisted on using Drive B, even though the XT had none. To solve this problem, you had to assign all references from Drive B to Drive C:

assign b=c

If a computer has 2 floppy drives, you can redirect everything from either floppy disk to Drive C using:

assign b=c a=c

To undo the changes made by **assign**, simply reassign a drive to itself:

assign b=b

Using **assign** with no drives specified resets everything back to normal.

Some Warnings

Plain and simple, if you don't need **assign**, don't bother with it. There are a few DOS commands which may become totally confused when you reassign drives. Even **copy** can do dastardly things if the **assign** command is used improperly.

SUBSTituting a Drive for a Pathname

The **subst** command provides a neat way for dividing a hard drive (or floppy). It lets you treat a subdirectory as if it were an independent disk drive. This is useful because you can specify a drive letter instead of a lengthy pathname.

Go to the root directory of Drive A on your *system* diskette, and type:

subst b: a:\doscmds ENTER
dir b: ENTER

If you have a "real" Drive B, it's now sitting idle. DOS shows the directory listing of the DOSCMDS subdirectory on Drive A instead. Make this new B the active drive:

b:

and **dir** again. Even though **dot** and **dot dot** are listed, the substitution makes DOS think we're in the root of B. Try:

cd .. ENTER

Invalid directory, because we're in the root. Try:

**cd ** ENTER
dir ENTER

We're still in imaginary Drive B. The other files on the diskette are not available unless we change to A.

a: ENTER
dir ENTER

By the way, you can still access DOSCMDS as a subdirectory of A. Try:

cd doscmds ENTER
dir ENTER

and

cd .. ENTER

This time it works.

Without parameters, **subst** lists the substitutions:

subst ENTER

```
B: => A:\DOSCMDS
```

In actual practice you can make several drive substitutions. They'd all be listed.

To remove a substituted drive, use the drive letter with **/d**. Do it now:

subst b: /d ENTER

The system is back to normal.

A good way to issue the **subst** commands is within AUTOEXEC.BAT. This way your most important subdirectories can have their own drive letters.

LASTDRIVE

You may prefer not to use A, B, C, or D for substitute drive letters because these letters apply to real drives. To use other letters, modify the CONFIG.SYS file so it contains:

lastdrive=h

Upon rebooting, DOS will allow you to use E, F, G and H (in addition to A-D) for drive substitutions. Use another letter (up to Z) with **lastdrive** if you need more, but remember that each additional letter uses more memory.

JOIN

The **join** command is entered like **subst**, but it does the opposite. It allows a name to be used in place of a disk drive letter.

With the *system* disk in Drive A, and the `A>` prompts showing, try this:

join c: mary

If you don't have a Drive C, insert a formatted diskette into, and use **b** instead. The **join** command is of no value to single drive users.

Now try to use Drive C:

dir c: ENTER

```
Invalid drive specification
```

It's invalid because the drive's name is now MARY. Type:

dir mary ENTER

There it is, a directory of Drive C. Make it the current directory:

cd mary ENTER

and **dir** again if you wish.

Now change back to Drive A for another directory:

a: ENTER
dir ENTER

Notice something new? MARY has been added as a subdirectory on Drive A. That's how DOS remembers the names you've joined. Try:

join ENTER

and see:

```
C: => A:\MARY
```

join can use any valid subdirectory name. You may include the drive and path symbols (in this case, **a:**\) to control where the subdirectory is added. It's OK if the subdirectory already exists when you **join**, but it must be empty. Otherwise, DOS displays:

```
Directory not empty.
```

To return things to normal, enter the **join** command with the drive letter and **/d**:

join c: /d ENTER

In practice, floppy diskette drives and virtual disks are most often the targets for **join** commands.

Chapter 26 Summary

The **attrib** command has two options. **attrib +r anyfile** sets the file named ANYFILE to read-only status. It cannot be erased or changed. **attrib -r anyfile** removes the read-only protection. **attrib +a anyfile** turns on ANYFILE's attribute bit, and **attrib -a anyfile** turns it off.

The **assign** command switches drive designation letters. **assign b=c** replaces all Drive B designations with C's.

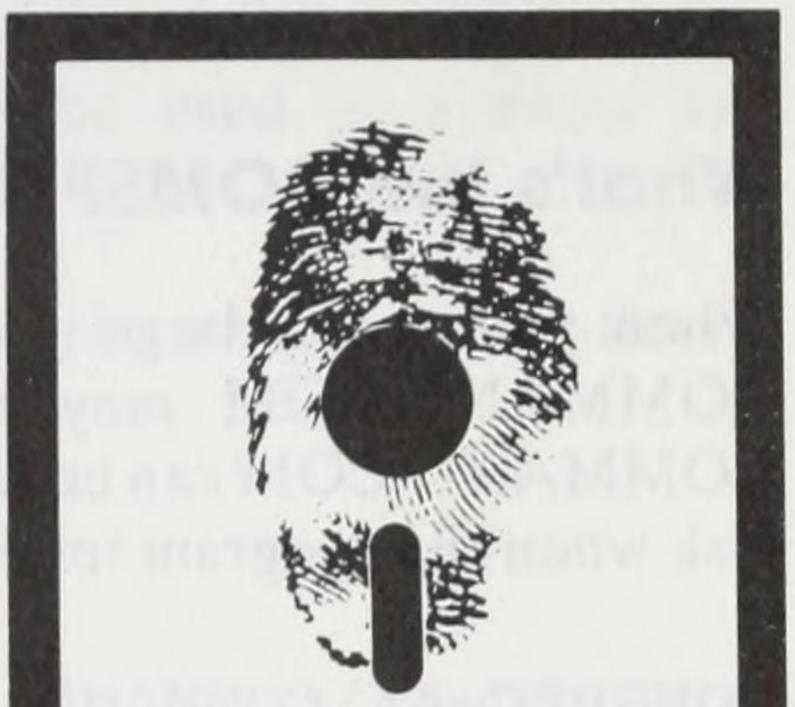
The **subst** command substitutes a drive letter for a path. **subst d: c:\finance\budget** substitutes **d:** for the BUDGET subdirectory. Enter **subst** without parameters to list the current substitutions. Use the drive letter and **/d** to remove a substitution: **subst d: /d**.

A **lastdrive=** entry may be included in CONFIG.SYS to make more drive letters available for substitution. **lastdrive=z** allocates 26 legal drive letters.

The **join** command allows a pathname to be used in place of a disk drive letter. **join b: c:\wpfiles** allows access to Drive B only by using the WPFILES subdirectory on C. The joining can be canceled with **join b: /d**. Without parameters, **join** lists all drives and subdirectories currently joined.

CHAPTER 27

The Environment



The SET Command

DOS has an internal "message board" where programs can post notices to one another. Let's take a look.

At the system prompt, type:

set ENTER

The **set** command reports what's on the "message board." It only "remembers" what was "posted" since the last boot, or change. Each message is called an *environment string*, part of the system's overall environment. At least 2 are always present, COMSPEC and PATH.

Let's enter a path:

path a:\;b: ENTER

Type **set** ENTER again to see what the environment looks like:

```
COMSPEC=A:\COMMAND.COM
PATH=A:\;B:\
```

COMMAND.COM will always be at the root, and the path we specified is there.

What's the COMSPEC?

When you run a large program, some of the memory cells used by COMMAND.COM may be overwritten. COMSPEC tells where COMMAND.COM can be found, so it may be automatically reloaded from disk when the program terminates.

```
COMSPEC=A:\COMMAND.COM
```

tells DOS to look in the root directory of Drive A.

The PROMPT String

Besides the PATH and COMSPEC strings, the environment also holds the current **prompt** setting, which we studied earlier. When it needs to know what prompt to display, DOS searches the list of environment strings until it finds the *prompt string*. If no entry is present, it simply defaults to the >.

To remove the PROMPT string from the environment and confirm the change, type:

prompt ENTER

and

set ENTER

Custom Environment Strings

The environment can hold other strings.

set user=Joe ENTER

set printer=DWP510 ENTER

Use **set** ENTER to see them.

user and **printer** are *parameter names*. **Joe** and **DWP510** are *replacement parameters*. Any string of characters may be used as a name and replacement.

To change an environment string, enter the parameter name and the replacement:

set user=Carol ENTER

Now **user** is **Carol**.

Unsetting

The strings in the environment are in memory only, not on disk. When you turn the computer off, they are lost. To remove a string during a session, just type the parameters name and an equal sign.

set printer= ENTER

The PRINTER environment string is gone.

A Quick Note Pad

DOS uses the environment as a message board for things *it* must remember. How about using it for something *you* want to remember?

set Jim's phone number = 555-3323 ENTER

Just type **set** ENTER to redisplay it:

```
JIM'S Phone Number = 555-3323
```

This may be handy the next time you can't find a pencil. Just don't turn the computer off!

Replacements in Batch Files

A more practical use for custom environment strings is in batch files. Batch files can hold new string settings and can replace values in existing parameters. Enter this one:

copy con hello.bat ENTER
echo off ENTER
cls ENTER
echo Hello %user% ENTER
CTRL Z ENTER

A string between percent signs indicates a replacement is to be made. In this case, **%user%** is to be replaced. Type:

hello ENTER

and see:

```
Hello Carol
```

Remember Carol? She is the user.

Hard Drive or Floppy

Suppose you are writing batch files to make it easier for inexperienced operators to run your business programs. There are several computers in the office. Some have hard drives, others don't, but you want the same batch files to work on all of them. Use a batch file like this, and call it HARDINFO.BAT.

```
echo off
cls
if %comspec%==A:\COMMAND.COM goto floppy
set drive=hard
goto end
:floppy
set drive=floppy
:end
echo I booted from the %drive% disk.
```

Study it carefully. If `%comspec%` is `A:\COMMAND.COM`, HARDINFO.BAT assumes the user is running a floppy disk system, so it jumps to `:floppy` and sets a new parameter in the environment. Otherwise it assumes the user has a hard drive.

On hard drive systems, `drive` equals `hard`. On floppy disk systems, it equals `floppy`. The final line of the batch file uses `%drive%` to display the message:

```
I booted from the floppy disk.
```

or:

```
I booted from the hard disk.
```

The other batch files to be run in the same session can also be made smart enough to display the right message, depending on the type of system. With floppy disks, it's sometimes necessary to stop for a replacement. A batch file might use the following lines:

```
if %drive%==floppy echo Please replace your data disk.
if %drive%==floppy pause
```

This results in:

```
Please replace your data disk.
Strike any key when ready...
```

...but only when the environment contains `drive=floppy`.

Blanks and Capitals

Watch your blanks and capitalization carefully when using the **set** command. To the left of the equal sign, blanks are significant, but capitalization doesn't matter. To the right of the equal sign, blanks and capitalization must be used consistently for proper matching by batch file **if** statements.

For example, **set drive=floppy** is not the same as **set drive = floppy**. In the first case, you can use **%drive%** for replacements in a batch file. In the second case, you must use **%drive %** (note the space).

On the other hand, **set drive=FLOPPY** is the same as **set DRIVE=FLOPPY**, but not the same as **set drive=floppy**.

Programs and the Environment

Environment strings may also be found in application programs. When the program is loaded, DOS assumes the parameters specified for the new environment. When the application program terminates, the original environment strings resume control.

If you purchase a program that uses environment strings, the instructions should tell you what **set** commands are needed. Or it may come with batch files that set the environment automatically.

Reboot the computer.

Chapter 27 Summary

The *environment* is an information list that stores current details about the system during a session.

Two strings are always in the environment, `COMSPEC=`, which tells where to find COMMAND.COM, and `PATH=`, which stores the current path setting. If a **prompt** command has been entered, a `PROMPT=` entry in the environment holds the PROMPT string.

The **set** command lists the environment strings, and adds new ones. **set printer=serial** adds `PRINTER=serial` to the environment.

Rebooting the computer will return the original environment strings.

Batch files may use environment strings. When DOS finds **echo %printer%** in a batch file, it searches for the PRINTER string in the environment list and makes a replacement.

CHAPTER 28

Country Customizing and Networking

Date, time, and money amounts are expressed differently in different countries. Languages other than English often require specially accented characters. MS-DOS can accommodate many of these differences.

To indicate a country other than the United States of America, include a code number (usually the international telephone prefix) as an entry in the CONFIG.SYS file. Here's an example:

country=049

When the system is rebooted, DOS will use the date and time format for Germany. You'll see the differences when date and time are requested and in the date and time columns of the directory display.

In application programs, the date, time, and currency formats might not change. Only those programs which have been designed to ask DOS for the country-dependent information are affected. The following country codes are recognized by DOS version 3.x and others as noted:

United States	001*	Denmark	045**
Canada (French)	002	Sweden	046**
Netherlands	031**	Norway	047**
Belgium	032**	Germany	049*
France	033*	Australia	061**
Spain	034**	Portugal	351
Italy	039**	Finland	358**
Switzerland	041**	Middle East	785
United Kingdom	044**	Israel	972**

* DOS version 2.11 and higher.
** DOS version 3.1 and higher.

(DOS version 2.11 users may skip to the next chapter.)

Keyboard Drivers

Several keyboard drivers are provided with DOS 3.x. They change the keyboard configuration and screen displays. To list them:

dir \doscmds\key*.com ENTER

KEYTCF.COM for Canadian French
KEYTFR.COM for France
KEYTGR.COM for Germany
KEYTIT.COM for Italy
KEYTSP.COM for Spain
KEYTUK.COM for United Kingdom

For an example, try:

keytgr ENTER

which changes the keyboard to the German format. Press a few keys to see the difference.

The :; key becomes ö, ' key displays ä, and [{ key displays ü.

The + = key has become a special "dead key." Press it, then A, E, I, O, or U to type the accented version of the letter.

A modified keyboard can cause compatibility problems with some programs, so special procedures are available to switch the "U.S." keyboard in and out:

On the 1000 Series keyboard, use the *left* SHIFT key instead of the ALT key indicated below.

CTRL ALT F1 selects the U.S. layout.
CTRL ALT F2 returns to the default layout you selected.

To run some programs, it may be necessary to use the keyboard driver with the /**us** option. For example:

keytgr /us ENTER

SELECT.COM

SELECT.COM is a special program that will do the country setups for you. Try:

select 033 ENTER

The date, time, and currency formats are now changed for France, but the keyboard has not been changed. Use **dir**, **date**, or **time** to see the differences.

With the addition of a keyboard code (**us**, **cf**, **fr**, **gr**, **it**, **sp**, or **uk**), **select** makes a copy of the DOS disk, adds a CONFIG.SYS file with a **country=** entry, and an AUTOEXEC.BAT file that runs the appropriate KEY*.COM program. If you wish to give it a try, here's an example:

(Place the *system* diskette in Drive A, and have a blank formatted diskette ready.)

select 044 uk ENTER

or:

select 044 uk /us ENTER

On completion, boot with the new *system* diskette that was just created.

With the keyboard modified to your needs, delete the SELECT.COM and the KEYT*.COM programs you are not using from this working copy of the *system* disk.

DOS and Local Networks

Hooking computers together so they can share the same disk files and devices, such as printers, is called *networking*. You need cables and special components in each computer to make the connections, plus additional software to control how the data flows. A system like this that doesn't rely on telephone communications is often called a *local area network* or LAN.

DOS does *not* include the software required for networking. But with version 3.2 and later, it provides accessory features which link to LAN software. Some of these features may be useful to you even if you don't use a network.

File Sharing

The main problem in a network operation is file sharing. Suppose two users want to access the same inventory file. User A brings up "widgets" on his screen and sees that 1,000 units are available, so he sells them. But what if User B is posting a sale of 500 widgets at the same instant? The problem goes beyond being out of stock and not knowing it. The file becomes corrupted with incorrect data.

The solution is *file locking*. Make all other users wait when one user is actively changing data in a shared file. Once the data has been changed, the file is unlocked, and other users have access to it. An entire file may be locked for a particular user, or parts of a file may be locked for multiple users. How and when is up to the programmer.

If files are to be shared, the programs that use them should be designed with that capability in mind. If not, it is up to the network users to make agreements to avoid conflicts.

To handle file locking, DOS needs to set aside space for remembering which files and which records within them are locked. SHARE.EXE does the

preparation. It should only be executed once in a session, so a **share** command is normally placed in the AUTOEXEC.BAT file. Here's an example:

share /f:1024 /l:10

The number after **/f:** tells how much space to reserve for tracking the file and pathnames being shared. With each pathname for an open file, DOS needs an extra 11 bytes, so assuming an average pathname of 20 bytes, this example allows about 33 files to be shared. ([11+20]*33=1023.)

The number after **/l:** defines the number of locks. In this case, up to 10 files or records may be locked simultaneously.

When **share** is used without parameters, the defaults are 2048 bytes for file information and 20 locks.

File Control Blocks and Networking

As you remember, CONFIG.SYS may contain a **files=** entry. The number tells how many files can be open at once. In a network system, you'll probably want to make the number bigger because more users means more files in use at the same time.

There's one other consideration to be aware of. In the early days of DOS, the programmer had to define a File Control Block, an area in memory which DOS used to maintain the file's statistics in order to open a file. Since version 2.0's release, programmers have had the luxury of simply giving DOS the file and pathname. DOS returns a file handle number, and the program can take actions with the file by referring to its handle. The areas for maintaining these statistics are managed automatically.

Many application programs still use File Control Blocks (FCB's for short). The advantage is that CONFIG.SYS's **files=** entry doesn't matter because the program creates its own FCB's as needed. With networking, however, DOS needs some help to make these programs work properly. It needs space to maintain additional information for each open file.

Application program instructions usually don't indicate whether they've used FCB's for file handling, so you may need to do some testing on your network, watching for error messages.

To avoid the problem, DOS allows for a special entry in the CONFIG.SYS file. Here's an example:

fcbs=10,4

The first number may range from 1 to 255. The second may be from 0 to 255, as long as it is not greater than the first. In the above example, DOS is told to allow for **10** files opened by the FCB method. In the event more files are opened, DOS is given permission to close the least recently used file, if it is *not* one of the first **4** files that were opened.

Unexpected closing of files can cause error conditions, so you may want to make the second number equal to the first to insure that none are closed. Lacking an **fcbs** entry, DOS uses 4 and 0 as the defaults: 4 files can be opened by FCB's at once, and all are subject to automatic closing.

Chapter 28 Summary

DOS can be customized to accommodate international differences. A **country=** entry in the CONFIG.SYS file loads country-dependent information. Programs designed for international use can ask DOS for the country code and the applicable date, time, and currency formats.

The **select** command changes the date, time, and currency formats, plus the keyboard and screen displays, to those of a country designated by a code. **select 041** changes the formats to Switzerland's.

Several programs are supplied on the DOS 3.x disk for modifying the keyboard. For daily use, the applicable KEY*.COM program can be run at startup by AUTOEXEC.BAT. When a special keyboard driver is installed, it may be necessary to press CTRL ALT F1 (or CTRL *left* SHIFT F1) for compatibility with some programs. CTRL ALT F2 (or CTRL *left* SHIFT F2) reactivates the selected country's keyboard driver.

To return to the default (U.S.) date, time, currency, and keyboard formats, remove the **country=** line from CONFIG.SYS and the **key*** command from AUTOEXEC.BAT, then reboot.

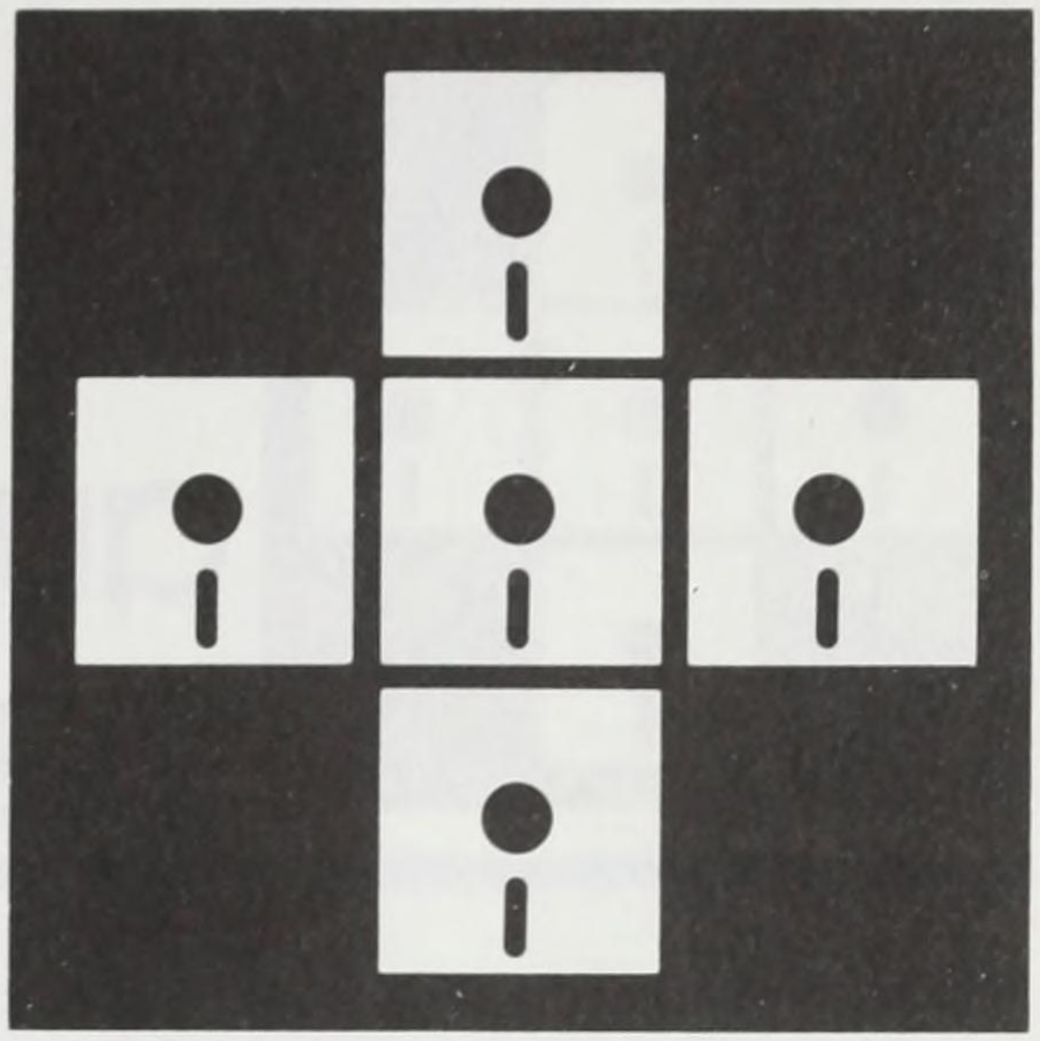
DOS version 3.2 and later has features that enable networking when additional hardware and software are added.

The **share** command sets aside workspace in memory so DOS can track which files are locked and unlocked. When a file or a portion of a file is locked, it is available only to the user whose program locked it. When unlocked, it becomes available to other users in the network.

SHARE.EXE may be run only once in a session, after which, DOS checks all read and write requests.

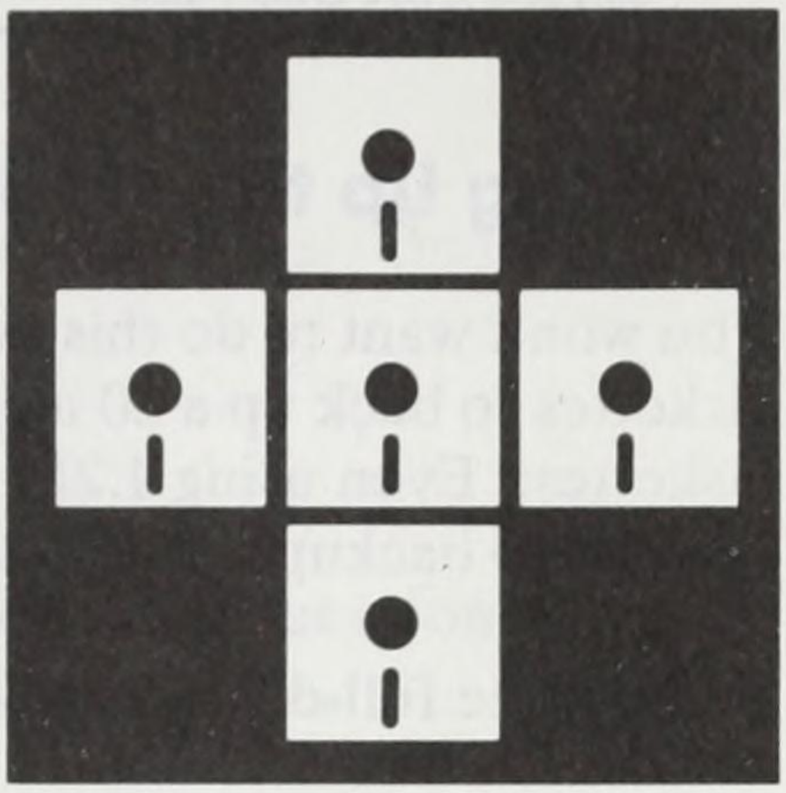
With networks it is sometimes necessary to include an **fcbs=** entry in the CONFIG.SYS file. When programs employ the *File Control Block* method of handling files, DOS needs work space to maintain the statistics. **fcbs=12,12** reserves space for 12 control blocks, and none of the 12 files may be closed by DOS to make room for others.

PART 6



Keeping your system healthy

CHAPTER 29



Backup Strategy

Backup or duplicate copies of files are insurance against loss, but no one insurance policy is right for everyone. Most computer users combine a variety of backup methods.

One method of protection is to keep 2 versions of a file on the same disk by simply copying and renaming. The use of a subdirectory dedicated to holding copies of files under their original name is similar. Both techniques have merit, but neither is totally satisfactory. Some programs, such as **edlin**, automatically create a BAK file of the original, prior to editing.

For better protection, backup files should be kept on separate diskettes. Floppy-only users can make backup disks with **diskcopy**. Those with hard drives can use the **backup** command.

Non-hard drive users can skip to the section titled "XCOPY," later in this chapter.

BACKUP and RESTORE (Hard Drive Only)

backup copies files from a hard drive onto one or more floppy diskettes.
restore copies files from the floppy diskettes back onto the hard drive.

BACKUP.COM and RESTORE.COM may be supplied on a Supplemental Program disk or Hard Drive Utility disk.

Backing Up the Whole Hard Drive

You won't want to do this every day. It takes over 4 dozen standard 360K diskettes to back up a 20 megabyte hard drive, or more than 2 dozen 720K diskettes. Even using 1.2M or 1.44M high density diskettes, it takes a big handful to backup a 20, 40 or 80 megabyte hard drive.

A complete full-drive backup is appropriate:

- just before having the system serviced,
- when transferring everything on one computer's hard drive to another computer, or
- on a periodic basis, every week or month, in addition to other more frequent partial backups.

You first need to determine first how many formatted diskettes are needed. With the C> prompt showing, type:

chkdsk ENTER

Look at the number of bytes in user files, plus the number of bytes in directories. Suppose the display shows:

```
2191360 bytes in 158 user files
```

Add the bytes in directories, then divide that number of bytes by the capacity of each diskette. Then round the result up to the next whole number. In this example, seven 360K, four 720K, or two 1.2M or 1.44M diskettes should do the job. It's a good idea to have an extra formatted diskette on hand, just in case.

If they're fresh from the box, **format** the diskettes before proceeding. There's no point in making them system disks so don't bother with **format**'s **/s** or **/b** switches. If they are already formatted, **backup** will erase old data on them before it copies on the new.

The full command is in this form:

backup c:*.* a: /s

If **backup** is not in the root of Drive C, you must set the path so DOS can find it.

c:*.* designates all files in the root of the source drive. Omitting the \ defaults to the current directory.

a: designates the target drive.

/s tells **backup** to also include all subdirectories. Without it, only the current directory (the root in this case) is backed up.

Easy Does It

To keep our first try simple, we'll backup only the root directory. Use the command:

backup c:*.* a: ENTER

backup responds with a request for the first diskette. Insert it and press ENTER. Each file name is displayed as the backup progresses.

If you get an error message during the procedure, and you're unable to correct the situation (by pressing R for retry), you'll need to restart. You'll also need to restart if you run out of formatted diskettes!

When the C> prompt returns, the backup is complete.

Remove the diskette and write something like "Root Backup - 8/15/xx - #1" on the label. (If it takes more than one diskette, you'll number the next one "#2" and so on.)

Restoring the Hard Drive

restore is the opposite of **backup**. You'll probably only use **restore** as an emergency measure when files on the hard drive have been erased or damaged.

The hard drive must be ready before you can use **restore**. If you are unable to boot your computer from Drive C and display a directory, you must perform the initial setup procedures. (See Part 8 and Appendix A.) Otherwise, initialize the hard drive only when absolutely necessary:

- when you are loading files onto a new hard drive,
- on the recommendation of a technician after servicing,
- when disk error messages have become a recurring problem on the hard drive, or
- when installing a new version of DOS, if the accompanying instructions recommend reformatting.

Go to the Drive C root directory and type:

restore a: c:\/s ENTER

a: designates the drive which will read the backup diskettes.

c: designates the target drive and the root directory.

/s tells the program to also restore files that are in subdirectories. Without **/s**, only the current directory is restored.

restore with **/s** will also rebuild the tree-structure on the hard drive, if necessary. If, for example, a file was in the ACCOUNTS subdirectory at the time you backed up, **restore** will do an automatic **mkdir accounts**.

Error Messages

backup puts a file called BACKUPID.@@@ or CONTROL on every backup diskette. **restore** uses the information in this file to make sure the diskettes are inserted in order.

Diskettes inserted out of order or those not created by **backup** cause the `Insert diskette` message to reappear. Press CTRL C if you can't find the correct diskette.

Selective BACKUP and RESTORE

We'll create two practice files to illustrate **backup**'s special features and switches, avoiding the possibility of goofing up any important files. Create the following in the root directory of your hard drive:

copy con c:\demo1.txt ENTER
This is demo file #1. ENTER
CTRL Z ENTER

copy con c:\demo2.txt ENTER
This is demo file #2. ENTER
CTRL Z ENTER

Within limits, **backup** and **restore** can handle individual files or groups of files. For example, let's back up only those files which begin with the name DEMO, and are in the root.

The *source* and *target* parameters are arranged just like in a **copy** command. Type:

backup c:\demo?.* a: ENTER

Examining the Backup Diskette

Display the directory of Drive A. DOS version 3.2 lists the files this way:

```
BACKUPID @@@      128  5-15-xx  3:38p
DEMO1    TXT      152  5-15-xx  3:35p
DEMO2    TXT      152  5-15-xx  3:35p
```

The BACKUPID.@@@ file identifies the diskette as a backup disk. It contains:

- a code to indicate this is diskette 1 of the backup,
- the date of the backup, and
- a code to indicate this is the last backup diskette. (This backup didn't require additional diskettes.)

Notice that all backed-up files are 128 bytes longer than the originals on the hard drive. **backup** adds a *header* to each file which contains:

- the pathname for the file,
- a code to indicate whether or not the file is continued on another diskette, and
- for continuation files, a number to ensure proper sequence when restoring.

Because of the 128-byte header, the files and programs on a backup diskette cannot be used until they are restored to the hard drive.

A Word About DOS Version 3.3

It seems like every time MS-DOS is updated, the way **backup** works is changed. I'm not sure it's always for the better, but that's the way it is.

For starters, DOS 3.3 stores the backup this way, clustering everything in just 2 files:

```
BACKUPID 001          48   5-15-xx   3:38p
CONTROL  001         277   5-15-xx   3:35p
```

That won't ruin our whole day, but we certainly can't tell what's in those files. And we can't follow the changes which come in the rest of this chapter as easily as with DOS 3.2 on the 1000 TX, which is what this book is written around. Users with 3.3 should read along and do the balance of the steps in this chapter, recognizing that there are things that you can do, but can't always see.

Version 3.3 users must specify the source (C:), below. It is optional for version 3.2 users.

BACKUP's /A Switch

Suppose you want to put more than one group of files on a set of backup diskettes. The /**a** switch (for *add* or *append*) makes it possible. To try it, create 2 more files on the hard drive. Call them DEMO1.XXX and DEMO2.XXX:

copy demo?.txt *.xxx ENTER

DEMO1.TXT and DEMO2.TXT are already on the backup diskette. The **/a** option lets you add files to a backup diskette without erasing what's already there. Type:

backup demo?.xxx a: /a ENTER

Display the directory of Drive A to check it out. (Can't tell much on version 3.3.)

What if the backup disk already has the files you are adding? Unlike **copy**, which would overwrite the old files, **backup**'s **/a** option gives the duplicate files new names.

DEMO1.TXT and DEMO2.TXT are already on the backup diskette. Before trying **/a** with these files, let's modify the hard drive versions. At the C> prompt, type:

copy demo1.txt+con ENTER

DOS responds with:

```
DEMO1.TXT
CON
```

Type:

This is the new version. ENTER
CTRL Z ENTER

Do the same for DEMO2.TXT.

Now type:

backup demo?.txt a: /a ENTER

Display the Drive A directory. DEMO1.@01 and DEMO2.@01 have been added. No two files in a directory can have the same name, so **backup** invented its own names.

DEMO1.TXT and DEMO2.TXT are the old versions. DEMO1.@01 and DEMO2.@01 are the new versions of the same files. (When more than

one version of the same file name is on a backup diskette, they are named @01, @02, @03, etc.)

backup uses this same renaming method when it comes across two files with the same name in different subdirectories. It keeps the actual name and path in the 128 byte header of each file.

Let's restore the files. Type:

restore a: c: ENTER

Use **type** to view DEMO1.TXT on your hard drive.

Surprise! It is the most recent version.

BACKUP's /M Switch

The /**m** switch selects just those files that have been modified since the last backup. Used with /**a**, it can be a real time saver.

DOS stores an *attribute byte* with each entry in the directory. The bits in the attribute byte tell DOS certain things about a file. One of them is called the *archive bit*. Whenever DOS writes to a file and closes it, the *archive bit* is changed to 1. When **backup** copies the file, it changes the *archive bit* back to 0. This way, **backup** knows what files may have been modified.

Enter this command:

backup demo?.* a: /a/m ENTER

backup responds with:

```
File Not Found
```

You haven't modified any DEMO files since last backing them up, so nothing was copied. Modify one of them:

copy demo1.xxx+con ENTER

DOS displays:

```
DEMO1.XXX
CON
```

Type:

This is version 2. ENTER
CTRL Z ENTER

Let's also make a new file:

copy demo1.txt *.yyy ENTER

And, try the **backup** command again:

backup demo?.* a: /a/m ENTER

Only the DEMO1.XXX and DEMO1.YYY files are copied. DEMO1.XXX is backed up because we *modified* it. DEMO1.YYY is backed-up because it is *new*.

The /**m** switch can be used alone or together with /**a** and /**s**.

Backing Up by Date

Another backup option is /**d**. It allows selective backups based on the dates in the directory. For all root files created or modified on or after 5/16/99, try:

backup demo?.* a: /d:05-16-99 ENTER

Since all of the demo files were created on (your date), none of them are backed up. Use:

backup demo?.* a:/d:(your date) ENTER

This time **backup** copied them. See your own specific factory MS-DOS manual for other possible **backup** switches.

RESTORE's /P Switch

The /**p** option for **restore** is related to the /**m** option for **backup**. Use it when you want the system to pause for your approval before restoring hidden and invisible files on diskettes that were backed up using an earlier version of DOS. Normally you will not want to restore those specific files.

XCOPY

The **xcopy** command is a powerful combination of **backup** and **diskcopy**, but can use two diskettes of different sizes. (**diskcopy** requires that both diskettes be the same format and size. **backup** doesn't work between floppies.)

xcopy copies all the files from one diskette to another. If the second diskette is smaller than the first, **xcopy** will copy files until full, but, unlike **backup**, **xcopy** will not split a file in two.

To **xcopy** all files from the diskette in Drive B to Drive A, use:

xcopy b:*.* a:

To **xcopy** all files from the subdirectory BASIC on Drive C to the diskette in Drive A, use:

xcopy c:\basic a:

To **xcopy** all files from Drive A to a subdirectory DISKA on Drive C, use:

xcopy a: c:\diska

If all the subdirectories of Drive A are to be copied to Drive C, the /**s** option is added:

xcopy a: c:\diska /s

xcopy copies the entire disk, subdirectories and all, up to Drive C. To make sure even empty subdirectories are copied, add the /**e** option:

xcopy a: c:\diska /s/e

Other options, similar to those used with the **backup** and **restore** commands are: /**m**, /**a**, /**p**, and /**d**.

/**m** copies only files that have been modified since the last **backup** or **xcopy** command.

/**a** works the same as /**m**, except the *attribute bit* is not turned off. The file is still considered modified since the last **backup**.

/**p** prompts before each file is copied. DOS displays the file name along with a (Y/N)? prompt.

/**d** copies files created or modified after a specific date. /**d:11/10/99** allows only files created or modified on or after 11/10/99 to be copied.

The /**v** option is used like the /**v** option for the **copy** command.

The final **xcopy** option is /**w**, which stands for wait. After typing:

xcopy b: a: /w

the following message is displayed:

```
Press any key when ready to start copying files
```

Chapter 29 Summary

copy, **diskcopy**, **backup**, **restore**, and **xcopy** allow you to duplicate files.

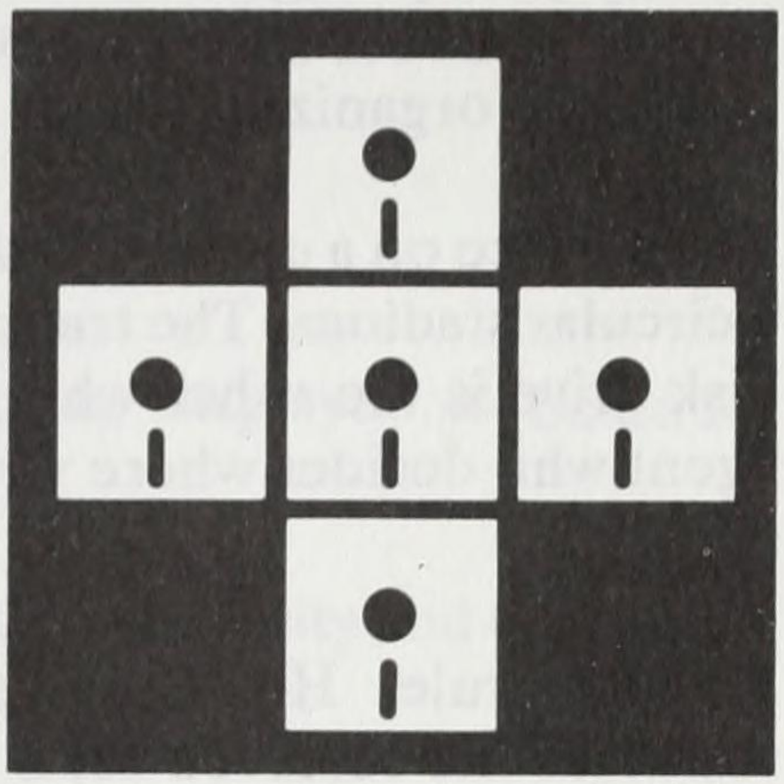
backup copies files from a hard drive to one or more diskettes for safekeeping. Unlike **copy**, it can split files where necessary, so they may be continued from one diskette to another.

backup has optional switches, which may be used together or separately. /**s** causes files in all subdirectories of the current directory to be included in the backup. /**a** adds files to a backup diskette without erasing those that may already be present. /**m** backs up only those files which were modified since the last backup. /**d:mm/dd/yy** backs up only those files created on or after the given month, day, and year.

Files stored on floppy diskettes by **backup** may be reinstated onto the hard drive with **restore**. **restore a:** copies backup files from Drive A. **restore a: /s** copies backup files from Drive A, including any which may have been backup up from subdirectories of the current directory on Drive C. **restore a: /p** pauses for approval before restoring invisible or read-only files.

xcopy copies files from one disk to another. The 2 disks need *not* be the same format or size. **xcopy**'s **/s, /m, /a, /p** and **/d** are similar to **backup**'s and **restore**'s. **xcopy b:*.* a: /s/e** copies all files, including the empty subdirectories, from Drive B to A.

CHAPTER 30



Advanced CHKDSK

We've used **chkdsk** to determine disk capacity and memory space. It can also examine a diskette's directory and make "repairs" if it isn't organized correctly.

Why CHKDSK?

With dozens of files being created, updated, lengthened, and deleted, the bookkeeping for what data belongs to which file, what space is available, and how each file "chains" from one point to another on a diskette becomes quite complex. MS-DOS does it well, except when...

- the computer is turned off or rebooted before the directory is updated,
- diskettes are swapped or removed before data being held in memory buffers is written,
- Control-C is used to terminate a program while writing data to a file, or
- incompatible programs or commands are used.

These conditions don't always result in a problem, but **chkdsk** should be used after any of the above, or if anything suspicious occurs. **chkdsk** checks the "integrity" of the directory to see if the statistics agree with one another.

How DOS Organizes Diskettes

To understand the messages **chkdsk** may display, you need to understand how DOS organizes data on a disk.

The surface on a diskette is divided into tracks and sectors. Think of it like a circular stadium. The tracks are the rows. The sectors are the aisles. The disk drive is the usher who puts you in the right seat. DOS is the ticket agent who decides where you'll sit. Let's see how this "ticket agent" does his job.

He has a rule. He sells all seats in *clusters* of 2. This makes it easier to maintain the reserved seats chart. Each box on the seating chart represents 2 seats.

When someone buys a ticket, he looks for an available box on the seating chart and marks it. Then he writes the buyer's name in a name list with a number that refers to the box on the seating chart.

Suppose a buyer wants 5 seats. The ticket agent finds 3 empty boxes on the seating chart. In the 1st one he puts the number of the 2nd, and in the 2nd one he puts the number of the 3rd. In the 3rd box he writes a special mark. Then he puts the number of the 1st box next to the buyer's name so he can follow the "chain" of numbers to all the seats that have been reserved by one buyer.

The FAT and the Directory

The DOS "seating chart" is called the *file allocation table*, or FAT. The list of names is the *directory*. Each entry in the file allocation table is a cluster. Each cluster is 2 adjacent *sectors* of 512 bytes each. (Better read that again.)

When you format a disk, DOS creates the file allocation table and root directory, just as a ticket agent prepares an empty seating chart and name list for an upcoming event. If a bad sector is found during formatting, DOS "locks it out" by putting a notation at the corresponding location in the FAT.

Each FAT entry requires 2 bytes, and each entry in the root directory requires 32. DOS sets aside this space during formatting, depending on whether 1 or 2 sides of the disk and 8 or 9 sectors per track are to be used.

Total Disk Space

When running CHKDSK.COM, the first number displayed is `total disk space:`.

We can account for the difference between diskette capacity and **chkdsk**'s total disk space statistic if we know two more things:

- The first sector on the first track of each diskette is the *boot sector*. It contains the logic needed to load the DOS, or in the case of a non-system disk, the commands to display a non-system disk message.

- *Two* copies of the FAT are stored on the disk. DOS keeps them both up to date so it can use the second one if there's a problem with the first.

Hidden Files

chkdsk also shows the total bytes used by *hidden files*. One of the 32 bytes in each directory entry is the *attribute byte*. You'll remember that one bit in the attribute byte tells whether the file has been backed-up or not. Another bit tells whether it is a *hidden file*. Hidden files are not listed by the **dir** command and are shielded from commands like **copy** and **delete**.

For each file with the hidden attribute, **chkdsk** follows the chain to the FAT table, counts the clusters, and (since there are two 512-byte sectors per cluster) multiplies the number of clusters by 1024.

Directories

Only the root directory is limited in size. On diskettes, it occupies 7 sectors, enough space for 112 entries. On the hard drive, it occupies 16 sectors, allowing up to 512 entries.

Subdirectories can have any number of entries because they are stored as variable-length files. When you use **mkdir**, DOS puts the new subdirectory's name in the current directory, just as it would if you were creating a file. The only difference is the attribute byte. Another of the 8 bits in the attribute byte identifies the filename as a subdirectory.

If **chkdsk** finds a subdirectory's name, it follows the chain through the FAT table and tallies the number of sectors. The result is displayed in a message such as:

```
xxxx bytes in x directories
```

User Files

Your programs and data are stored in *user files*. Whenever **chkdsk** finds a user file in the directory, it follows the chain to the FAT. Then, as it does for hidden files and subdirectories, it follows the numbers in the file allocation table until it arrives at an "end of chain" code. The message shows total bytes and a count of the files:

```
xxxxxx bytes in xx user files
```

Have you noticed that all the byte counts **chkdsk** shows (except those for memory) are evenly divisible by 1024? Like the ticket agent who sells a minimum of two seats at a time, DOS allocates a minimum of two 512-byte sectors at a time.

Bad Sectors

To find the number of bytes in bad sectors, **chkdsk** goes through the file allocation table and counts the clusters flagged as being bad. Then it multiplies by 1024 for the number to show in its message:

```
xxxxx bytes in bad sectors
```

It's important to know that **chkdsk** doesn't read the entire diskette looking for bad sectors. It reads the file allocation table and directory.

The bad sectors noted in the file allocation table are usually those that FORMAT.COM found when the diskette was formatted. The only other program that updates the FAT for bad sectors is RECOVER.COM. We'll learn about the **recover** command in the next chapter.

Bytes Available

`Bytes available` is what remains after subtracting space for the boot sector, file allocation tables, the root directory, hidden files, subdirectories, user files, and bad sectors. **chkdsk** shows the count:

```
xxxxx bytes available on disk
```

To find bytes available, **chkdsk** simply goes through the FAT from beginning to end. At each location, it will find one of four things:

- A number which refers to another location in the file allocation table. This means the corresponding cluster is part of a file.
- A code meaning "end of chain." This means the corresponding cluster is the last one in a file.
- A code meaning the corresponding cluster contains a bad sector.
- A zero meaning that the corresponding cluster is unused and available.

When creating a new file, DOS does the same. It looks through the file allocation table until it finds a 0 entry. For example, if the first 0 entry is in the 40th cluster, it writes the first 1024 bytes of the new file onto the 80th and 81st sectors of the diskette.

Lost Clusters

Since we can't get any real hands on practice with lost clusters, we'll just *pretend* to go through situations you might encounter so you'll know what to do if they occur.

chkdsk may display a message like:

```
Errors found, F parameter not specified.
Corrections will not be written to disk.
```

```
x lost clusters found in x chains.
Convert lost chains to files (Y/N)?
```

The "F parameter" refers to **chkdsk**'s *fix* mode. We'll see how it is used in a second.

The "lost clusters" message means there is information on the disk, and DOS doesn't know what file "owns" it.

It's like the ticket agent who rechecks reservations. Starting with each number in the name list, he goes to the corresponding box on the seating chart and puts a check mark in it. If there's a number in the box, he follows *that* number to another box, and checks it off. After following the chains from all the names, suppose he finds that he can't account for boxes 10, 11, 12, 30, 35, and 36 on the seating chart.

Studying it further, he finds that the unclaimed boxes are in two different chains. Box 10 has an 11 in it, box 11 has a 12 in it, and box 12 has an "end of chain" code.

At the same time, box 30 has a 35 in it, box 35 has a 36, and box 36 has an "end of chain" code. The solution? Put two "dummy" names in the list to hold the seats in case someone comes back to claim them. For now, "John Doe 1" and "John Doe 2" will do. Next to "John Doe 1" he records a "10" to take care of the first chain. Next to "John Doe 2" he records a "30" to take care of the other chain.

chkdsk can do the same thing. It asks:

```
Convert lost chains to files (Y/N)?
```

If you press Y for yes, along with other statistics it says:

```
xxxx bytes would be in
   x recovered files
```

Notice it says "would be." This has been a "dry run." To do it for real, the entry would be:

chkdsk /f ENTER

...or **chkdsk** and the drive letter, **a:**, **b:**, or **c:**, followed by /**f**.

Then it would say:

```
xxxx bytes in x recovered files
```

Checking the Recovered Files

After **chkdsk** has done its work, it is up to you to decide what to do with the recovered files. The first step is to display a directory. **chkdsk** uses CHK to identify files it has recovered: The command is:

dir *.chk ENTER

Perhaps you see:

```
FILE0000 CHK     1024
FILE0001 CHK     5120
```

These are the "John Doe" file names **chkdsk** assigned. Use the **type** command on each file. If you recognize the contents, you'll know what programs or data files the lost clusters came from.

In some cases, just renaming a CHK file to its correct name will make the data as good as new. In other cases, you may be able to use **edlin** or your word processor to combine information from a CHK file back into the file where it belongs.

If you don't recognize what a CHK file contains, you should test every program that uses the disk you fixed. If some data turns up missing or a program doesn't work, part of it may be a CHK file. In either case, you can go to the backup diskette or try to reconstruct the files.

If your programs and data files look OK, the CHK files probably just contained obsolete data. Delete them to recover the disk space the lost clusters were using.

Cross-Linked Files

Another possible problem is *cross-linked* files. The situation is like a ticket agent who accidentally reserves the same seats for two different parties.

For each file name involved, **chkdsk** displays a message:

```
A:\PRODUCTS.TXT
Is cross linked on cluster 31
A:\CUSTMERS.TXT
Is cross linked on cluster 31
```

To fix it, do what a ticket agent again would do. Assign new seats. This means copying the cross-linked files and deleting the originals. Here's how the CUSTMERS and PRODUCTS files could be fixed:

```
copy products.txt products.new   ENTER
copy custmers.txt custmers.new   ENTER
del products.txt   ENTER
del custmers.txt   ENTER
rename products.new products.txt   ENTER
rename custmers.new custmers.txt   ENTER
```

Now the problem is solved, except for one thing. The CUSTMERS.TXT file contains data from the PRODUCTS.TXT file, or vice versa. If they are in ASCII, you can use **edlin** or a word processor to correct the faulty file. If they are programs or other types of binary files, the most practical fix is to restore the bad file from a backup copy.

Allocation Errors

The size of a file is recorded two ways on disk. First, DOS uses the directory. With each file name, it keeps an exact count of the bytes.

chkdsk has another way. It can follow a file's chain through the file allocation table, counting 1024 bytes per cluster.

When **chkdsk**'s count is smaller than the size recorded in the directory, i reports an allocation error:

```
A:\SCORES.DAT
 Allocation error, size adjusted.
```

Suppose, for example, the directory says that SCORES.DAT is 40123 bytes, but **chkdsk** only finds 38 clusters in the chain. 38 clusters are 38912 bytes. To resolve the difference, **chkdsk** changes the length in the directory entry to 38912.

The opposite can also occur. **chkdsk** may find more clusters chained to a file than would be indicated by the number in the directory. **chkdsk** solves the problem by writing an "end of chain" code in the FAT to make the chain's length agree with the directory. It displays a message, such as:

```
A:\LETTER.TXT
 Has invalid cluster, file truncated.
```

There's still another possibility. Suppose an entry in the directory chains to an unassigned cluster. **chkdsk** displays:

```
A:\PHONELST.TXT
 First cluster number is invalid
 entry truncated
```

In this case, **chkdsk** handled the problem by changing the file's length, as recorded in the directory, to zero.

To correct each of these allocation errors, you must use **chkdsk** with the **/f** option.

Probable Non-DOS Disk

DOS uses the first byte of the FAT to record information about the size and format of a disk. It tells whether it is single or double sided, and how many tracks it has.

If **chkdsk** finds an invalid code at the beginning of the FAT, it warns that the disk may be incompatible:

```
Probable non-DOS disk
Continue (Y/N)?
```

If you press Y for yes, **chkdsk** will try to continue.

Problems in Subdirectories

If there are subdirectories on your disk, **chkdsk** may report errors pertaining to them. For example:

```
A:\CONTACTS
 Invalid sub-directory entry.
 tree past this point not processed.
 Convert directory to file (Y/N)?
```

Press Y, and **chkdsk** will probably go on to indicate some lost clusters:

```
2 lost clusters found in 2 chains.
Convert lost chains to files (Y/N)?
```

You should answer Y to this question too. Here's what you've done:

The first "yes" tells **chkdsk** to create a file called CONTACTS. It contains the directory information formerly contained in the CONTACTS subdirectory.

The second "yes" tells **chkdsk** to create CHK files. If the CONTACTS subdirectory contained two files, FILE0000.CHK will probably be the first one, and FILE0001.CHK will probably be the second. If they are ASCII files, rename them, use **edlin** to "clean them up," then recreate the CONTACTS subdirectory.

Again, you must use **/f** for **chkdsk** to write its corrections to disk. Before using **/f**, see what you can correct yourself with the **copy** command. Sometimes you can successfully copy good files out of a bad directory. That's much easier than trying to determine which file is which after **chkdsk** has recovered them.

Other CHKDSK Errors

The /**f** parameter corrects two other types of errors **chkdsk** may find:

```
Invalid sub-directory entry
```

and

```
Cannot CHDIR to filename
```

If **chkdsk** reports `Disk error reading FAT` or `Disk error writing FAT`, you should attempt to **copy** (or **xcopy**) all the files to another diskette.

If a message says `Processing cannot continue`, try restarting your system and rerunning CHKDSK.COM. If that doesn't work, try the **recover** command.

Sometimes **chkdsk** will create so many CHK files that the root directory becomes full. The solution is to **copy** (or **xcopy**), then delete the CHK files and try again.

Finding a "Lost" File

If you type:

chkdsk /v ENTER

chkdsk will list the files on the disk as it checks them. The visual effect is similar to the **tree** command. This is ideal if you've put a file in a subdirectory and can't remember where it is. Suppose you want to find all files having the word "LUCK" somewhere in their file names. Just type:

chkdsk /v | find "LUCK"

Be sure the search string is in all upper case letters. If **chkdsk** finds it, the complete pathname is displayed:

```
A:\EXAMPLES\POTLUCK.FRM
```

Use this technique only when you're sure the disk has no errors. (Do a regular check of the disk first.)

CHKDSK with a File Name

One or more file names can be specified in the **chkdsk** command. For example:

chkdsk all3.txt potluck.frm ENTER

This is a way to find out if the clusters each file occupies are next to one another. If it says:

```
A:\ALL3.TXT
 Contains 3 non-contiguous blocks.
```

you know that DOS has to do some "jumping around" each time it reads ALL3.TXT. If a file is contiguous, DOS can read it faster. You can make files contiguous by copying them to an empty diskette.

QUIZ: What is the bottom line of this long chapter?
ANSWER: **MAKE FREQUENT BACKUPS!!!**

Chapter 30 Summary

Besides giving the statistics about disk space, **chkdsk** is able to report and correct certain errors.

A diskette is accessed by *track* and *sector*. Two contiguous sectors are called a *cluster*. When managing disk space, MS-DOS allocates space by cluster.

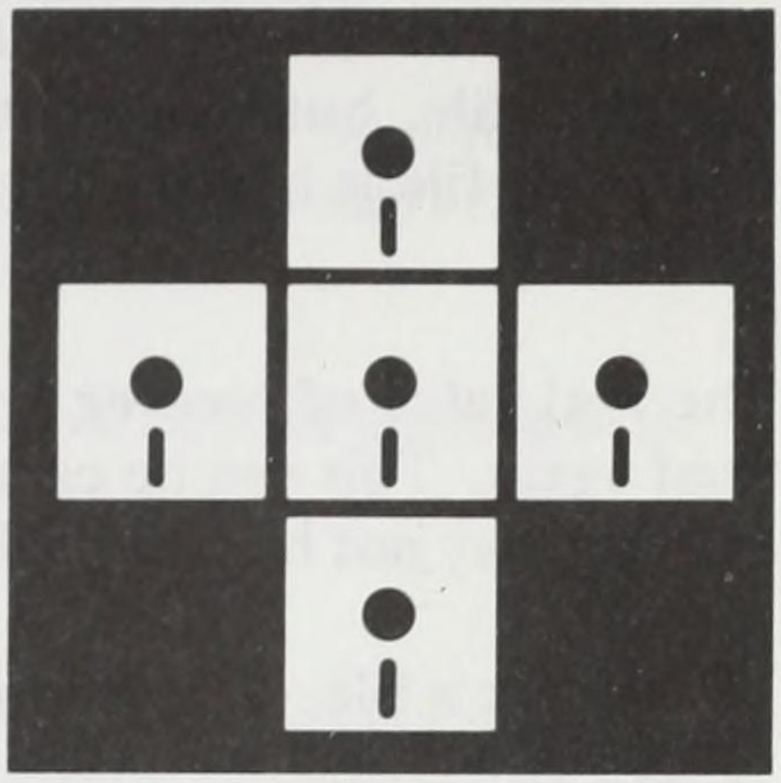
chkdsk examines the file allocation table and directory. The *file allocation table* (FAT) contains one entry for each cluster, telling whether or not the cluster is in use by a file, and if so, what cluster contains the next 1024 bytes of the file.

The *directory* contains all the file names with their creation dates, lengths, attribute codes, and starting cluster numbers. Each starting cluster number leads DOS to a position in the file allocation table, which tells where the file's data is located.

Use **chkdsk /f** when you want **chkdsk** to correct the errors it finds. For some types of errors, **chkdsk** creates files to hold the data it recovers. It names them FILE0000.CHK, FILE0001.CHK, and so forth. Other types of errors can be corrected by copying files to another disk.

The /**v** switch can be used with **chkdsk** to list the file names as they are checked. Follow **chkdsk** with a file name to determine whether the file occupies contiguous clusters on the disk.

CHAPTER 31



RECOVER: A Life Saver

chkdsk corrects *logical* problems on a disk. It makes sure the directory agrees with the file allocation table, and the file allocation table agrees with itself.

recover specializes in correcting *physical* problems. A fingerprint can flaw a sector. Setting a diskette near a magnet can make a file unreadable.

Or it may be no fault of your own. A sudden surge or dip in power can cause a bad sector. Perhaps the disk drive is out of alignment, or a portion of the diskette has just "worn out."

`Write fault error`, `read fault error`, and `sector not found` are typical messages that should prompt you to consider using **recover**.

RECOVERing a File

Before using **recover** to "rescue" a file, it is best to attempt other remedies. Try copying the file to another file, or another disk. If you get an `Abort, Retry, Ignore` message, be sure to retry several times. If possible, use another disk drive or another computer to copy the file. If you have a recent backup, consider using it, instead. Many recovery attempts are not successful.

recover reads a file until it gets to a bad sector. It marks the corresponding entry in the FAT so the sector's cluster won't be used again. Then it links the *prior* cluster in the file's chain to the next cluster. The result is a readable file, but one or more 1024-byte sections are missing. If the recovered file is in ASCII, perhaps you can re-enter the missing parts via **edlin**.

The real value of **recover** is that it locks out bad sectors so they won't be used again. This can be especially important on a hard drive where reformatting may not be practical.

To recover a file, type **recover** and the file or pathname. For example:

recover a:carbills.dat ENTER

On completion, **recover** tells how much of the file it was able to rescue:

```
xxxx of xxxx bytes recovered.
```

Now you can delete the file, copy it, or continue using it.

Bad Sectors Outside Files

recover can lock out bad sectors only when they are within files. It can't help you with bad sectors occupying "unused" disk space. Remember this rule:

Never delete a file if you know it has a bad sector. ***recover*** *it first, then delete it.*

Bad sectors in unused space cause problems when adding new data and when trying to use the disk as a source for a **diskcopy**. The only way to lock out bad sectors *outside files* is to reformat. Fortunately, you can preserve the data by using **copy** first.

RECOVERing a Directory

This can be a life saver, but it is strong medicine! Think carefully before using it.

Suppose the directory has a bad sector. The symptom: when using **dir**, you get an error message, or only part of the directory displays.

recover can rebuild it, but remember, DOS may not be able to load RECOVER.COM from a bad directory. You may need to start the recover program from a different *system* disk.

If the diskette having the bad directory is in Drive B, your command is:

recover b:

recover waits for your go-ahead, so you can swap diskettes (or press CTRL C) if necessary. To prepare a usable directory, first it deletes all file names and locks out the flawed sectors. Then it scans the file allocation table and creates a new directory entry for each chain it finds. **recover** doesn't know the file names, so it uses FILE0001.REC, FILE0002.REC, FILE0003.REC, and so forth.

When the directory has been recovered, copy all the REC files onto another diskette, and delete them from the recovered directory. Try the **recover** command again. More files may be recoverable.

Now comes the hard part. It is your job to figure out which REC file is which. Fortunately, you are only interested in the files for which you have no backup. Rename these. For the rest, use your most recent backup copies.

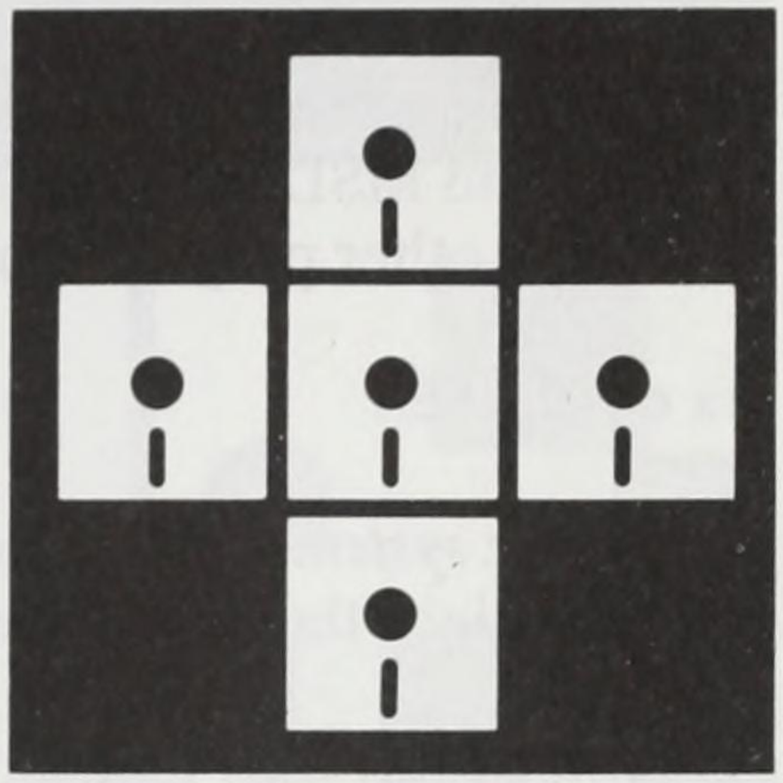
Chapter 31 Summary

The **recover** command locks out bad sectors on a diskette or hard drive without reformatting. Type **recover**, followed by the name of the file that contains the bad sector: **recover notes.lhr**.

recover can also rebuild a damaged directory. Type **recover**, followed by the drive letter: **recover a:**.

Special care must be taken. When recovering files, some of the data may be unrecoverable. When recovering a directory, it is up to you to reconstruct the file names.

CHAPTER 32



Keeping Up to Date

Keeping your system healthy also means keeping it up to date. From time to time, new versions of DOS are released:

- to correct a "bug" in a prior program,
- to improve compatibility with the latest computers, peripherals, and programs,
- to add new commands and features, or
- to improve speed or efficiency.

The VER Command

To identify the generalized version of DOS you are using, type:

ver ENTER

To identify the specific Tandy version, reboot and read the message on the screen.

The SYS Command

Before installing a new version of DOS, it's best to make a complete set of backups of all files and programs just in case the new release is incom-

patible. Then, turn off the computer and reboot with the new *system* disk in Drive A.

The **sys** command installs the new IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS) files on a formatted diskette that may already be storing other programs and data files. For example:

sys c: ENTER

installs *the system* files on the hard drive. Use **copy** to put the other new DOS files into the directories where they're stored.

Sometimes new files take more space than the old ones. If an `Insufficient Disk Space` error results, you need to make new decisions about what to put on each disk and how to organize the data files.

When using the **sys** command, you may get `No room for system transfer on destination disk  Incompatible system size`. This means you must format a new disk using the new **format /s**. After reformatting, you can recopy all your old files, then copy the new ones.

In addition to installing the latest version of DOS on a disk that is already a *system* disk, **sys** can be used to install DOS on program diskettes you have purchased or on disks created on a different computer model.

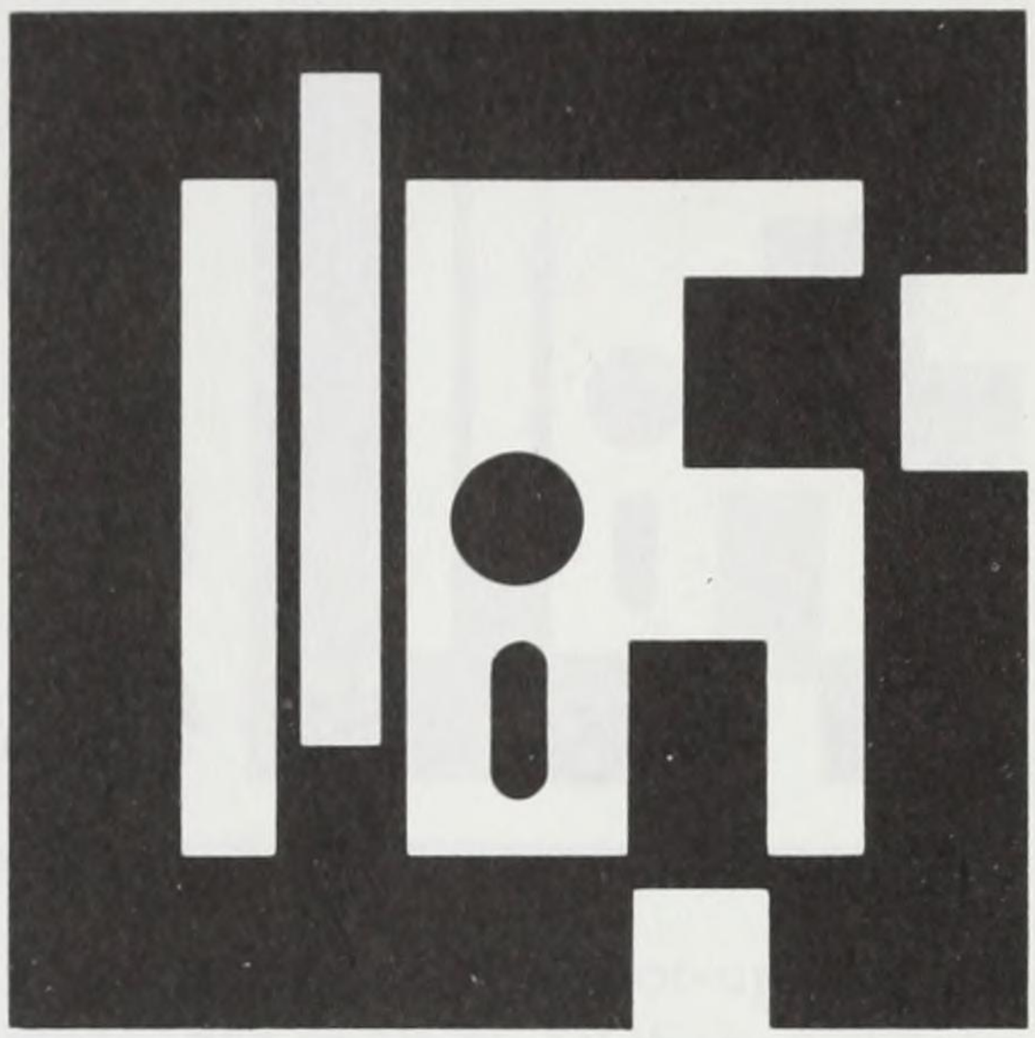
Chapter 32 Summary

The **ver** command tells what general version of DOS you are using.

To identify the specific version and release number, reboot and read the screen. If the message doesn't appear, reboot using an unmodified *system* diskette.

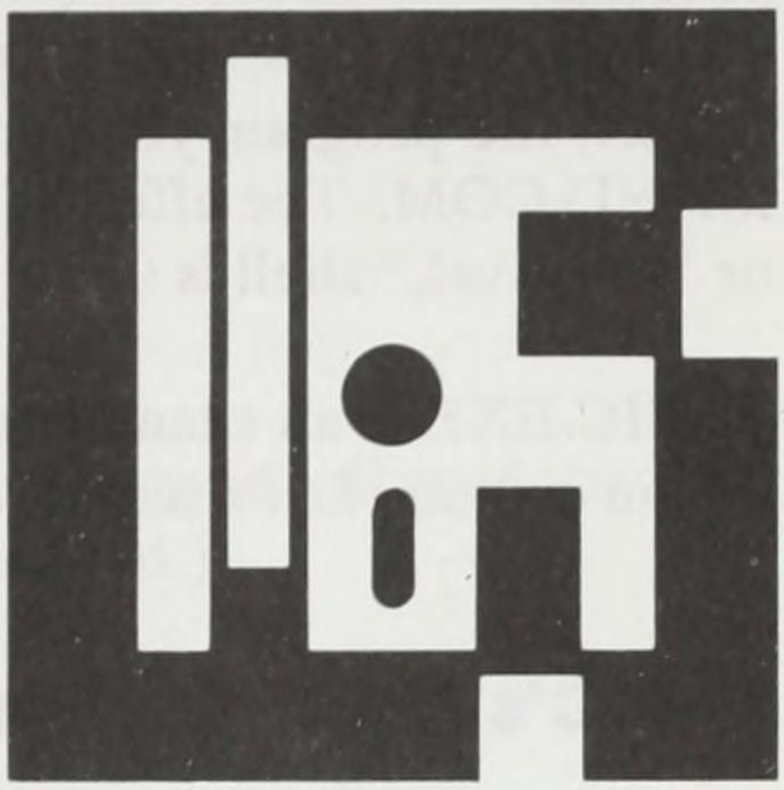
The **sys** command installs the new IBMBIO.COM and IBMDOS.COM (or IO.SYS and MSDOS.SYS) files on another disk. The rest of the files can be copied by the **copy** command.

PART 7



Behind the scenes

CHAPTER 33



DOS in Disguise

Let's think a little more deeply about what happens in the boot-up process. The computer first reads the *system* disk to load in DOS. DOS then checks for an optional CONFIG.SYS file, which may include information about how to allocate memory and which device drivers, e.g. ANSI.SYS and MOUSE.SYS, to load.

At this point, DOS has all its capabilities but is like an engine without a steering wheel. To install the steering wheel, it loads and executes COMMAND.COM

COMMAND.COM searches the root directory for an optional AUTOEXEC.BAT file. After executing the commands it contains, the system prompt appears and you have your hands on the wheel!

Shells

To run a program you simply enter its name. COMMAND.COM passes the name on with a load and execute request, and the EXEC function within DOS takes control. EXEC searches the disk, checks for adequate memory, loads the program, and runs it.

When one program reserves memory in which to run another program, it is creating a *shell*. COMMAND.COM forms a shell in which to run *your* program by giving DOS the EXEC request. Your program might make an

EXEC request itself, creating a smaller shell in which to run yet another program.

In fact, the program your program runs might be a *second* copy of COMMAND.COM. The effect is a shell within a shell within a shell. The outer, or "top level," shell is COMMAND.COM.

BASIC.EXE is an example of a program that can run COMMAND.COM within a shell. Let's see how it works.

BASIC's SHELL Command

If you deleted BASIC, put your *system master* in Drive A.

Go into BASIC by typing:

basic ENTER

BASIC is a programming language. BASIC.EXE is the *program* that allows you to use the language. COMMAND.COM has formed the shell in which BASIC.EXE is running.

Below the `Ok` prompt, type the following:

```
10 print "Now we're in BASIC"  ENTER
20 print "Press any key to give control to COMMAND.COM"  ENTER
30 a$=input$(1)  ENTER
40 shell  ENTER
50 print "Now we're back in BASIC"  ENTER
```

To see how BASIC.EXE creates a shell in which it will run COMMAND.COM, type:

run ENTER

Respond to the `Press any key` request, and the `A>` or `C>` system prompt appears.

The **shell** command in line 40 told BASIC to load and execute COMMAND.COM within a shell. Prove it to yourself by typing **dir** to display a directory. Try other commands such as **copy**, **delete**, and **rename** if you wish.

The EXIT Command

COMMAND.COM is now in control. Its **exit** command will send you back to BASIC. (BASIC does not have an **exit** command.)

exit ENTER

The BASIC program finishes execution by saying `Now we're back in BASIC.`

We created a shell for COMMAND.COM, executed some DOS commands, and used **exit** to return to the same BASIC programs. Think of the advantages. Suppose you are running a program and discover you need more disk space. If the program has a shell feature, you can erase files or format a disk from the program itself, without ending and restarting the session. Use:

system ENTER

to exit BASIC. Now the *original* COMMAND.COM is back in control!

Custom Command Processors

MS-DOS has a provision for an optional entry in the CONFIG.SYS file, just in case you want to use a "custom steering wheel." Instead of COMMAND.COM, another program can be the command processor.

The custom processor might be designed to accept foreign-language commands in place of the **delete**, **copy**, and **rename** we're used to in English. Or it might allow entry of DOS commands with a mouse and the keyboard's arrow keys. Another possibility is a command processor for a dedicated purpose, such as inventory control with a bar code reader.

Suppose a file called CONTROL.COM is to be used in place of COMMAND.COM. To notify DOS, the following entry is needed in CONFIG.SYS:

shell=control.com

This causes DOS to execute CONTROL.COM instead of COMMAND.COM after booting. To the user it may not look like DOS at all. It is DOS in disguise!

Relocating COMMAND.COM

You can also use the **shell** entry to designate a "not so normal" location for COMMAND.COM. Consider this:

shell=a:\doscmds\command.com a:\doscmds /p

The first part tells DOS that the "top level" command processor is COMMAND.COM, in the DOSCMDS subdirectory. As soon as COMMAND.COM receives control, it reads the second part, `a:\doscmds`, to inform itself of its location. (This becomes the COMSPEC.) The `/p` tells this COMMAND.COM that it is the *parent* COMMAND.COM in memory. Being the parent, it will ignore any **exit** commands you might give it.

It's best not to put COMMAND.COM in a subdirectory this way, and there isn't any advantage other than to gain a better understanding. (If you try it and make an error, you won't be able to boot from the disk. In case of an error, boot from another *system* disk, then copy COMMAND.COM into the root and delete CONFIG.SYS.)

DOS Commands from BASIC

Let's conclude the topic of shells by talking some more about BASIC.

Many BASIC commands allow you to perform functions normally done at the DOS system prompt. A shell is not necessary. Here are some examples:

mkdir "\examples"	makes a subdirectory called EXAMPLES.
chdir "\scores\raiders"	makes SCORES\RAIDERS the current directory.
rmdir "\workdata"	removes the WORKDATA directory if it is empty.
files "\teamstat*.*"	displays a list of files in the TEAMSTAT directory.
files	displays a list of files in the current directory.

These commands can be entered after BASIC's Ok prompt, or by adding line numbers, can be made into statements within a BASIC program. In a program, variable names can be used in place of the quoted strings. For example:

```
10 a$="*.*"
20 files a$
```

You can also follow BASIC's **shell** command with a quoted string or variable name. Here are two examples:

```
shell "dir *.*"

a$="dir *.*" : shell a$
```

Done this way, exiting from COMMAND.COM is automatic.

Some commands and programs won't work while in a shell. Insufficient memory is the most common reason, but other conflicts may arise because some programs "break the rules" by storing data outside their assigned memory area. A little experimentation will tell you what works and what doesn't.

Don't forget. Use:

system ENTER

to exit BASIC and return to DOS.

Chapter 33 Summary

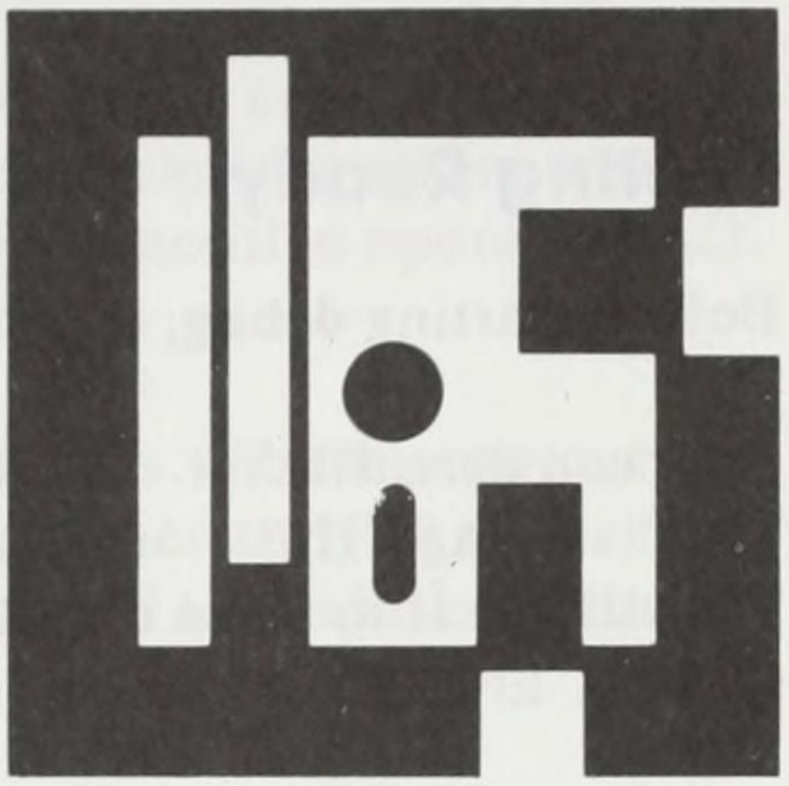
DOS allows programs to create *shells*. A shell is an area of memory into which a second program can be loaded and executed.

When you boot the system, DOS creates a shell into which it loads COMMAND.COM. COMMAND.COM is considered the "top-level" command processor. The CONFIG.SYS file may include a **shell=** entry if a top-level processor other than COMMAND.COM is to be used.

Some programs allow you to temporarily "drop into DOS" to enter commands such as **dir**, **copy**, and **delete**. They do this by creating a shell into which a second copy of COMMAND.COM is loaded. To return to the *parent* program, use the **exit** command.

CHAPTER 34

DEBUG: A Look Under the Hood



Only a tiny percentage of computer users know anything about *machine language*. Most of the rest don't need to. The next 3 optional chapters are written to give you a sample of what's "under the hood" of your computer, without getting your hands greasy. You are not going to "fix" anything, but will tinker around without breaking anything. Do **not** perform these exercises on a hard drive until you know a lot more than can be covered in this book! Use a floppy.

We won't need a screwdriver and pliers. DEBUG.COM is our toolbox. Make a safety copy of your working *system* diskette before continuing, just in case.

Why DEBUG?

Professional programmers use **debug** to test their work. It lets them watch the computer's operation while stepping through a program in slow motion, and is a tool for finding and correcting *bugs*.

We will use some **debug** as part of learning MS-DOS. Anything beyond "a peek at **debug**" is outside the scope of this book.

What you see on your screen will be different in some unimportant ways from what I see because our computers are not identical. Look for the similarities, not the differences. It won't be long before you sense what's

going on, and by the end of this first chapter you should be able to see some potential for **debug**.

Getting Ready

Before starting **debug**, let's create a file. At the system prompt, type:

copy con demofile.txt ENTER
This is an ASCII file. ENTER
We will use it to learn about DEBUG. ENTER
CTRL Z ENTER

In and Out of DEBUG

To start **debug**, simply type:

debug ENTER

The command prompt in **debug** is a hyphen. All commands are a single letter with one or more optional parameters.

The command to exit **debug** is **q**. We'll use it later.

Viewing the Registers

Enter the *registers* command at the hyphen prompt. Type:

r ENTER

debug displays the current contents of the registers, similar to:

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=093D  ES=093D  SS=093D  CS=093D  IP=0100   NV UP EI PL NZ NA PO NC
0931:0100 0000          ADD     [BX+SI],AL                  DS:0000=CD
```

Registers are temporary holding areas within the computer's brain. Each register holds 2 bytes of information, expressed as 4 hexadecimal digits.

(We'll learn about hexadecimal numbers in a minute. Your display may vary slightly.)

Notice AX, BX, CX, and DX all contain 0000. These are the main *working registers*. When adding two numbers, for example, a program may put one in AX and the other in BX. When doing a repetitive operation, CX often keeps track of the count.

SP, BP, SI, and DI are the *pointer* and *index registers*. The microprocessor uses them to point to addresses within memory. When copying a byte from one location to another, for example, it might use SI to hold the source address and DI to hold the destination address.

DS, ES, SS, and CS are the *segment registers*. They identify large regions (of up to 65,536 bytes) within memory. Right now, they all contain the same number, meaning that all the "work" the computer is prepared to do will take place within the same segment. Sometimes programs put different values in the segment registers so more than 65,536 bytes of memory can be used.

DS is the *data segment*, the region where the data being processed is located. ES is the *extra segment*, usually the destination when data is being transferred. SS is the *stack segment*. Just as you might stack papers on a table to make temporary space on you desktop, the computer can *push* data from the registers onto the *stack* in the stack segment. When needed later, it can be *popped* back.

CS is the *code segment*, the region where the program currently running is located. IP is the *instruction pointer*. It marks the location (within the code segment) of the next instruction to be executed.

Next, the eight *flags* are shown. These indicate certain conditions in the *F register*. For example, NZ means the last arithmetic operation had a non-zero result. Had the last operation given a zero, it would be ZR. Each flag is displayed as one of two opposite codes, meaningful to machine language programmers.

Condition:	OV	DN	EI	NG	ZR	AC	PE	CY
Opposite:	NV	UP	DI	PL	NZ	NA	PO	NC

The third line of the register display shows what instruction the computer will execute if given the go-ahead.

Hexadecimal

Before we continue, an explanation is in order.

We're accustomed to base-10, the decimal numbering system. In **debug**, the numbers are in *hexadecimal*, the base-16 system. Instead of the digits 0 through 9, hexadecimal uses 0 through 9 and A through F. A (hex) is 10, B (hex) is 11, C (hex) is 12, D (hex) is 13, E (hex) is 14, and F (hex) is 15.

To convert any hexadecimal number to decimal, work from *right to left*. Multiply the first digit by 1, the second by 16, the third by 256 and the fourth by 4096. Then add. E2F3 "hex" is 58099 decimal. Here's the computation:

```
hexadecimal:  E  2  F  3
                    3 x 1 =     3
                  15 x 16 =   240
                 2 x 256 =    512
              14 x 4096 = 57344

                              ------
                              58099 decimal
```

If you'd like to work through a few for practice, try these:

000B=11	001F=31	0020=32	0046=70
00A2=162	00FF=255	0100=256	0F2A=3882
1000=4096	1234=4660	8000=32768	FFFF=65535

Looking at Memory

debug's *dump* command shows what's in memory. Let's start by looking at the very lowest addresses. Type:

d 0000:0000 ENTER

Eight lines are displayed. The contents you see may be different, but they are in a format like this:

```
-d 0000:0000
0000:0000 79 2C 11 01 64 00 70 00-53 FF 00 F0 64 00 70 00  y,..d.p.S..pd.p.
0000:0010 64 00 70 00 54 FF 00 F0-53 FF 00 F0 53 FF 00 F0  d.p.T..pS..pS..p
0000:0020 E3 DB 00 F0 87 E9 00 F0-4E 08 00 CB A7 E3 00 F0  c[.p.i.pN..H'c.p
0000:0030 A7 E3 00 F0 A7 E3 00 F0-EC D0 00 F0 64 00 70 00  'c.p'c.plP.pd.p.
.
.
```

etc.

Each line displays 16 bytes of memory. The starting address for each line is on the left, so the first line is displaying the contents of addresses 0000:0000 through 0000:000F. The second line is displaying what is stored in addresses 0000:0010 through 0000:001F.

The middle section shows the hexadecimal codes for each byte. For example, the byte value at `0000:0000` is `79` (hex). The byte at `0000:0031` is `E3` (hex). (The *values* on your display may differ, but the *format* is the same.)

The right section shows the same memory contents, converted to ASCII. The `79` (hex) at `0000:0000`, for example, is decimal 121, which is the ASCII code for "y". Find the `y` on the right. `2C` is the code for comma, thus a comma follows the `y`. Some codes, such as `08`, are not *printable* ASCII characters. **debug** simply displays periods to mark their addresses.

No text is stored in memory at this point, so even the ASCII information is not meaningful to us. (Actually, this particular area of memory stores a table of addresses to be used when certain *interrupts* occur.)

IF YOU MUST KNOW: The four bytes at 0000:0000 above, for example, specify the "division by zero" interrupt. If a program attempts to divide by zero, the instructions at 010B:2C79 are executed. (Numbers are usually stored *least significant byte* first. That explains why they appear "backwards".) For the purposes of this book, don't worry about it.

Segments and Offsets

The address numbers on the left side of your screen are also in hexadecimal, so `0000:0010` is 16, `0000:0020` is 32, `0000:0030` is 48 and so on. The first part of each address is the *segment*. The second part is the *offset*. Currently, you see offsets 0000 through 007F displayed within segment 0000.

A segment is any range of 65,536 addresses starting at a multiple of 16. Having seen the first 128 bytes of segment 0000, let's look at the first 128 bytes of segment 0001. Type:

d 0001:0000 ENTER

Again, 8 lines are displayed. Here are the first four:

```
-d 0001:0000
0001:0000 64 00 70 00 54 FF 00 F0-53 FF 00 F0 53 FF 00 F0  d.p.T..pS.pS..p
0001:0010 E3 DB 00 F0 87 E9 00 F0-4E 08 00 C8 A7 E3 00 F0  c[.p.i.pN..H'c.p
0001:0020 A7 E3 00 F0 A7 E3 00 F0-EC D0 00 F0 64 00 70 00  'c.p'c.plP.pd.p.
0001:0030 65 F0 00 F0 1E D9 00 F0-2E D9 00 F0 3D 02 00 C8  ep.p,Y..Y.p=..H
```

.

.

etc.

Compare the 8 new lines to the previous 8 lines on your screen. Notice that the bytes at `0001:0000` and beyond are the same as the bytes at `0000:0010` and beyond. Segments overlap like shingles on a roof.

Ranges and Byte Counts

The *dump* command displays 128 bytes of memory, starting at the address specified. You can ask for more (or fewer) than 128 bytes by indicating a range. For instance, type:

d 001f:02a5 02ff ENTER

and see addresses 02A5 to 02FF in segment 1F.

If you insert an **l** (the letter "l") after the address, **debug** interprets the request as a count of the bytes to be processed:

d 0000:00fa l 0004 ENTER

The 4 bytes at offset FA in segment 0 are displayed.

Many **debug** commands allow a range or byte count, just like *dump*. The only requirement is that all bytes must be within the same segment. Otherwise, **debug** rejects the request.

A Ride Through Memory

How about a look at what is in each of the first 65,536 bytes? Type:

d 0000:0000 ffff ENTER or **d 0000:0000 l 0000** ENTER

> In **debug, l 0000** (the letter "l" followed by a space and four zeros) specifies a count of 65,536 bytes. That's why the second command (above) works.

As the contents of memory roll by, watch the right side of the screen. From time to time you'll see words you can recognize, such as disk file names, copyright notices, messages that MS-DOS uses, and leftover text from programs run earlier in the session. You may use CTRL S to halt and start the display, or CTRL C to cancel it before it finishes.

To see the second 65,536 bytes, use 1000:0000 for the starting address. If you have more than 128K, continue by using 2000:0000, 3000:0000, and so forth. **debug**, used this way, can help recover "lost" data. Suppose, for example, you accidentally leave **edlin** or your word processor without saving. The data may still be in memory. Use *dump* to look for it and *write* to put it on disk. (We'll learn the *write* command in a few minutes.)

Viewing a File

debug is a good way to examine and modify disk files. Unlike **edlin**, **debug** can handle binary, as well as ASCII files. It does so by loading the complete file (or a portion of it) into memory.

The first step is to use the *name* command. We'll specify DEMOFILE.TXT, the file we made at the start of this chapter.

n demofile.txt ENTER

Now that **debug** knows the name, use the *load* command:

l ENTER

Before looking at what you've loaded, use **r** for another peek at the registers. After loading a file, **debug** puts the length in registers BX and

CX. (Multiply what's in BX by 65,536 and add CX.) In this case, the display shows:

```
BX=0000   CX=003E
```

so 54 bytes have been loaded.

> On your screen, CX might be `003F`. The extra byte is an end of file code.

We didn't include a memory address with our *load* command, so the file has been loaded to the segment indicated by register CS and the offset indicated by OP. To display it, type:

d cs:100 ENTER

As a convenience, **debug** accepts **cs, ds, es,** or **ss** as the first part of an address. Leading zeros, (as in 0100, can also be eliminated). Here are the first four lines:

```
-d cs:100
093D:0100 54 68 69 73 20 69 73 20-61 6E 20 41 53 43 49 49  This is an ASCII
093D:0110 20 66 69 6C 65 2E 0D 0A-57 65 20 77 69 6C 6C 20   file...We will
093D:0120 75 73 65 20 69 74 20 74-6F 20 6C 65 61 72 6E 20  use it to learn
093D:0130 61 62 6F 75 74 20 44 65-62 75 67 2E 0D 0A 00 00  about Debug.....
```

The segment numbers on the left may be different for you.

The middle section contains the ASCII codes (in hex) for the text characters. The end of each text line is shown as `0D 0A`. These are the codes for carriage return and line feed. `Debug` is the last word in this file, so anything beyond offset 13E is just whatever happened to be in memory before DEMOFILE.TXT was loaded.

Searching and Changing

Let's change the word "Debug" to "MS-DOS". To do so, we need the exact address. You can find it by counting across the fourth line, but just for practice, how about using the *search* command to find it. Type:

s cs:100 l 3e "Debug" ENTER

The search starts at **cs:100**, and we limited it to **3e** bytes, the length of this file. **debug** lists each address where it finds a match. There's only one match in this case — at offset 0136.

The **debug** *enter* command allows you to make changes in memory. The replacement can be a string enclosed in quotes, hexadecimal byte values, or both. Type:

e cs:136 "MS-DOS." 0d 0a ENTER

Use **d cs:100** to display your work. Notice:

```
093D:0130 61 62 6F 75 74 20 44 65-62 75 67 2E 0D 0A 00 00 about Debug.....
```

has been changed to:

```
093D:0130 61 62 6F 75 74 20 4D 53-2D 44 4F 53 2E 0D 0A 00  about MS-DOS....
```

Writing a File to Disk

The *write* command can save the modified file on disk. It uses the same file name specified in the last *name* command and writes the number of bytes specified in CX.

There's no point in changing the name now, but we've lengthened the file by one byte. We need to update CX. Type:

r cx ENTER

debug displays the current contents of CX and provides a place for typing a replacement. Type **3f** ENTER.

Now you can issue the *write* command:

w cs:100 ENTER

debug displays:

```
Writing 003F bytes.
```

Press **q** ENTER to return the DOS prompt. Enter **type demofile.txt** to see the change you made.

Using What You Learned

The ability to modify files with **debug** may prove valuable. Suppose you have an accounting program that displays the word "Dividends" at the bottom of a financial statement. You want it to say "Draw," but the program provides no way to change it. **debug** to the rescue! Just remember four things:

- Make safety copies of the files before modifying them.
- If you want to modify an EXE file, rename it first, so the extension is something other than EXE. After modifying the file, change its name back. (We'll see why in the next chapter.)
- In binary and program files, you usually cannot change a word or description to something longer than the original. Programs expect certain data to be at a particular position.
- You are operating at your own risk, so test your work thoroughly. Software vendors aren't responsible for programs you've modified!

By leaving blanks in the right places beforehand, you can also use **debug** to put ANSI escape sequences into text files. The hexadecimal code for escape is 1B. Printer codes can be inserted the same way. (See your printer manual.)

More DEBUG Commands

A few other **debug** commands may come in handy. Restart **debug** and try them if you wish.

Fill

The *fill* command fills a range of addresses with a selected byte value . For example:

f cs:100 ffff 00 ENTER

fills all addresses from CS:100 to CS:FFFF with zeros, in effect, clearing the memory.

f cs:100 200 "Unused" 0d 0a ENTER

fills CS:100 to CS:200 with the word "Unused", followed by a carriage return and line feed, over and over again.

Move

The *move* command copies a block of bytes from one address to another.

m fb0:1af 1be cs:100 ENTER

copies the 16 bytes at 0FB0:01AF to 0FB0:01BE. The destination is CS:100. As with other commands, a byte count can be used instead of a range:

m fb0:1af l 10 cs:1 00 ENTER

Compare

To compare two blocks of memory, use the **c** command. For example:

c ecd0:1 00 1 0A 300 ENTER

The 11 bytes from ECD0:0100 to ECD0:010A are compared to the 11 bytes at DS:0300. (**debug** assumes DS as the segment.) If the two blocks of memory differ, the addresses and differences are listed.

Hex

The *hex* command does arithmetic in hexadecimal.

h 13fe a01 ENTER

adds 13FE to A01 and subtracts A01 from 13FE then displays both results in hexadecimal.

Enter

We've tried the *enter* command already, but a variation is available. The **e** command may be given with just an address. For example:

e ds:2f0 ENTER

The byte at DS:2F0 is displayed. You can type a replacement (in hex), then press the space bar to continue to the byte at DS:02F1, or just press the space bar alone to go to the next byte without making a change. The minus key moves you back to the prior byte. ENTER returns **debug**'s hyphen prompt.

One-Byte Registers

AX, BX, CX, and DX each hold two bytes. Actually, each working register is two one-byte registers:

AX is AH and AL
BX is BH and BL
CX is CH and CL
DX is DH and DL

If, for instance, AX contains 0B31, the AH register is 0B and the AL register is 31. Since **debug**'s **r** command won't allow changing a one-byte register, you must change the *high* and *low* bytes together. Suppose AX is 0B31 and you want to change AL to 32. The command is:

r ax ENTER

then...

0B32 ENTER

Been working the examples? Good. Quit to DOS now:

q ENTER.

DEBUG with a File Name

If you know what file you are going to examine, you may include the file name when starting **debug** from the system prompt. For example:

debug demofile.txt ENTER

This way, you don't need the *name* and *load* commands.

Use **d** and **r** to check it out. Seen enough? Quit back to DOS.

Chapter 34 Summary

debug allows you to examine and modify the computer's registers and memory. It also lets you view and modify disk files, whether ASCII or binary.

The **r** command displays the registers. To make a change, follow **r** with the name of the register. To change the AX register, enter **r ax**. Then enter a new value and press ENTER. To set the *carry* flag, enter **r f**, then **cy**.

To load a disk file, specify the name with the **n** command. Then use **l** (with an optional destination address) to load it. The default address for loading is CS:100. Registers BX and CX show the length after a *load*.

The **w** command (with an optional source address) writes data from memory to disk. BX and CX specify the length.

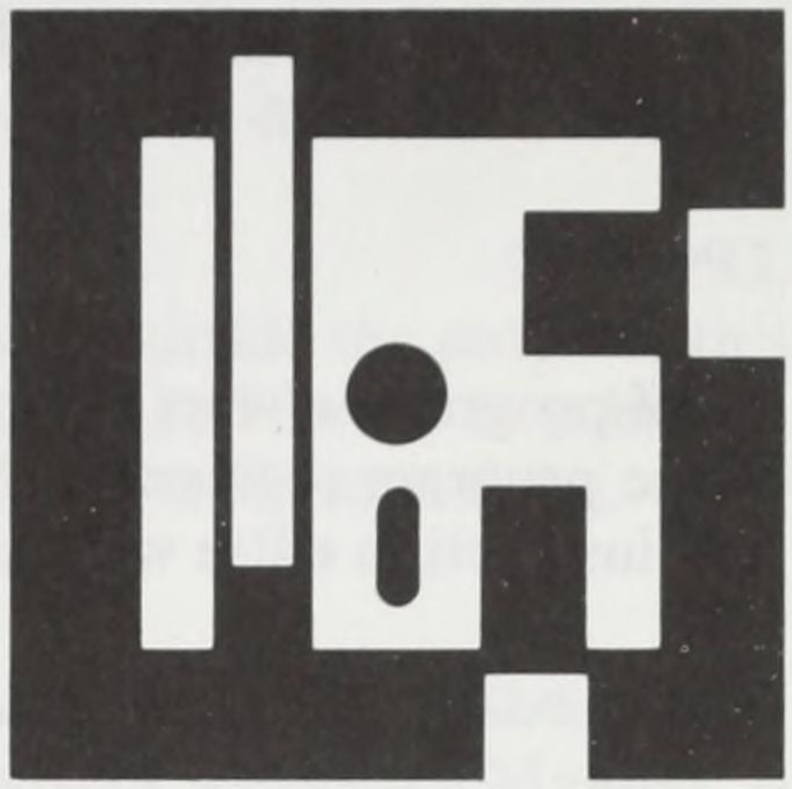
Use the **d** command (with an optional address or address range) to display memory. **d 0000:0000** displays the first 8 lines of the very lowest memory addresses.

The **e** command allows entering data at a specific address. **e 10fd:1e3 1b "[7m"** puts an escape code, followed by "[7m" at address 10FD:1E3.

Search, fill, move, and *compare* are other commands you may need. Each requires a starting and ending address, or a starting address, the letter "L", and a number of bytes. For *search* and *fill*, provide a string in quotes or a list of bytes. For *move* and *compare*, provide an address for the destination or comparison.

The **q** command quits **debug** and returns the DOS system prompt.

CHAPTER 35



Deeper into DEBUG

Now that we've had a ride through memory, an "x-ray" view of a data file, and an insider's look at the registers, let's learn what it means to DOS.

To work through the examples, make sure that DEBUG.COM, EDLIN.COM, and DEMOFILE.TXT are on your disk and in the DOSCMDS directory. At the system prompt, type:

debug edlin.com demofile.txt ENTER

This command is just like **edlin demofile.txt**, but we're also telling **debug** to load **edlin**. Notice the COM extension. **debug** requires it.

> If `File not found` appears, use **q** ENTER to exit **debug**. Once again, make sure the three files are in the current directory on your disk.

When **debug** displays the hyphen prompt, we're ready to examine **edlin** in memory.

The Program Segment

When you request a program, DOS must decide where to put it in memory. The location it selects is called the *program segment*.

Use **debug**'s **r** command. The value in CS is where the program segment begins. Notice that DS, ES, and SS have the same value. This is always the case for COM (but not EXE) programs. Also notice the value in the instruction pointer.

```
IP=0100
```

COM programs always begin execution with the instruction at offset 0100 in the program segment. The third line of the register display shows the first instruction **edlin** will execute:

```
1117:0100 E9D11C    JMP    1DD4
```

The first command happens to be a "jump" to offset 1DD4. Let's see what's there:

d 1DD4 ENTER (Use your value)

Nothing but a bunch of meaningless hexadecimal numbers!

Unassembling

They're not meaningless to the computer. You are seeing *machine language*. We can use **debug**'s *unassemble* command to convert it to something more understandable. Type:

u 1DD4 ENTER (Use your value)

Understandable? To programmers, perhaps. This is *assembly language*. (The values you get may be different from those shown below.)

```
0931:0F12 C606B22200  MOV     BYTE PTR [22B2],00
0931:0F17 BC3923      MOV     SP,2339
0931:0F1A 50          PUSH    AX
0931:0F1B B430        MOV     AH,30
0931:0F1D CD21        INT     21
0931:0F1F 3C02        CMP     AL,02
0931:0F21 7305        JNB     0F28
.
.
```

etc.

On the left, the addresses and machine language codes are shown. In the middle, you can see the instructions that **edlin**'s programmer wrote. MOV stands for MOVe. The first instruction moves 0 to the byte at address 22B2. Then 2339 is to be moved into the SP register. Next, the contents of AX are to be pushed onto the stack.

Understanding assembly language instructions is outside the scope of this book, but notice the one that reads `INT 21`. An INTerrupt 21 is known as an *MS-DOS function call.* It is a request for DOS to do something. Over 70 DOS functions are available.

In this case, the programmer has specified function 30 hex. (Notice `MOV AH,30`.) Function 30, according to the **MS-DOS Reference Manual**, is "get version number." Apparently, one of the first things **edlin** does is verify that you are using DOS version 2.0 or higher. (Notice the CoMPare: and Jump if Not Below instructions: `CMP AL,02` and `JNB 0F28`.)

The Reference Manual may be ordered from your local Radio Shack dealer.

The Program Segment Prefix

The DOS function calls make the programmer's job easier because they take care of complex tasks like disk directory management and console input/output. DOS provides a standard area in memory where a user program can find information it needs. This area is called the *Program Segment Prefix.*

This prefix is the first 256 bytes of the program segment. To display the PSP that DOS constructed for this session with **edlin**, type:

d 000 l 100 ENTER

Any parameters typed with the command to start a program can be found here. Offset 0081 tells the length. Offset 0082 has the actual text. If the parameter is a disk file name, DOS formats another copy of that name at 005D. If two disk file names are given, the second is at 006C. This is how **edlin** knows we want to edit DEMOFILE.TXT.

A user program can check offset 0002 to see how much memory is available. For example, 00 50 means all memory beyond address 5000:0000 is either non-existent or off limits.

Much more information is in the program segment prefix. The reference manual cited gives a complete rundown for advanced programmers.

DEBUG's Go Command

debug is still waiting for us to say "go." Type:

```
g=100   ENTER
```

The *go* command executes the program in memory, starting from the address specified. The program in memory is **edlin**, and we've started it from the beginning. See:

```
End of input file
*
```

just as if we had started **edlin** the normal way. Type **l** to list the file, then **q** to quit **edlin**, and **y** to confirm.

```
*l  ENTER
    1:*This is an ASCII file.
    2: We will use it to learn about MS-DOS.
*q  ENTER
Abort edit (Y/N)? y  ENTER
```

That was a quick **edlin** session. Rather than ending at the system prompt, we are still in **debug**. It displays:

```
Program terminated normally
```

Enter **q** to quit **debug** and return to DOS.

EXE vs. COM Programs

Why do some programs have the EXE extension, while others use COM?

Generally speaking, EXE programs are bigger and more complex. The decision whether to create a COM or EXE file is made by the programmer, based on how much memory will be required, and often, on the programming language being used.

We can see the difference with **debug**. BASIC is an EXE program. Let's look at it:

debug basic.exe ENTER

At the hyphen prompt, type **r** ENTER to see the registers.

The main thing to notice is that the segment registers, DS, ES, SS, and CS, have different values. If it was a COM file, they would all be the same.

The data segment is different from the code segment, which is different from the stack segment. When DOS loads an EXE file, it *relocates* parts of the program and the data it uses to separate areas in memory. This makes the programmer's job easier, especially in applications that require more than 65,536 bytes.

Look at the IP register. The first instruction is not at 0100, as it would be in a COM file. Instead, DOS sets the *entry point* according to what's specified in the EXE file.

Now you've seen some differences. Press **q** ENTER to exit **debug**.

Viewing Disk Sectors

In the last chapter we saw how **debug** can examine a disk file. It can actually look at any part of a disk. Let's view a diskette as DOS sees it. Go into **debug** again:

debug ENTER

Make sure a diskette is in Drive A and type:

l 0100 0 0 10 ENTER (First character is a lower case "L".)

This is the *absolute sectors* version of the *load* command. The four parameters are:

- destination address in memory. We selected **0100**.
- disk drive. (0=A, 1=B, and 2=C.) We selected **0** for Drive A.
- starting sector number. We selected **0**, the first sector.
- sector count. We requested **10** hex, meaning 16 sectors or 8192 bytes.

Enter **d 100** to display the first 128 bytes. You are looking at the *boot sector* of the disk. Press **d** ENTER a few times to display some more. Near offset 0280 you'll see some of the messages stored in the boot sector, such as `Non system disk or Disk error` and `Boot failure`. You'll also recognize file names: IBMDOS.COM and IBMBIO.COM.

Type **d** ENTER several times. The area is the file allocation table. To us it is just more meaningless hexadecimal codes. Enter **d 0AB0** to see the second copy of the FAT.

Forgot what the FAT is? Review Chapter 30.

Try **d 0F00** for the diskette's directory.

```
22FC:0F00 49 42 4D 42 49 4F 20 20-43 4F 4D 27 00 00 00 00  IBMBIO  COM'....
22FC:0F10 00 00 00 00 00 00 00 78-F0 0E 02 00 AD 39 00 00  .......x.....9..
22FC:0F20 49 42 4D 44 4F 53 20 20-43 4F 4D 27 00 00 00 00  IBMDOS  COM'....
22FC:0F30 00 00 00 00 00 00 00 78-F0 0E 11 00 30 6F 00 00  .......x....0o..
22FC:0F40 43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
22FC:0F50 00 00 00 00 00 00 00 78-F0 0E 2D 00 3C 5C 00 00  .......x..-.<\..
22FC:0F60 41 55 54 4F 45 58 45 43-42 41 54 20 00 00 00 00  AUTOEXECBAT ....
22FC:0F70 00 00 00 00 00 00 2E 7F-69 10 49 02 29 00 00 00  ........i.I.)...
```

Each directory entry is 32 bytes. The first 11 bytes are the file name and extension. The 12th byte of each entry is the attribute. In this case, we see `27` as the attribute byte for IBMBIO.COM and IBMDOS.COM and `20` for COMMAND.COM and AUTOEXEC.BAT.

Other information is in each directory entry, but is stored in a way that is difficult to decode:

23rd-24th bytes	time the file was created or last updated
25th-26th bytes	date the file was created or last updated
27th-28th bytes	starting cluster number for the file
29th-32nd bytes	file size

Attribute Bytes

The 12th position in each directory entry is the *attribute byte*. Press **d** ENTER a few times to move down through the directory. You might see these attribute bytes:

00 or 20	normal file
01 or 21	normal file, "read only"
02 or 22	normal file, "hidden"
03 or 23	normal file, "read only" and "hidden"
04 or 24	system file
07 or 27	system file, "read only" and "hidden"
08	volume label
10	sbdirectory

If the first digit of the attribute is 0, no changes have been made to the file since the last backup. A first digit of 2 indicates a new file, or one that has been modified. The backup program (Chapter 29) changes the 2's to 0's.

Adding a Volume Label

Now that we know the *load* and *write absolute* commands, we can do some useful things. The first project is adding a volume label to a formatted diskette that doesn't have one — something MS-DOS prior to version 3.2 doesn't normally allow.

Exit **debug**, then **format** or **diskcopy** a new disk before trying these exercises. Safety first!

Have a disk ready? Put it in Drive A and display the label:

vol a: ENTER

```
Volume in drive A: has no label
```

We know that the volume label is just a directory entry having an attribute code of 08. The first job is to create a dummy file with a length of zero bytes. This command will do it:

type nul > a:dummy ENTER

Now go into **debug** and (if necessary) put the diskette back in A. Load the directory to memory with:

l 100 0 5 7 ENTER

This command reads, "Load to CS:100, from Drive A, starting at sectors 5, 7." Now the directory is in memory from CS:100 to CS:EFF. Use **d cs:100 eff** ENTER to make sure.

This command will work on all 360K 5-1/4" diskettes, but will not work on all 3-1/2" 720K diskettes.

To find the DUMMY file, type:

s 1 00 eff "DUMMY" ENTER

If the search is successful, the address is displayed. For example:

```
1101:0720
```

Display the directory entry at the offset where **debug** found it by typing:

d 720 1 20 ENTER

```
1101:0720 44 55 4D 4D 59 20 20 20-20 20 20 00 00 00 00 00  DUMMY       .....
1101:0730 00 00 00 00 00 00 85 64-73 10 00 00 00 00 00 00  .......ds.......
```

We're ready to make it into a volume label. Type:

e 720 "My Own Disk" 08 ENTER

Replace "720" with the offset **debug** found for you. Replace "My Own Disk" with the volume label you want, but make sure *exactly* 11 characters (including spaces, if necessary) are between the quotes.

Redisplay the entry.

d 720 1 20 ENTER

```
093D:0720 4D 79 20 4F 77 6E 20 44-69 73 6B 08 00 00 00 00  My Own Disk.....
093D:0730 00 00 00 00 00 00 85 64-73 10 00 00 00 00 00 00  .......ds.......
```

Make sure the 08 is in the 12th position. If you made an error, it's best to use **q** to exit **debug** and try again.

If everything looks good, *write* the updated directory to disk:

w 100 0 5 7 ENTER

and exit:

q ENTER

Display the new volume label:

vol a: ENTER

```
Volume in drive A is My Own Disk
```

Also check the directory. No DUMMY here!

Changing a Volume Label

To change an already-existing volume label, just load the directory, search for the label, and re-enter it. Try changing "My Own Disk" to "Diskette 01". Here are the **debug** commands:

l 100 0 5 7 ENTER
s 100 eff "My Own Disk" 08 ENTER

If your typing was perfect (including case), the address and offset, such as 1101:0720 are displayed. Make the change at the indicated offset:

e 720 "Diskette 01" ENTER

Now write the updated directory and exit:

w 100 0 5 7 ENTER
q ENTER
vol a: ENTER

Making a Read-Only File

A read-only file is one that can't be erased or modified. Let's make one on the practice diskette. Type:

copy con a:readonly.txt ENTER
This is a read-only file. ENTER
CTRL Z ENTER

Load the directory again with **debug**:

debug ENTER
l 100 0 5 7 ENTER

and search for the READONLY.TXT entry:

s 100 eff "READONLYTXT"

Now display 32 bytes at the offset indicated. For example, if the address is 1101:0740, type:

d 740 1 20

Add hexadecimal B to the offset to get the address of the attribute byte. (740 + B = 74B.) Use the *enter* command to make the change:

e 74b ENTER

The old attribute byte displays:

20

If it is an odd number, the file is already "read only." If it is an even number, add 1 to make it odd:

21 ENTER

and check it again.

e 74b ENTER

Finally, (if everything went as expected) write the directory to disk, and go back to DOS:

w 100 0 5 7 ENTER
q ENTER

Check it out. First:

type readonly.txt ENTER

```
This is a read-only file.
```

Then try:

del readonly.txt ENTER

```
Access denied
```

You can't. Look for it in the directory:

dir readonly.txt ENTER

Sure enough, it's there.

Can we deliberately (or accidentally) overwrite it?

copy con readonly.txt ENTER
This is a test. ENTER
CTRL Z ENTER

```
File creation error
        0 File(s) copied
```

This is a genuine read-only file!

Unprotecting a Read-Only File

The procedure to unprotect is just the opposite. Use **debug** to load the directory. Search for the file name, locate the attribute byte and change it. If the attribute byte is odd, subtract 1 to make it even. (If the attribute

byte is even, it is already a read-write file.) Then write the directory to disk.

Go ahead and unprotect READONLY.TXT.

The DOS command to change this byte is **attrib**, remember?

Hiding and Unhiding a File

Now that we know how to change a file's read-only status, we also know how to hide and unhide. Simply change the attribute byte.

Like a read-only file, a hidden file can be typed, but it cannot be erased, copied, or renamed. Hidden files are excluded from directory listings. Some programs are able to update and modify hidden files, but most cannot.

You can hide files to reduce clutter in the directory, but it's better to use subdirectories because **debug** can be dangerous to your data if you make a mistake. Sometimes files are hidden to help prevent unauthorized snooping.

Check it out by hiding and unhiding READONLY.TXT. Change the attribute byte to 22 to hide it, then 20 to unhide it.

If you wish, try other attribute values for READONLY.TXT. Refer to page 293 for the options. When you're done, change the attribute back to 20 and delete the file.

---

debug can modify the hard drive's directory, too, but be *extra cautious*. Since the FAT and directory are bigger, you need to change the parameters for the *load* and *write* commands. Best leave it alone!

Surprise, Surprise!

Now that we've done a little **debug** the hard way, you'll be happy to learn that "utility programs" are available to make all this a snap. We'll learn about the *Norton Utilities* and *PC Tools* in Part 8.

Advanced DEBUG Commands

debug has five other commands. They'll be of interest to advanced programmers:

Command	Parameters	Examples
Assemble	address	**a cs:100**
Go	address list	**g=100 1fe 20A 1b20**
Trace	address	**t=100 5**
Input	port	**i 2f8**
Output	port, byte	**o 2f8 ff**

Assemble takes the assembly language code at the specified memory location and puts it into a format the computer can execute.

The *go* command, when followed by a list of up to 10 addresses, causes the program in memory to be started. As each address is encountered, **debug** halts the program and displays the current registers. You can press **g** ENTER to continue to the next *breakpoint address* in the list.

Trace is similar to *go*, but it displays the registers as each instruction is executed. A second optional parameter tells how many instructions to trace. Enter **t** without parameters to trace one instruction at a time.

Input and *Output* allow access to the computer's ports. *Ports* are used for sending and receiving information from devices, such as the printer and video display.

Chapter 35 Summary

When a program is loaded, MS-DOS decides where to put it in memory. The area it selects is called the *program segment*. The program segment prefix occupies the first 256 bytes of the program segment. This is where a program, while running, can "see" what parameters have been entered, how much memory is available, and more.

To run a COM program, DOS just copies it from disk into memory and sets the instruction pointer to 0100. To run an EXE program, DOS *relocates* the program logic and data and sets the instruction pointer according to information it finds in the EXE file.

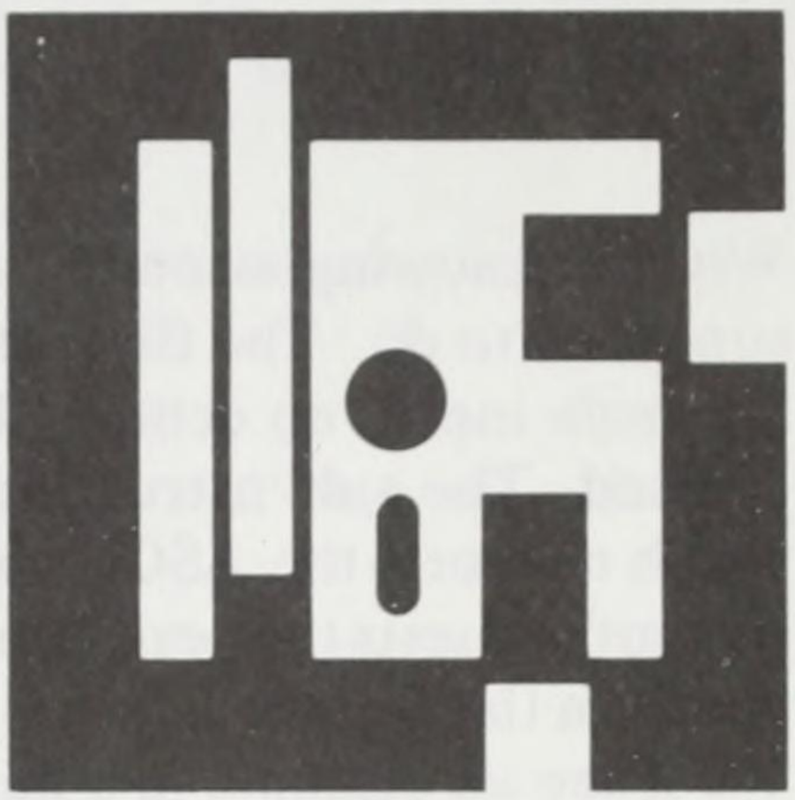
The computer understands *machine language*. To us, machine language is meaningless. We can use **debug**'s *unassemble* command to translate machine language to *assembly language*. Advanced programmers are able to read assembly language. For additional help in following the logic of a program, they can use **debug**'s **g, t, i,** and **o** commands.

Programs are able to perform common operations such as waiting for a key to be pressed or searching the directory by issuing *DOS function calls*. Such a request is called an *interrupt 21*.

debug makes it possible for us to view a disk as DOS sees it. The *load* command, when entered with a memory address, drive number, starting sector number, and sector count, copies *absolute sectors* from a disk into memory. The *write* command, with the same parameters, copies what's in memory back onto a diskette.

CHAPTER 36

Where Do Programs Come From?



In the last chapter, we saw what a program looks like in memory. In this chapter, we take a brief look at the many ways programs can be written under MS-DOS.

Assembly Language

Assembly is called a *low level language* because its instructions translate directly into microprocessor instructions. When programming in assembly language, we tell the computer's CPU (Central Processing Unit) exactly what to do.

Let's create a short program using assembly language. We can use **debug**'s built-in assembler. First, go into **debug**:

debug ENTER

Now enter the *assemble* command:

a 100 ENTER

and the assembly language instructions:

mov ah,8 ENTER
int 21 ENTER

```
sub al,30   ENTER
mov ah, 4c   ENTER
int 21   ENTER
ENTER
```

Without knowing assembly language, it's hard to tell what this program is supposed to do. The first interrupt (**int 21**) requests MS-DOS function **8**, "console input, no echo." This causes execution to wait for a key to be pressed. The **sub** instruction subtracts **30** hex from the key's ASCII code, which converts the ASCII character into a numeric value. The second interrupt requests the "exit" function. The program terminates, and the number from the depressed key is returned for use in an **if errorlevel** statement. We'll see an example in a minute, but first, we must save the program.

```
r cx   ENTER
000a   ENTER
```

That specifies the length. Now we name it:

```
a selectit.com   ENTER
```

and write it to disk:

```
w   ENTER
```

Use **q** to quit **debug**.

Check the directory to find SELECTIT.COM.

Using the New Program

We used **debug**'s assembler to create a powerful new command for making *menus* in batch files. SELECTIT.COM waits for a number key to be pressed and sets the error level accordingly. Review this example to see how it can be used:

```
echo off
:again
echo Press [1] for directory or [2] to check disk.
selectit
if errorlevel 3 goto again
if errorlevel 2 goto two
if errorlevel 1 goto one
goto again
:one
dir
goto end
:two
chkdsk
:end
```

Notice `selectit` in line 4. It executes our new program, SELECTIT.COM. Then the `errorlevel` tests in lines 5, 6 and 7 act upon the result. Depending on which key is pressed, a directory is displayed, **chkdsk** is executed, or you are told to try again. Use **edlin** or **copy** to create this file as SELECDEM.BAT. Then just type **selecdem** ENTER.

The Assembler

SELECTIT.COM is only 10 bytes long. Most programs are *much* more complex.

debug isn't suitable for writing complex assembly language programs. It's just too difficult! Instead, assembly language programmers use **edlin** (or some other text editor) to create their *source file.*

The source file contains assembly language instructions (like we used), but it may also contain *comments* and *labels*. Comments help the programmer remember what he intended each instruction to do. Labels identify data storage areas, destinations for *jump* commands, and other interesting locations within the program. Other instructions in the source file specify how data and program logic is to be grouped in memory segments when the program is executed. Assembly language source files are usually saved with the ASM file extension.

After the source file is created, an assembler is used. It reads the text in the ASM file to create an *object file*. The assembler is a separate program, not included with DOS, but just to get the idea, let's "sit in" on a session to see how a source file named READKEY.ASM might be assembled.

```
A>masm    ENTER
Microsoft MACRO Assembler  Version x.xx
(C)Copyright Microsoft Corp

Source filename [.ASM]: readkey   ENTER
Object filename [READKEY.OBJ]:   ENTER
Source listing [NUL.LST]:   ENTER
Cross refernce [NUL.CRF]:  ENTER
```

In this case, the operator pressed ENTER to accept the defaults (shown in brackets) for the object file, source listing, and cross reference.

Assuming there are no errors in the source file, **masm** (the name of the Microsoft Macro Assembler program) creates and saves an object file named READKEY.OBJ.

The Linker

After assembling, the result is an *object file*. Because object files cannot be executed, another step is needed. This is where the *linker* comes in.

LINK.EXE is the linker program included with DOS. To link READKEY.OBJ, here's how it might be used:

```
A>link    ENTER

   Microsoft Object Linker Vx.xx
(C) Copyright by Microsoft Inc.

Object Modules              readkey   ENTER
Run File                      ENTER
List File                   ENTER
Libraries                 ENTER
```

If all goes well, READKEY.EXE is created and saved to disk, ready to run.

In this example, **link** just checked READKEY.OBJ and created a new file in EXE format. In practice, advanced programmers often use the linker to link several object modules together to form a larger EXE program. Sometimes data areas or program routines are contained in one OBJ file and used by another. It's the linker's job to see that all the needed modules are present and to "fill in" their addresses so they may be found when the program is executed.

To link more than one file, the programmer lists the file names (separated by "+" symbols) next to the `Object Modules` prompt. For more information on this and the other linker options, see the **MS-DOS Reference Manual** for your computer.

LIB.EXE

Sometimes programmers have dozens of commonly used OBJ files. LIB.EXE is provided with DOS to combine them into a single *library file*. This reduces clutter on the disk and simplifies linking. The name of the LIB file can be entered at the linker's `Library` prompt instead of listing all the object files.

Here's a sample session which creates a library file called SCRCALLS.LIB, containing two object files, INPUTFRM.OBJ and COLORTBL.OBJ:

```
A>lib  ENTER

Library name: scrcalls  ENTER
Library does not exist, Create?  y  ENTER
Operations: +inputfrm+colortbl  ENTER
List file: prn  ENTER
```

Next to the `Operations` prompt, the object files to be added are listed, each preceded by a "+" symbol. Object files can be deleted from a library with "-", or extracted with "*". Specifying **prn** as the list file sends a useful cross reference to the printer.

Creating a COM File

The linker always creates an EXE file. To create a COM program, two more steps are required. Suppose we created READKEY with the inten-

tion of making it a COM file. After the linker has created READKEY.EXE, the DOS **exe2bin** program is used (not available in all DOS versions):

exe2bin readkey ENTER

exe2bin removes the relocation information from an EXE file and converts it to binary format. In binary format, the file contains an exact image of what will be loaded to memory. The result is READKEY.BIN.

The final step is to:

rename readkey.bin readkey.com

and it's ready to run.

Not all EXE programs can be converted to COM programs — in fact, most cannot. The program logic and data in the EXE file must be less than 64K, and there must be no stack segment. These are technical matters, so check the **MS-DOS Reference Manual** if you have reason to use **exe2bin**.

Higher Level Languages

COBOL, FORTRAN, Pascal and BASIC are *high level languages*. Compared to assembly language, they are much easier to use and are much more standardized among different computer types. In contrast to *low level languages*, they use English words instead of microprocessor instructions (as assembly language does). The programming words in the higher level languages represent procedures, often consisting of hundreds of assembly language instructions — all predefined by the language to make programming easier.

Intermediate level languages, such as C, are a mixture of both high and low level languages. C combines the best of both worlds, allowing a program to use detailed microprocessor instructions as well as predefined words, or routines.

A text editor is used for entering the program in whatever language the programmer has selected. A special program called a *compiler* then converts the source file (text) into an object file, which can be converted to the executable program. Each language has its own compiler.

The language a programmer selects depends on several things. FORTRAN is preferred for scientific applications. COBOL, because it is often used on mainframes, may be favored for business applications. Commercial software developers may select C for its efficiency. Pascal might be chosen because of its structure. BASIC, on the other hand, might be chosen for its ease of use and *lack* of structure!

Each language has advantages and disadvantages. The choice often depends on what's most practical:

- What language are the other company programs written in?
- What compilers/interpreters are available?
- What language is the programmer most familiar with?

The BASIC Interpreter

You are already familiar with the BASIC interpreter. It is BASIC.EXE on the *system* disk. BASIC.EXE contains hundreds of machine language routines. When a BASIC program is running, it interprets each statement to determine which of the prewritten routines to execute. The beauty of *interpretive* BASIC is that you can enter program statements and run them immediately. There is no need to compile, assemble, or link.

Learning BASIC is the logical starting point if you wish to write programs for yourself. With what you now know about your computer and DOS, you have a tremendous head start!

For an excellent tutorial, see **Learning BASIC for Tandy Computers** (Cat. No. 25-1500) by David A. Lien.

Chapter 36 Summary

Programs are written in *programming languages*.

Assembly language is the most difficult, but the resulting program is usually compact and very fast. Assembly instructions correspond to operations the "machine," or CPU, is able to perform. Assembly language programs can be entered with **debug**'s *assemble* command, or they can be stored in a file and assembled with an assembler program.

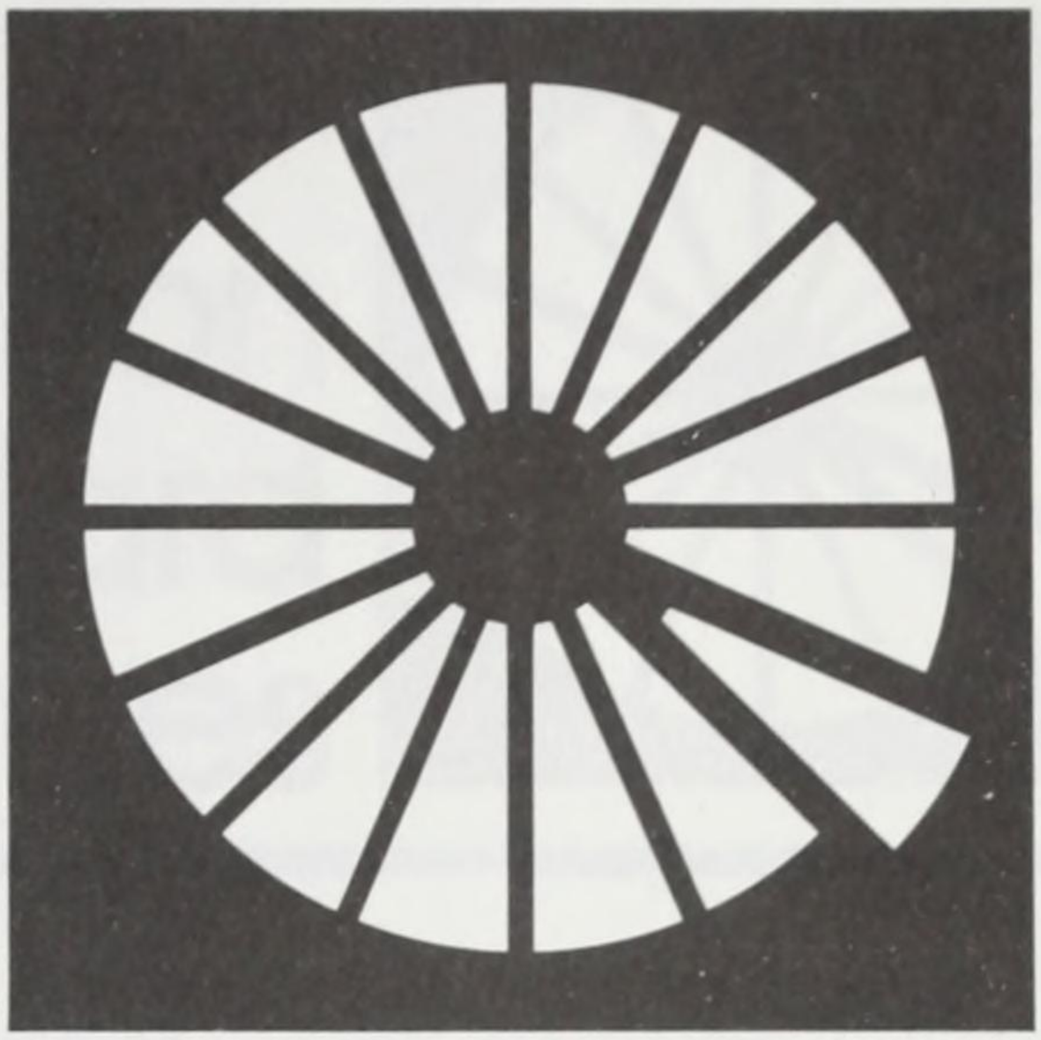
A *compiler* translates a program written in a high level language to an *object file*. The *linker* combines one or more object files, and sometimes prewritten routines in a *library file*, to produce an EXE program, which can be run at the system prompt.

Some EXE programs may be converted to COM programs. **exe2bin** creates a binary file from an EXE file. The BIN file may be renamed with a COM extension.

The *higher level languages* are COBOL, FORTRAN, Pascal, C, and BASIC. They are easier to write programs in because they understand words and abbreviations instead of numbers, closer to the way we think. A single statement in a higher level language may represent hundreds of assembly language instructions.

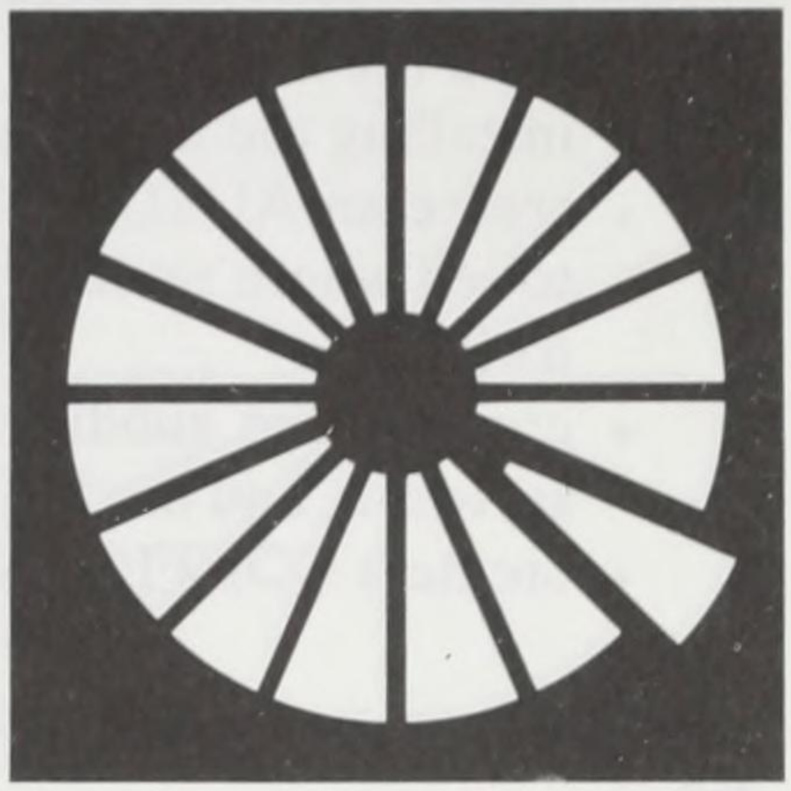
The *BASIC interpreter* allows you to write and run programs without compiling, assembling or linking — and uses descriptive English words. It is often the quickest way to create a custom program.

PART 8



Hard drive organization

CHAPTER 37



Introduction to Hard Drives

Each hard drive installation is slightly different from every other. The reason is that the hard drive can hold so much information it allows the user lots of flexibility.

Despite that caveat, I'm sure I can "hand-hold" you through your hard drive setup, even if what I tell you is only 99% applicable. The elements in common far exceed the differences. You do have to be a *little* bit flexible!

Actually, the biggest variations come in the mechanical installation of the hard drive itself. If your hard drive is already installed, fear not. You will enjoy this section, and will be amazed how what you have learned in this book comes together in such a useful and practical way.

I have to assume that your hard drive is already installed in the computer. If not, return here after either you or your Tandy Computer Center or Radio Shack Store complete the installation, and we will get on with learning how to organize and obtain maximum benefit from a hard drive.

The Objective

In this section, I will show you how to take a new, hard drive equipped computer fresh from the box and:

- **format** its hard drive,
- make a subdirectory to store MS-DOS, by copying MS-DOS in from the diskettes, placing COMMAND.COM in the root, and installing the invisible *system* files.
- create an AUTOEXEC.BAT file which will change the cursor to tell where we are, and set **path**s to find software in subdirectories,
- create more subdirectories for specific software, load in that software, and modify the AUTOEXEC.BAT file to access it,
- create a CONFIG.SYS file for specific software,

plus many more helpful techniques the average user needs for everyday use.

Sometimes this part will seem like review. That's because I have incorporated a running review of the most important DOS features you have already learned. It will help you separate the really important features from those which are less important. Return to earlier chapters for review, if necessary.

Some additional popular software is called for. You don't really need to have any of it in order to learn, but the more you have, the better. This book is about MS-DOS, so perform all the DOS instructions and as many of the software-specific instructions as you are able.

Having set up countless computer systems over the years, my approach here reflects some strong opinions about how it should be done. I recognize, however, that the "generic" approach we take here is not the only way to organize a hard disk. Learn this "clean and mean" way first, and you will understand the power of MS-DOS on a hard disk. If you find another way you prefer later, be my guest.

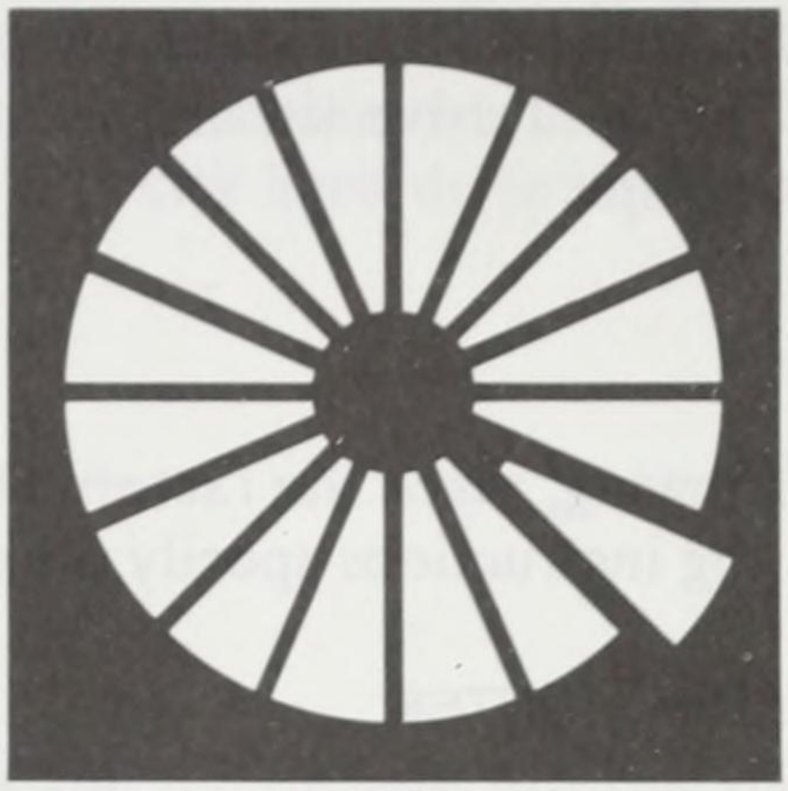
Meanwhile, in this final part of the book I will be a little simplistic, but VERY specific. Are you ready?

Chapter 37 Summary

Although hard drive installation varies among the different hard drives, there are more similarities than differences.

CHAPTER 38

Formatting the Hard Drive



Specific hard drive formatting instructions are furnished with each drive. The instructions vary widely, usually allow far too many options, and require too many decisions for the average user. We are not going to intellectualize the process here, we're just going to charge straight ahead, selecting those options that meet the needs of 98% of all computer users. The other 2% don't need to read this book anyway.

I assume you have studied this book up to this point, and know how to run your computer. You also need to be comfortable using basic **edlin**, as taught earlier. If you are rusty, better brush up on it right now.

It *is* necessary that you study the hard drive instructions that accompany your computer because the commands might be slightly different than the "majority opinion" shown here, or there may be an *automatic* installation program that is completely different from what follows. Consistency in hard drive formatting is not one of its hallmarks. But after the formatting is done, the rest of what we learn is fairly consistent.

To Start With

Do *not* add a hard drive to a new computer until you have that computer running without it. Start out very simply, with just a floppy drive, and get the monitor working so you can tell what's going on. Without the bare computer running properly, troubleshooting a hard drive installation can

be very tough. We will assume your hard drive is smaller than 33 megabytes.

The hard drive is named C, and the floppy drive is named A. With the prompt:

```
A>
```

showing, place the factory DOS diskette in Drive A. Unless your formatting instructions specify otherwise, type:

fdisk ENTER

While the questions on the screen will vary slightly from computer to computer, the answers are the same.

Several preliminary questions ask about *partitioning* the disk. On drives of less that 33 megabytes that usually means "dividing it into different sections" in order to run different versions of DOS, like MS-DOS on one half and XENIX on the other. That's pretty esoteric and not what most users need to be concerned about:

- We do *not* want to partition the disk into a number of different pieces, but do have to partition it into *one* piece (if that makes any sense). Answer the partition option question with **yes**, we want to create a partition, but only **1** .

- **Yes**, we want to use *all* the space on the hard drive.

The Actual Formatting

With these preliminaries complete, just as with a floppy disk, using the *system* option, type:

format c:/s ENTER

A precautionary message warns of an impending apocalypse if you continue. But unless the store did it for you, we do need to format the disk, so answer:

y ENTER

The actual formatting will now continue to completion, taking from several to tens of minutes. Different computers report the formatting progress in different ways, but if you've followed these instructions and feel confident about your responses, don't suspect a formatting problem for at least 15 minutes on a 20 megabyte drive. A larger capacity hard drive can take longer.

When the magic words:

```
Format complete
System Transferred
```

or the equivalent appear and the noise stops and the prompt returns, the formatting is complete.

Congratulations! Change the default drive to C by typing:

c: ENTER

and

dir ENTER

to see if you are really there.

Copying MS-DOS to the Hard Drive

At this point, most users innocently make a big mistake. They simply copy the MS-DOS floppy diskette(s) from Drive A into the hard drive root directory and begin creating a disorganized mess that only grows worse with time.

A major reason you bought a hard disk was to store a *lot* of information. That's fine. But if warehouse owners stored their inventory the way computer users store files, they would never find anything. Lots of computer users can't find anything because files from different software are mixed together, sometimes conflicting and causing crashes.

It is at this point we begin the discipline of *hard drive organization* by starting our subdirectory system. As you might suspect, efficient use of subdirectories is the key to effective hard drive management.

Type:

md dos ENTER

which means, Make a new subDirectory, and name it DOS.

Now, type:

cd dos ENTER

which Changes or moves us into that new subDirectory named DOS.

Do you have any indication that we have moved out of the root directory? No? Well, we'll fix that in just a minute. In the meantime, to keep from getting lost, just type:

dir ENTER

and note that although no files are found because the DOS subdirectory is empty, the title at the top says `c:\dos`. We are located in the subdirectory named DOS.

With the factory *system* diskette in Drive A, type:

copy a:*.* ENTER

which copies all the factory MS-DOS software program into the DOS subdirectory on the hard drive. (If it comes on several diskettes, copy them all in.)

Type **dir** to be sure MS-DOS made it to the hard drive.

Return to the root directory with:

cd ENTER

Was the System Installed?

We initially assumed that the visible and invisible *system* files were installed as part of the hard drive installation and formatting process. If that's not the case, or if you're not sure, it is necessary to:

- Install the *system* files in the root, change to Drive A, and:

  sys c: ENTER

- Copy COMMAND.COM to the *root* from A:

  copy command.com c: ENTER

- Remove the diskette from Drive A and reboot the computer.

If in doubt, no harm is done by installing these 3 files more than once.

Fixing the Prompt

Before doing anything else, let's change the prompt so it will tell us where we are on this big hard drive at all times.

Move to the root of Drive C. If you are not sure where you are, or have not followed these instructions exactly, type **dir**. The root directory should bear the title:

```
C:\
```

If not, **cd** again.

We need to create a *batch* file with a specific name that is reserved by DOS. The computer looks for this special file named AUTOEXEC.BAT and any instructions it might contain each time it is *booted*. Very carefully type:

copy con autoexec.bat ENTER

Nothing will happen, but the prompt dropped down one line. Continue the careful typing:

prompt=pg ENTER

Terminate this session by holding down the CTRL key and tapping Z once. Then press ENTER.

Now type **dir** ENTER to confirm that the new file named AUTOEXEC.BAT actually exists, then type:

type autoexec.bat ENTER

to confirm its contents. The last line should read:

```
prompt=$p$g
```

Assuming it's OK, it is necessary to reboot the computer so it will read this special new file. Any change in the AUTOEXEC.BAT file must *always* be followed by a reboot, since that is the only time this file is read. To reboot, press the CTRL ALT DEL keys all at the same time, or press the reset button.

Now look at the prompt. Instead of just reading C>, it tells where we are:

```
C:\>
```

We are in the root directory. Put this new prompt reporting system to the test by returning to the DOS subdirectory to see what happens:

cd dos ENTER

and there we are. It says:

```
C:\DOS>
```

The more complicated our *tree structure* becomes by the addition of more subdirectories, the more valuable this new prompt becomes. Return to the root by:

cd ENTER

and move on to the next chapter.

Chapter 38 Summary

Formatting instructions for hard drives vary widely.

fdisk prepares the hard drive for formatting and lets you decide how many partitions to create on the hard drive. Typing **1** creates one.

format c:/s formats Drive C and automatically copies the 3 *system* files (IBMBIO.COM or IO.SYS, IBMDOS.COM or MSDOS.SYS, and COMMAND.COM) to the hard drive.

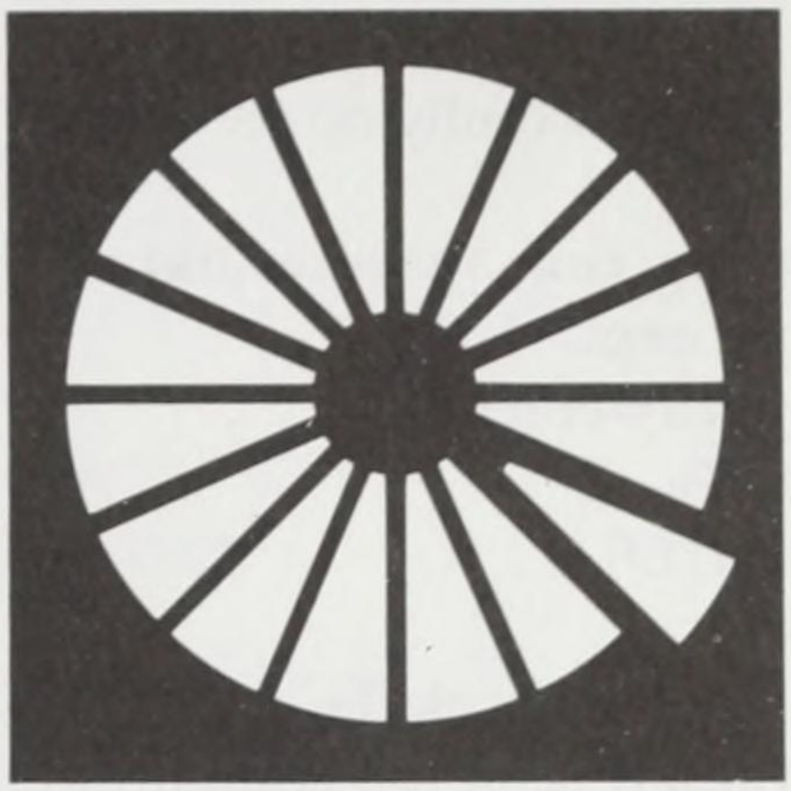
Subdirectories keep the hard drive organized. Type **md dos** ENTER to make a subdirectory named DOS, **cd dos** ENTER to change to that subdirectory, and, with the factory *system* diskette in Drive A, **copy a:*.*** ENTER to copy all factory MS-DOS files to the DOS subdirectory.

If the invisible *system* files were not installed, type **sys c:** ENTER, then **copy command.com c:** ENTER with the *system* disk in default Drive A.

cd ENTER returns you from a subdirectory to the root directory.

Adding **prompt=pg** to the AUTOEXEC.BAT file creates a prompt that tells you where you are, in the root directory or in a specific subdirectory.

CHAPTER 39



Setting the Path

As you may have discovered, computers cease to be fun if you can't access files without continually changing diskettes. That's part of the glory of the hard drive. But you'll have the same frustration if the files end up hidden in some obscure subdirectory on a hard disk. One problem with putting MS-DOS in a subdirectory is when we need to use features such as **format** or **chkdsk**, we can't get at them without shifting into that DOS subdirectory.

To demonstrate the point, from the root, type:

basic ENTER

to try to enter the BASIC computer language that comes with DOS from Tandy. See what happens? Only an error message. We can't get to there from here.

Using **cd** to change to the DOS subdirectory in order to access BASIC or any other file might not seem overly burdensome at this time, but becomes a problem very quickly as we add more files to the disk. Installing a single new major software program can easily add a hundred files. And since the software may be new to us, we probably don't know which files do what.

The solution to solving the subdirectory access problem involves putting a **path** command in the AUTOEXEC.BAT file. At this point we can't even get to the **edlin** editor from the root to change our AUTOEXEC.BAT file

since **edlin** is also in the DOS subdirectory. We'll have to use the less powerful **copy con** once more:

Very carefully now:

```
copy con autoexec.bat
prompt=$p$g
path=c:\;c:\dos
CTRL Z
ENTER
```

Double check the AUTOEXEC.BAT file contents with the **type** command, then reboot the computer.

Now type:

basic ENTER

and see the BASIC language interpreter ready for use. Since BASIC is not what this book is about, leave BASIC by typing:

system ENTER

to return to the root directory.

It is one thing to organize the warehouse, but quite another to make a map showing the paths to where things are stored. So far, we have instructed the computer to look first for a file in the root directory, specified by the **path** command `C:\`. If the file is not found there, look for it next in the DOS subdirectory, specified by `C:\DOS`. Look carefully and see how ; is used to separate the 2 **path** commands.

Adding High Powered Help

As you know by now, DOS is neither the easiest nor the friendliest software to use. You may be surprised to learn that neither is it the most powerful. This has brought to market some additional software called *utility programs,* which I consider absolutely essential, especially for hard drive users. They are reasonably priced and I highly recommend them. Without elaborating on their seemingly endless features, among the most important things these utilities can do are:

- "resurrect" files which you accidentally deleted,
- rearrange files in alphabetical order,
- rearrange files in date or time order,
- rename files, disks, and directories,
- draw a good map of the drive organization,
- provide simpler, alternate DOS commands,
- check drive speed for troubleshooting,
- run other tests on the computer itself,
- "unfragment" and compress files for speed and space
- "prune" and "graft" entire directories on the tree

and many others, any one of which can justify the purchase. If it sounds like I'm sold, it's because these extra *utility programs* have saved my bacon countless times, not to mention hundreds of hours of time and frustration.

In order to proceed with this book, we will prepare subdirectories for 2 such disk utility packages, to be copied in as soon as you purchase them:

The Norton Utilities, and
PC Tools

They are available at every good computer store, discount software store, Tandy Software Express, and direct from their manufacturers. Each should cost well under $100, and you should obtain both. If you can only afford one at this time, buy *PC Tools* first.

You may choose to purchase other utilities, and there are other very good ones, but please follow my instructions here faithfully or the rest of the book might not hang together for you.

To prepare for these new utilities, **cd** to DOS and create 2 new sub-sub-directories:

md nu ENTER
md pc ENTER

Assuming you have this *utility* software on hand, **cd** to NU and **copy** in the *Norton Utilities* from its floppy disk(s).

Return to the DOS subdirectory, then **cd** to PC, and **copy** in *PC Tools*.

Now what do you think we have to do? Right! We have to adjust the **path** instructions in the AUTOEXEC.BAT file. You will need to access these utilities from new programs, which you will store in every obscure corner of the hard disk, so they must be readily available from everywhere.

Since each subdirectory must be reachable from the root, each path must be specified from the root. From any subdirectory we can always return immediately to the root by typing:

cd ENTER

Do it. Now use **edlin** to edit the **path** part of the AUTOEXEC.BAT file so it reads:

path=c:\;c:\dos;c:\dos\nu;c:\dos\pc

Make these and all other DOS related changes even if you don't yet have the required utility software.

The **path** command now contains 4 separate commands, separated by semicolons. When a file is sought, if the computer doesn't find it in the subdirectory where you are presently located, it will search first in the ROOT, then the DOS directory, then in the NU directory, then in the PC directory.

Since AUTOEXEC.BAT was changed, reboot and we'll do some simple but helpful things with these utilities.

Not every subdirectory we create will need to be in the *search path* — only the DOS files, special utility files, and certain major programs that have subdirectories added to them.

Sub and sub-subdirectories are still just directories. Once familiar with the concept, except for instructional purposes, there is usually little point in identifying them as other than simply *directories*. So from this point on, we will usually just say *directories*.

Using Special Utilities

Take a directory of the root. Observe that the files are not in alphabetical order. Also, the DOS directory name is mixed in with the file names. Type:

ds n ENTER

and take the root directory again.

> **ds n** stands for Directory Sort, by Name. It's a feature of *Norton Utilities*. The instructions that come with *Norton* explain each of its many other features and commands. *PC Tools* has a similar alphabetizing feature.

The text part of the directory now reads like this:

```
Volume in drive C:  has no label
Directory of C:\

DOS          <DIR>              x.xx.xx    x.xxx
AUTOEXEC.BAT          xx    x.xx.xx    x.xxx
AUTOEXEC.BAK          xx    x.xx.xx    x.xxx
COMMAND.COM           xx    x.xx.xx    x.xxx
      xx File(s)     xxxxxxxx bytes free
```

> NOTE: The installation software which comes with some hard drives automatically adds another file or two to the root. So does certain other software, such as Tandy's *DeskMate*. If so, there is no problem. Just make the additions mentally as we go along.

The DOS directory, which has no extension, now appears at the top. The files with extensions follow, in alphabetical order.

Now let's **cd** to the DOS directory and alphabetize its files.

Then alphabetize directories NU and PC.

Return to the root and type:

pc ENTER
ENTER
f3
d ENTER

and see a chart of the tree structure, showing the root directory and its subdirectories. No matter how complex you make the file structure, this sequence of commands from *PC Tools* will draw a clear chart and instantly clarify the directory structure for you.

Press ESC to return to the prior *PC Tools* menu.

You may have noticed that Drive C has no label or name. (If you didn't, look again at the above directory listing.) Let's name the drive after you. Type the following commands and put in your name. You can use up to 11 characters, but avoid periods, commas and certain other punctuation.

r ENTER
(Type your name here.)
ENTER
ESC
y

Take the directory and see your name as the label for Volume C.

The last utilities were features from *PC Tools*. The instructions that come with it explain each of the many other features and commands. *Norton* has similar features. MS-DOS has a **tree** command, and version 3.2 and later have a **label** command.

Note that because of the way we set the **path** in the AUTOEXEC.BAT file, you can call up either of these *add-on* utility programs at any time. They do what we have already done, plus dozens of other powerful things, most of which are either impossible using DOS by itself, or possible only with much more tedium and with less satisfying results.

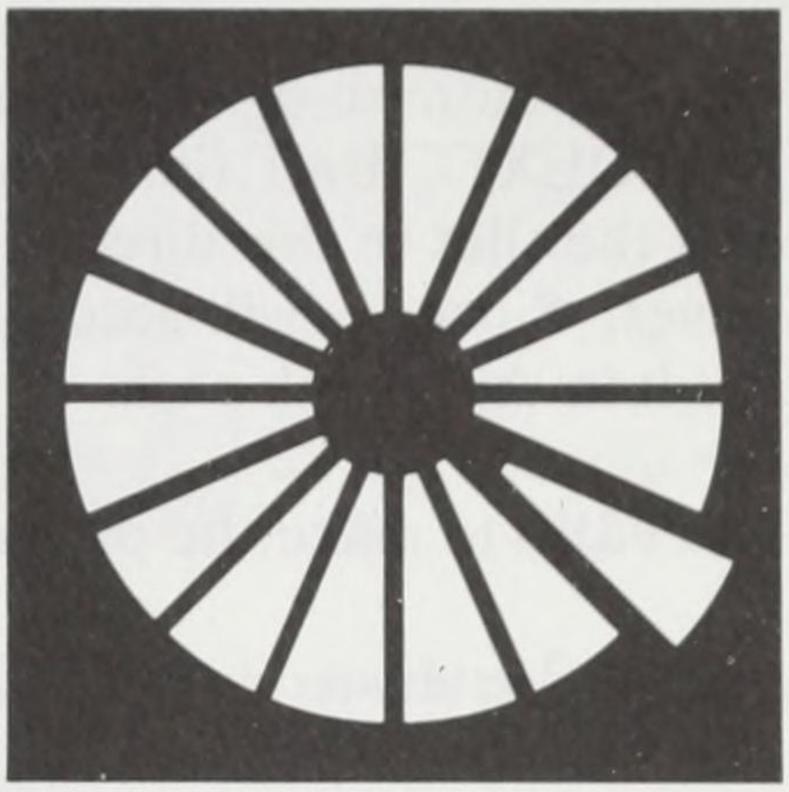
One caution. As you learn to use these utilities on your own, please do not make any additional changes to what we are doing here. To do so may arrange your files so they will not coincide with what we have to do in the rest of this important section.

Chapter 39 Summary

Adding **path=c:\;c:\dos** to the AUTOEXEC.BAT file instructs the computer to first look for a file in the root directory and, if it can't find it, to search the DOS subdirectory.

Utility programs, such as *The Norton Utilities* and *PC Tools*, provide powerful features not available with MS-DOS. When adding them to your hard drive, place them in sub-subdirectories in the DOS subdirectory and rewrite AUTOEXEC's **path** command to direct the computer to search these directories.

CHAPTER 40



Adding User Software

BASIC Language

An important piece of user software comes with DOS. It is the BASIC language interpreter and is now stored in the DOS directory. Whether or not you plan to do a lot of programming in BASIC, we'll add a special subdirectory at this time to hold the BASIC software and future BASIC programs. If you don't choose to use BASIC, you can delete it after we are done with this section and you understand what you are doing.

OK, where should we add it? As another directory off the root? That's as good a place as any — for now.

From the root:

cd dos ENTER

Take a directory of DOS and find the programs named BASIC and/or BASICA. We want to move them to a new directory named BASIC by a process of first copying, then deleting them from the DOS directory. Return to the root and:

md basic ENTER
cd basic ENTER
copy c:\dos\basic?.* ENTER

Now return to DOS to delete these **basic?.*** programs. Then return to the root.

Oh, one more thing. Do you suppose we'll have to change anything in the AUTOEXEC.BAT file? That depends on whether or not we want to access the files in this directory from other parts of the drive or from other drives. Since we will access BASIC from the root from time to time, let's put it in the search path.

Use **edlin** to make the path read:

path=c:\;c:\dos;c:\dos\nu;c:\dos\pc;c:\basic

Reboot, and we're set to add more software.

Word Processing

If you are like most average hard drive users, you will install a word processing program. Since I wrote this book using Microsoft *Word*, we'll install *Word*. If you use a different word processor such as Tandy's *Varsity Script*, install that instead. As far as MS-DOS and the hard drive are concerned, there is not much difference between word processors.

We first need to add a directory off the root. Type:

md word ENTER

then

cd word ENTER

and follow the installation instructions that come with your word processor. In the case of Microsoft *Word*, change the default drive to A, insert the *Utilities* diskette, type **setup** ENTER, and follow the instructions on the screen.

NOTE: If you install another word processor, say *Varsity Script*, and choose to name the directory VS, don't forget to make the corresponding name changes throughout the rest of this book. If your word processor requires placing **device** commands in a CONFIG.SYS file, do so.

After the word processing software is installed, proceed to make several subdirectories off WORD to hold different kinds of documents. From WORD, let's make 3, and name them:

BIZ
LEGAL
PERS

for business, legal, and personal. Add them yourself, then examine the new tree structure using either *The Norton Utilities* or *PC Tools*.

To use a word processing program properly, we always move to, then work from the sub-subdirectory which actually contains the letters and documents we create and modify. Do *not* work from the WORD subdirectory that contains the word processing program files. Failure to create directories like BIZ, LEGAL and PERS and work from them means all the documents end up mixed in with the word processing program files, and we have failed to use the power of DOS to keep our hard drive organized and manageable.

Hard drive clutter can come upon you even faster than garage or closet clutter. The net result is about the same.

Word processing is a good example of where many subdirectories having specialized uses (in this case to keep documents organized) are attached to a major piece of software, which is itself in a subdirectory. If we always work from one of these sub-sub directories, how do you suppose the computer knows enough to search the main program, in the subdirectory, to find needed program files such as spellchecking? The answer is simple. It doesn't! We have to tell it.

As with many things, there are 2 ways to do it, the hard way, and the easy way. We already know the hard way quite well. Until now, it was the only way. When in doubt, add very specific instructions to the **path** command. You could solve the above **path** problem without further help from me.

This situation is special, however, in that it involves checking for program files in a *parent* directory which is one branch closer to the root.

We have seen at the beginning of every subdirectory:

```
.       <DIR>          x.xx.xx    x.xxx
..      <DIR>          x.xx.xx    x.xxx
```

The "double dot" is an indication that this subdirectory will search its parent directory next closest to the root, if told to do so. That's our cue. We need to place a single new instruction in the **path** command, then *all* subdirectories on the entire drive will check their parent directory if they don't find what they are looking for in their own subdirectory. That's real power!

Since we earlier determined that checking the root itself was the most important path command, we placed it first in the **path**. This *parent* search path is 2nd most important, so we will place it 2nd. Return to the root and change the AUTOEXEC.BAT file to read:

path=c:\;..\;c:\dos;c:\dos\nu;c:\dos\pc;c:\basic

and reboot.

NOTE: The **path** command does not have to fit on one line. Let it flow over to as many lines as are required, but do not interrupt it with a carriage return. Be sure to type in all the semicolons.

By now you should be picking up the tune and tempo of what we are doing. There are really only 3 relevant questions to ask when adding to or rearranging disk files:

- Where on the disk do we place these new files?
- Do we need to change the path instructions?
- Do we need to change the CONFIG.SYS or AUTOEXEC.BAT files?

Storing Laser Printer Fonts

Many hard drive users in a business environment use Tandy *laser* printers. Because of special publishing needs, a special selection of printer fonts are downloaded into my printer's memory when it is turned on each day. In this section we will provide a place to store those fonts on the drive. Later, we'll create a batch file to do the downloading automatically. Do not skip this section if your Tandy *laser* printer has not yet arrived. It will, and the same techniques used here apply to many other applications.

These printing fonts are not accessed frequently, perhaps just once a day, but we bought a hard drive to minimize fussing with floppy disks so we do want to store the fonts on the hard drive. The font files can be stored anyplace out of the way, as long as we are willing to move to their directory when they are needed, or can use a **path** command to find them.

Not all **path** commands need to be stored in the AUTOEXEC.BAT file — only those which lead to directories we *always* want to be quickly available from any other directory.

Probably the simplest way to store these files is to create a directory right off the root and keep the path instruction short and to the point.

From the root, type:

md lf ENTER

then:

cd lf ENTER

then copy in the laser fonts from the floppy disks on which they are supplied.

Since the LF directory does not need to be accessed often, nor from different places, and its presence on the hard drive is just a matter of convenient storage, we will *not* change the AUTOEXEC.BAT file. We will issue a **path** command, however, at the appropriate time and place. Stay tuned.

Return to the root and read the directory. All these new subdirectories have goofed up its neatness. You know what to do to clean it up. Realphabetize it now, and as often hereafter as you wish.

Telecommunication Software

Most telecommunication software programs install easily and do not impose special requirements on the rest of the system. I use a simple subdirectory named TCOM and place it just off the root. You know how to do that by now, so go ahead and create one.

I often use a communication package named *Smartcom*, but you go ahead and copy in your own communication software. Make no change in the AUTOEXEC.BAT file at this time.

Screen Protection

In order to protect the screen from "burning" when subjected to an unchanging image for long periods while the user answers the phone or goes to lunch, screen protection programs are available. I sometimes use an old one called ***Phosphor Friend***. The question is, where on the hard drive to put special utility programs, such as this one, which consist of only one file? It really doesn't matter where they go, as long as the path tells where to find them.

It would be too easy to store a single file like this in with DOS, or with NU or PC, but that innocent act changes to trouble when it comes time to upgrade DOS, or NU, or PC. Don't do it! Create a special subdirectory which contains only MISC utility programs and add to the path accordingly.

Programs such as *Phosphor Friend* require a *trigger* in the AUTOEXEC file to turn it on. For this particular software, the trigger is **pf 3** which means, turn on **P**hosphor **F**riend, and if no key is struck for **3** minutes, shut off the screen. To turn the screen back on, just tap a SHIFT key.

Add this command to the AUTOEXEC.BAT file and update the **path** command:

```
pf 3
```

The AUTOEXEC file should now look like this:

```
prompt=$p$g
path=c:\;..\;c:\dos;c:\dos\nu;c:\dos\pc;c:\dos\misc;c:\basic
pf 3
```

Reboot the computer.

Program Interference

A screen protection program such as *Phosphor Friend* is *RAM resident*. Unfortunately, as is the case with many other specialized software programs, it occupies memory space that is also used by other programs, or works at cross purposes to it. This *can* cause some strange and unsettling problems. Our mission is to solve problems, not create them, so let's see what it takes to solve this one.

The last 2 programs we installed, *Phosphor Friend* and *Smartcom* interfere with each other. While running the communication program, the screen goes dark at the most inopportune times. *Phosphor Friend* must be turned off *prior* to using *Smartcom*, then turned on again when the communication program is finished. This can be done automatically, which leads us to the next topic and next chapter.

Chapter 40 Summary

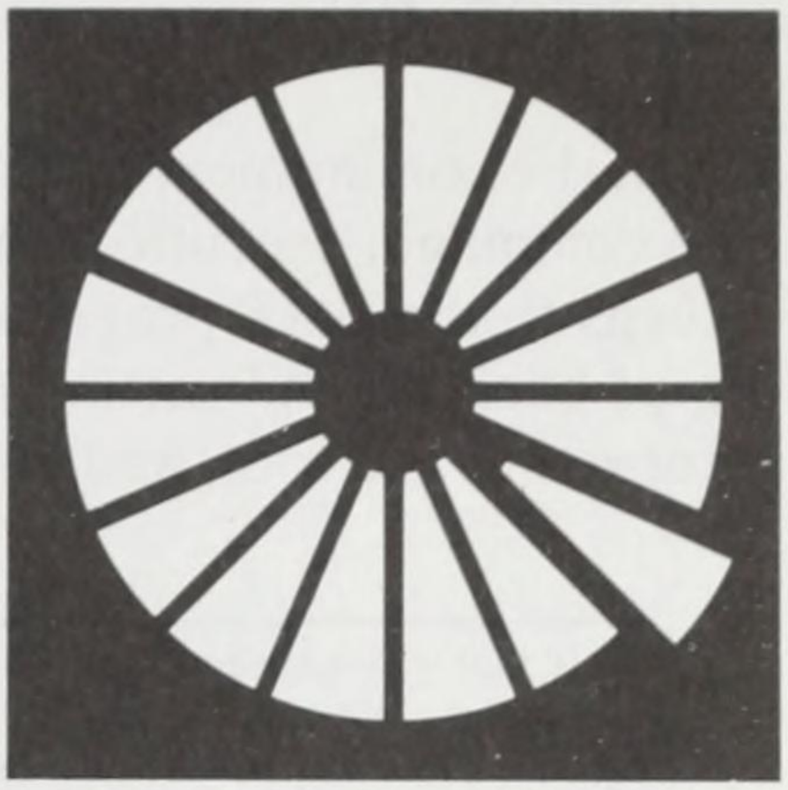
Important software programs that came with MS-DOS can be moved from the DOS subdirectory to their own subdirectories. Create a BASIC subdirectory, use **copy c:\dos\basic?.*** ENTER to copy the BASIC files to it, delete the copied files from the DOS subdirectory, and add this new subdirectory to AUTOEXEC's **path** command.

Word processing software should be placed in its own subdirectory. Sub-subdirectories, such as BIZ, LEGAL, PERS, should be created to hold documents created by the word processor.

Inserting **..\;** between **c:\;** and **c:\dos** in the **path** command in AUTOEXEC.BAT instructs the computer to always check the *parent* directory of a subdirectory whenever a file is not found in the subdirectory.

CAUTION: Some specialized software programs that are *ram resident* occupy memory space and can cause problems when used at the same time as other programs.

CHAPTER 41



Hard Drive Batch Files

Avoiding Conflicts

As we learned earlier, a batch file simply combines a sequence of instructions which are executed by a single command. In the case of our problem with conflicting communication and screen protect programs, the solution is simple. At the root, call up **edlin** and create a new batch file named TCOM.BAT:

edlin tcom.bat ENTER
i ENTER
pf off
cd tcom
scom
pf 3
cd
CTRL C ENTER

e ENTER

Leave this BAT file in the root directory for now. When you wish to call up the communication software from the root, using its **scom** command, simply type:

tcom ENTER

Because of the **path** command in AUTOEXEC.BAT, TCOM.BAT first finds *Phosphor Friend* in the MISC directory and turns it off. Then it moves to the TCOM directory and activates the communication program.

When the communicating is done and we exit the SCOM program using its own command, execution does not end in the TCOM directory. Instead, it returns to the BAT program, which proceeds to the next command, turning **pf** back on and setting the time delay to 3 minutes. Control is then returned to the root, and we're back where we started.

> Note that in this case **pf** was twice found in the MISC directory via the previously set **path** command. We could have returned to the root before searching for **pf**. It made no difference this time, but the next example shows how to find a file which is not in the search path.

Downloading Fonts

Downloading our laser fonts requires several steps, which can be easily combined into one simple BAT file. Use **edlin** to create a file named LASER.BAT at the root where most BAT files are stored:

```
cd lf
lasfonts lf.prs
cd\
```

> Nothing says the BAT files cannot all be placed in a special BAT subdirectory with **bat** in the AUTOEXEC.BAT search path, but I leave such creeping elegance to your own efforts.

Typing **laser** ENTER at the root causes execution to change to the LF directory and downloads a laser font file named LF.PRS to the printer via a special command named LASFONTS, which is found in the LF directory. When the downloading is complete, control is returned to the root directory. It works real slick.

Since special fonts are often (but not always) downloaded in preparation for word processing, it's easy to expand this BAT file so it continues, loading the word processor software, then even moving to an area set aside for a specific employee. For example, go ahead and type:

copy laser.bat irving.bat ENTER

then, using **edlin**, add to the end of IRVING.BAT:

```
cd word
cd irving
word
cd\
```

When Irving, who knows nothing about computers and cares less, comes to work in the morning, he has only to turn on the printer and computer, and when the `C:\>` prompt appears, type his name and press the ENTER key. Within several minutes his fonts are downloaded, the word processor is loaded, and his work station is ready for him to start work.

The IRVING.BAT file can be copied to any other BAT name, edited slightly, and be made ready for use by another worker. For example:

copy irving.bat gertie.bat ENTER

Then edit the **cd irving** line to read **cd gertie** and she's ready to do the same thing.

Since the printer fonts only need to be loaded once a day, each user could have 2 BAT files, a longer, slower one, that includes font downloading, and a quick one that skips the font loading and moves from the root to a specific user's work area. The possibilities of using simple batch files to simplify the work environment for users who are not computer oriented are almost endless.

Automatic BACKUP File

The importance of making backup copies of files is given much lip service, but far less actual attention. Part of the reason is the apparent difficulty of doing it. Many users think they have to back up the entire hard drive every day, and that takes lots of floppy diskettes and lots of time. That is silly because you do have the original software diskettes and can reinstall the actual program files anytime after a failure. Why keep copying them over and over as part of a backup?

In fact, the only backups that need to be made daily are backups of the files which were *changed* since the *last backup*. That type of backup involves the **xcopy** command.

Create and store at the root this simple batch file named SAVEIT.BAT:

xcopy *.* a:/s/d:5/20/91

where **5/20/91** is today's date.

At the end of the day, put a formatted diskette in Drive A, and type:

saveit ENTER

The **xcopy** command copies files with any prefix and extension, from both the root and all subdirectories, *created* or *changed on* or *after* 5/20/1991, from where you are (Drive C) to Drive A.

It even goes so far as to set up matching subdirectories on the diskette. This is heavy duty stuff! Unless you have had an exceptionally busy day, it is very fast and all the changed files will probably fit on a single diskette. Set aside 5 diskettes named MON through FRI and rotate them through the week. You will never be more than one day behind, with a fat safety margin.

Use **edlin** to update the date each day if you want to. Weekly changes to the date should be enough if there aren't a lot of daily changes that could overfill the disk. Your own situation and common sense are the best guides.

Begin this **xcopy** backup system by first performing a *complete* hard drive backup, using the **backup** command and a handful of diskettes. This is absolutely necessary, as hard disks *do* fail. You are performing backups *both* as protection against short term idiot mist*e*akes, like accidental file erasure, and against catastrophic hard drive failure. Thereafter, perform a daily **saveit** as the last event before turning off the computer. Then, once every month or two, perform another complete backup.

Automatic Head Parking BAT File

Hard drives are relatively fragile. The drive itself may be built like a battle ship, but without getting excessively technical, the read and write heads which float only molecules above the high-speed rotating disk platters are not supposed to touch them. If the computer is jostled or bumped, they might touch and, in the process, may scratch a platter. If data is stored at the locations that are struck, it may not be retrievable.

This is not an imaginary problem. I have 2 portable computers with damaged hard drives and am now older and wiser. After we finish with this chapter, use either *Norton* or *PC Tools* to **map** your hard drive to see if it shows any damage.

Many very modern (and more expensive) hard drives have a feature called automatic head retraction. As the name implies, the read/write heads retract and go to a "parking zone" where they can't do any damage if jostled. In one type, the heads automatically retract when the drive is turned off. In another more elegant type, besides retracting when the computer is turned off, they automatically retract while the computer is on if the drive is not accessed for about 5 seconds. This latter type is found in the more expensive portable computers. Unless you can prove otherwise to your own satisfaction, do *not* assume that your hard drive has either of these features.

Now comes the good news. Virtually all hard drives respond to a command that activates a special utility program that retracts the heads. Unfortunately, this command varies widely among hard drive manufacturers. Typical commands are:

shiptrak
headpark
headback
hardrive
wddrive

and others which may or may not make sense. I can never remember which command goes with which computer, so solve the problem the simple way by using a batch file. Since each custom batch file stays on the drive for which it is created, I use my own universal command, which is simply **bye**.

A BAT file named BYE calls up the correct word to park that computer's hard drive heads. The last command given from the root before turning off the computer is:

bye

Create your own one-line head parking batch file which contains your specific head parking command. It will be found in the documentation that came with your computer or the hard drive. The actual head retracting utility program may be on the DOS diskette, which means it was already

copied into the DOS directory and is in the path. If so, it should be moved to the MISC directory.

If not, it is on an extra diskette which accompanied the drive. That program may have been automatically copied to your root directory when the hard drive was formatted, or you may have to copy it yourself into the MISC directory.

Damage Control

If your utility program indicates that your hard drive shows *damaged sectors*, try using the utility to reorganize the drive. In the process, it will *lock out* the damaged sectors leaving the drive still very useable. In fact, most new hard drives are delivered with some manufacturing defects, but they are locked out.

In the unlikely event bad sectors caused by the damage is severe, there is no alternative but to start at the beginning of this section and completely reformat the drive. Formatting automatically *locks out* the damaged sectors. Then reload all your software per the last few chapters.

Do you see why it really is important to back up your hard drive?

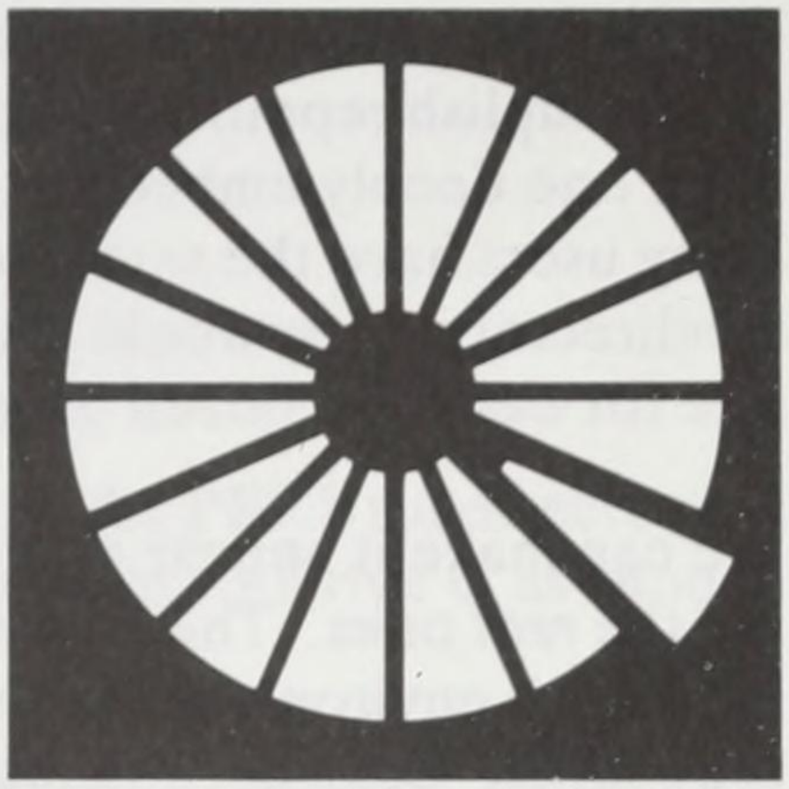
Chapter 41 Summary

Batch files can be created to handle a variety of tasks automatically.

Hard drives can be damaged in moving if touched by the read and write heads, even if they're turned off. Some computers have features to automatically prevent head contact; most others have a command to activate a utility program that retracts the heads.

CHAPTER 42

The CONFIG.SYS File



The CONFIG.SYS file has already been discussed. It takes on added importance with a hard drive since typically many different user programs are installed. The advantage of the CONFIG file is that it can hold many device drivers, including those that were not even thought of when DOS was designed. As you add more user programs do not fear the addition of special device drivers. Just add them into the CONFIG file with **edlin** as needed.

Not all are used at the same time, and many are turned ON and OFF either by a matching command file, which can be executed from the AUTOEXEC.BAT file, or manually when needed. Use of a mouse is a good example.

When installing a mouse, a MOUSE.SYS file is placed in the CONFIG.SYS file, and a MOUSE.COM program placed in the AUTOEXEC.BAT file. When the computer is booted, the mouse is automatically activated.

But there are times you don't want to use the mouse. It can be turned off manually with a different command, or via a command included in a batch file, that resets the number of buffers, turns off the screen protect program, and performs other specific system configurations.

More Disk Drives?

By this time you must be tired of repeating endless sequences of keystrokes to accomplish repetitive tasks. A good example of this repetition is moving from one deeply embedded subdirectory to another, as in the case when many users have the same word processing program, but each has his own subdirectory. How would you like to have over a dozen different *disk drives*, one for each of a dozen users or uses? It's easy and costs nothing.

We can make it appear that the computer has up to 26 disk drives, counting the *real* ones. The advantage is considerably simplified operation in a hard drive environment for users who don't want to type endless command sequences. Consider this simple example:

We want to add an *apparent* or *substitute* disk drive named P to hold the word processor and a subdirectory under it named PERS, for personal. Up to now, to go from the root to PERS, we had to type:

cd word\pers ENTER

By adding Drive P, instead of issuing the above command, we only have to change from Drive C to a new drive named P. Follow along with me.

Changing CONFIG.SYS

Insert this line in the CONFIG.SYS file:

lastdrive = p

This increases the number of drive designations to include *all* the letters between A and P, inclusive. (We could have designated 26 drives by making the last drive Z, but each such drive occupies some memory.)

Changing AUTOEXEC.BAT

Now add this line to the AUTOEXEC.BAT file:

subst p: c:\word\pers ENTER

and reboot.

From Drive C, change the active or default drive to P:

p: ENTER

and there it is:

```
P:\>
```

Take the directory and see that you really are in the PERS subdirectory of WORD. Proceed as usual, and when done, return to Drive C as though nothing had changed.

Add as many additional *substitute* drives as makes sense in your situation. (Don't forget to reboot after each change.)

CAVEAT: This ecstasy is not without limitation. It works just fine for *ordinary* applications, but these *drives* do not respond to esoteric maneuvers. The **path** commands do not recognize these drives, so to alphabetize and perform other disk management, it is necessary to move into the *real* subdirectories.

Using Specialized PATH Commands

Although the path is initially set by a **path** command in the AUTOEXEC.BAT file, it can be changed as often as desired without rebooting the computer. This can be an advantage when running certain high powered software that needs to access lots of specialized subdirectories that are of no interest to other programs, or even when accessing different drives to read or write data.

By including such specialized **path** commands in a batch file, the path can be changed when running the software, then returned to the default setting in the AUTOEXEC.BAT file when done. Since these path settings vary widely with user software, we won't give any examples, but contemplate for a moment the advantages of starting with a simple default path such as we currently have, then changing it as needed for other exotic requirements.

A Touch of Elegance

By the creative use of **echo**, **cls**, **dir**, **dir *.** and other DOS commands in the AUTOEXEC.BAT file, you can customize your computer to just about any level of elegance. If your (or someone else's) computer is used for a single purpose, you may want to expand on the ideas given earlier for automatic power up into a specific work area.

One addition, which is very practical, is to include:

chkdsk/f ENTER

in the AUTOEXEC.BAT file. This way the disk is checked and possible corrections are made each time the computer is turned on.

Upgrading DOS and Other Software

Software is continually being updated and upgraded. By following the clean and crisp organization method we have learned in this section, installing new software becomes a simple and painless task.

Imagine how you would install the next version of DOS. Since we have not corrupted the DOS directory with other files, the process is painless. Here is how it's done:

- Move to the DOS directory and delete all the files.
- Copy the new DOS diskette(s) from Drive A into the DOS directory.
- Copy COMMAND.COM from the DOS directory to the root directory, then delete the file from the DOS directory. **cd** to the root directory.
- Change the default drive to A, and type:

  sys c: ENTER

- Remove the diskette from A, and reboot the computer.

Installation of the upgrade is complete.

CAUTION: Minor upgrades from 3.1 to 3.2 to 3.3 etc. are routine. If you are making a major upgrade such as from 2.x to 3.x, the above may not be enough since the hard drive may need to be reformatted first. Read the instructions that come with the upgrade.

The Ideal Root Directory

As we learned much earlier in the book, the ideal root directory will contain only the mandatory files:

COMMAND.COM

plus

AUTOEXEC.BAT

and

CONFIG.SYS

In addition, it will contain the names of the subdirectories which are directly off the root. It is often possible to achieve this goal.

At this time, your root directory should not be very different. As the final step in this section, your assignment is to create a new subdirectory to hold all your batch files, and change the path so they are automatically found when called from the root.

Chapter 42 Summary

The CONFIG.SYS file enables use of special device drivers needed by many software programs.

A computer may have up to 26 *apparent*, or *substitute*, disk drives.

AUTOEXEC.BAT's **path** command does not recognize *substitute* drives.

The **path** command can be changed without rebooting to access specialized subdirectories.

DOS commands, such as **echo**, **cls**, etc. can be used in the AUTOEXEC.BAT file to customize your computer.

Minor upgrades to DOS involve moving to the DOS subdirectory and deleting its files, copying the files from the new DOS diskette from Drive A to the now-empty DOS subdirectory, copying COMMAND.COM to the root directory, and copying the invisible files to the root with **sys c:** then rebooting.

The ideal root directory should contain only the mandatory files (COMMAND.COM, AUTOEXEC.BAT, and CONFIG.SYS) plus the names of all the subdirectories that branch directly off the root.

Appendices

APPENDIX A

Hard Drive Formatting

This is a generic procedure. Use the specific instructions accompanying your hard drive instead, if available.

FORMATTING A HARD DRIVE USING DOS 3.X

(For standard hard drives of 32 or fewer megabytes.)

1. Copy the numbers from the *Media Error Map* on the bottom of the computer. Do this with the power off.
2. Insert the *system* disk in Drive A, boot, and at the `A>` prompt, type **hsect** ENTER.
3. Press C ENTER to select Drive C as the one to format.
4. Press Y ENTER to start formatting.
5. Press N ENTER if there were no media errors and go to step 7.
6. Press Y ENTER if there were errors and follow the instructions on the screen.
7. Press ENTER to accept the default interleave value.
8. At the `A>` prompt, replace the *system* disk with the Supplemental Programs diskette.
9. Type **fdisk** ENTER.
10. Press 1 ENTER to create a DOS partition.
11. Press Y ENTER to use the entire hard disk for DOS.
12. Replace the Supplemental Programs disk in Drive A with the *system* disk, and press the space bar.
13. Enter the date and time.

14. Type **format c: /s/v** ENTER.
15. Press Y ENTER to continue formatting.
16. When prompted, type **HARD_DISK** ENTER to give the drive a name.
17. Type **c:** ENTER to make Drive C the default drive.
18. Remove the *system* disk from Drive A and reboot.

APPENDIX B

Files That Come with MS-DOS

This appendix briefly describes the purpose of each file on the MS-DOS master and Supplemental Program disks which accompany Tandy computers. Different files come with different models, according to their needs. Utility files related to formatting a specific hard drive are not included, as they are constantly changing and come with their own specific documentation.

With the aid of this information, you can decide which files to delete when creating your operating *system* master. Be sure not to alter the original disks so you can start over again, if necessary.

The computers covered in this appendix are the Tandy 1000 SX, 1000 HX, 1000 TX, 1400 LT, 3000 HL, 3000 HD, and 4000. For detailed information on the use of each file, refer to the appropriate tutorial in this book, and the **MS-DOS Reference Manual** which accompanied your computer.

Some Notes on Our Format

Programs are stored in COM or EXE files. To run a program, type the file name (without COM or EXE), plus any required parameters, and press ENTER. Because programs may be stored as COM files in some versions of DOS and EXE in others, their suffix is indicated in this appendix by the wild card, *. Also, for programs such as keyboard drivers that do the same thing but whose names are spelled slightly differently, we have used a ? to represent the differing letter.

System programs, such as COMMAND.COM, are not intended to be run directly. DOS runs them as needed.

Batch files have the BAT extension. To run, type the file name without the BAT, plus any required parameters, then press ENTER.

BASIC programs often have a BAS extension. Type **basic** and the program name, then press ENTER. Or type **basic** ENTER, then **run** followed by the program name enclosed in quotes. Return to DOS by typing **system** ENTER.

ASCII files contain information that can be displayed by the **type** command or edited with **edlin**. BAT and TXT files are usually stored in ASCII format. DOC files may or may not be in ASCII.

Binary files contain information to be used with other programs. Most programs are, themselves, binary files.

Device drivers are files which customize DOS to add compatibility for certain devices and programs. CONFIG.SYS is a special ASCII file that you create to hold commands that tell DOS which device drivers to install when the system is booted. For example, if CONFIG.SYS contains **device = ansi.sys**, the ANSI driver is loaded.

Filters are special programs that route or change the output from other programs. To use a filter, enter the program name, a pipe symbol (|), and the name of the filter. (Not all programs can be filtered.)

The Files

ANSI.SYS	*Device driver.* When installed in the CONFIG.SYS file, can control video display color, reposition the cursor, and redefine keys on the keyboard with standard ANSI escape sequences.
APPEND.*	*Program.* Searches specified directories for files. When used without parameters, displays the current path.

ASSIGN.*	*Program*. Allows temporary reassignment of disk drives. For example, programs that access Drive B can be made to use Drive C instead.
ATTRIB.*	*Program*. Enables/disables a file's *read-only* status. Can also turn on or off a file's *modification bit*, which indicates if the file has been modified since the last backup.
AUTOEXEC.BAT	*Batch file* that you create. Contains commands to be automatically executed when the computer is booted.
AUTOFMT.*	*Program*. Automatically formats some hard drives.
BACKUP.*	*Program*. (For hard drive users). Copies files from the hard drive to floppy diskettes. Additional control information is saved so that **restore** can recombine files which span multiple diskettes and return them to their original subdirectories on the hard drive.
BASIC.*	*Program*. A BASIC language interpreter that allows you to run existing programs or write your own.
BASICA.*	*Program*. Loads BASIC.EXE and slightly modifies it to increase IBM PC compatibility.
CACHE.*	*Program*. Assigns memory to form CACHE buffer, which is similar to a RAM disk, but maintains a copy of its contents on a hard drive at all times.
CHKDSK.*	*Program*. Checks a disk and displays usage and free space statistics. Options allow automatic correction of faulty directory and file allocation tables.
COMMAND.COM	*System program*. Is automatically loaded upon startup and reloaded (if necessary) after user programs are run. Executes batch files and commands entered at the MS-DOS system prompt.
COMP.*	*Program*. Compares the sizes of two files. Example: **comp config.sys config.bak**

CONFIG.SYS	*ASCII file*. If present in the root directory of the *system* disk, tells MS-DOS how to allocate memory and which special device drivers to load when booting.
CPANEL.*	*Program*. Used to set mouse parameters.
DC.*	*Program*. Compresses files to take up less space. **CAUTION:** Do not use without instructions. Can destroy files.
DEBUG.*	*Program*. Allows examination and modification of memory, registers, disk files, and directories. Primarily used by advanced programmers when testing programs.
DISKCOMP.*	*Program*. Compares diskettes and reports differences.
DISKCOPY.*	*Program*. Copies an entire diskette to another formatted diskette. (Some versions format the destination diskette while copying.)
DISKOPT.*	*Program*. Optimizes file storage on a disk.
DISKTYPE.*	*Program*. Reports the capacity and format of a disk.
DRIVER.SYS	*Device driver*. Allows your system to support external drives.
EDLIN.*	*Program*. Creates and edits ASCII files. Useful for writing short programs and batch files, making lists, and simple word processing.
EXE2BIN.*	*Program*. For use by advanced programmers to create a BIN file from an EXE file. The BIN file may then be renamed to a COM file for execution.
FASTOPEN.*	*Program*. Decreases access time needed to retrieve frequently used programs.
FBACKUP.*	*Program*. Similar to **backup**, but faster.
FC.*	*Program*. Compares two files and reports differences.

FDISK.*	*Program*. Used in preparing a hard drive. Creates, changes, deletes, or displays the partitions assigned to MS-DOS and other operating systems.
FIND.*	*Filter*. Searches one or more input files for a string of characters. Outputs the lines where a match is found or (optionally) not found.
FMAT2000.*	*Program*. Allows other Tandy computers to format 5-1/4" diskettes in the Tandy 2000's 720K format.
FORMAT.*	*Program*. Prepares diskettes for use by DOS.
FRESTORE.*	*Program*. Similar to **restore**, but for use with **fbackup**.
GRAFTABL.*	*Program*. Loads into memory character definitions for ASCII characters 128 to 255.
GRAPHICS.*	*Program*. Installs logic for use with a CGP-220 printer (and others) to print a graphics screen.
HDRIVE.SYS	*Device driver*. Placed in CONFIG.SYS, it allows Tandy computers to use non-standard external hard disk drives. Reads the hard drive's parameters and changes those in your system to match.
HFORMAT.*	*Program*. Used with **hsect** to format a hard drive.
HSECT.*	*Program*. Used with **hformat** to format a hard drive.
JOIN.*	*Program*. Allows a name to be used in place of a drive letter.
KEY?FR.*	*Device driver*. Changes keyboard to French format.
KEY?GR.*	*Device driver*. Changes keyboard to German format.
KEY?IT.*	*Device driver*. Changes keyboard to Italian format.
KEY?SP.*	*Device driver*. Changes keyboard to Spanish format.

KEY?UK.*	*Device driver.* Changes keyboard to United Kingdom format.
KEYCNVRT.SYS	*Device driver.* Causes the keyboard to generate IBM compatible scan codes.
KEYTCF.*	*Device driver.* Changes keyboard to Canadian French format.
LABEL.*	*Program.* Assigns, changes, or removes a volume label.
LF.*	*Program.* Required for line feed compatibility with some printers. Use **lf** ENTER, then **mode lfon** or mode lfoff.
LIB.*	*Program.* For advanced programmers to create and manage library files for use by the linker.
LINK.*	*Program.* For advanced programmers to create an EXE file from one or more object files. Object files are created by a compiler or assembler, or may be stored in a library file.
LPDRVR.SYS	*Device driver.* Installs logic to convert control codes sent to the printer. Provides printer compatibility for certain programs that set vertical and horizontal tabs, lines per page, and characters per line.
LPSETUP.*	*Device driver.* Enables the setting and selection of printer options including tabulation, form feed functions, lines per page, etc.
MLFORMAT.*	*Program.* Formats a hard drive for multiple MS-DOS partitions (DOS1 and DOS2).
MLPART.*	*Program.* Creates multiple partitions on a hard disk drive, via MLPART.SYS.
MLPART.SYS	*Device driver.* Allows access to multiple, non-bootable DOS partitions on a hard disk drive.

MODE.*	*Program.* Changes video display, line printer, and communication settings.
MODEVM.*	*Device driver.* Provides international support of 50 Hz power source.
MON386.*	*Program.* Allows for up to 9 programs to be loaded at the same time.
MORE.*	*Filter.* Intercepts output from a program, command, or file and displays it one screenful at a time.
MOUSE.*	*Program.* Used to turn a mouse ON and OFF. MOUSE.SYS must be installed in the CONFIG.SYS file.
MOUSE.SYS	*Device Driver.* Contains the logic for a mouse. Requires MOUSE.* to turn ON and OFF.
PATCH.*	*Program.* Allows small changes to be made to disk files through the substitution of new text for old. Location of bytes to be changed must be known, and the new text must be the same length as the old.
PRINT.*	*Program.* Prints up to 10 ASCII files from disk. You may do other work on the computer while the printing is in progress.
RAMDISK.SYS	*Device driver.* See VDISK.SYS.
RCRYPT.*	*Program.* Scrambles files so they can be used only by those who have the key or password.
README.DOC	*ASCII file.* If present, contains updated instructions and information not included in the factory manuals. Use the **type** command to display the file on the screen or send it to a printer.
RECOVER.*	*Program.* Locks out bad sectors within files and rebuilds damaged directories.

REPLACE.*	*Program.* Replaces a file needing update(s) with the new version of the file. Also adds new files to a directory.
RESTORE.*	*Program.* (For hard drive users). Copies files onto the hard drive from diskettes created by **backup**.
SELECT.*	*Program.* Creates a *system* diskette designed to be used by a specified country.
SETUP.*	*Program.* Sets the internal clock and allows user to select correct hardware configuration options including amount of memory, number of disk drives, and type of monitor being used.
SETUPHX.*	*Program.* Similar to **setup**, but for the 1000 HX.
SHARE.*	*Program.* Used in networking. Sets aside space in memory for DOS to handle file locking.
SHIPTRAK.*	*Program.* "Parks" the hard drive's read-write head over a safety zone on the platters. Use **shiptrak** ENTER before moving the computer.
SMWCLOCK.*	*Driver program.* Activates and sets the Tandy Smart Watch internal clock.
SORT.*	*Filter.* Intercepts output from a program, command, or file and puts it in alphabetical order.
SPEED.*	*Program.* Slows CPU clock speed from 16 MHz to approximately 6 MHz.
SPOOLER.*	*Program.* Controls the spooler. Can turn the spooler OFF and ON, pause it, empty its buffer, and report its status. Requires installation of SPOOLER.SYS in the CONFIG.SYS file.
SPOOLER.SYS	*Device driver.* Allows data sent to the printer to continue printing while the computer processes other data.

SUBST.*	*Program.* Used to divide a drive among several users. Allows a subdirectory to be treated as a drive by changing its name to a drive letter. Avoids the use of long pathnames.
SYS.*	*Program.* Installs the invisible *system* files on a diskette or hard drive.
TEMM.SYS	*Device driver.* Expanded memory management driver for use with expanded memory board and **cache**.
TREE.*	*Program.* Lists all the subdirectories on a disk and shows the *tree structure*. The /**f** option provides a list of subdirectories and the files they contain.
VDISK.SYS	*Device driver.* If specified in CONFIG.SYS, creates a virtual, or simulated, disk in RAM after boot up. Storage size, sector size, and number of directory entries can be specified.
XCOPY.*	*Program.* Copies a file or a directory and, if designated, its subdirectories from one disk to another. The disks need not have the same format.

APPENDIX C

ASCII Chart

Decimal	Hex	Character	
000	00		(null)
001	01	☺	
002	02	☻	
003	03	♥	(Break)
004	04	♦	
005	05	♣	
006	06	♠	
007	07	•	(Beep)
008	08	◘	(Backspace)
009	09	○	(Tab)
010	0A	◙	(Line feed)
011	0B	♂	
012	0C	♀	(Form feed)
013	0D		(Carriage return)
014	0E	♫	
015	0F	☼	
016	10	►	
017	11	◄	
018	12	↕	

Decimal	Hex	Character
019	13	!!
020	14	¶
021	15	§
022	16	▬
023	17	↨
024	18	↑
025	19	↓
026	1A	→ (End of file)
027	1B	← (Escape)
028	1C	∟
029	1D	↔
030	1E	▲
031	1F	▼
032	20	(space)
033	21	!
034	22	“
035	23	#
036	24	$
037	25	%
038	26	&
039	27	’
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	-
046	2E	.
047	2F	/
048	30	0
049	31	1

Decimal	Hex	Character
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O
080	50	P

Decimal	Hex	Character
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	^
095	5F	—
096	60	`
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

Decimal	Hex	Character
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä

Decimal	Hex	Character
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	Pt
159	9F	ƒ
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡

Decimal	Hex	Character
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▒
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠

Decimal	Hex	Character
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	▉
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	ĖA	Ω

Decimal	Hex	Character
235	EB	δ
236	EC	∞
237	ED	∅
238	EE	∈
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	n
253	FD	^{2}
254	FE	■
255	FF	(blank)

Index

-B-

-G-

3rd Edition

VOLUME 2 ADVANCED APPLICATIONS

MS-DOS® ADVANCED APPLICATIONS, the second of this best-selling two-volume series, takes you inside MS-DOS. Because your computer's entire "personality" is determined by its *operating system*, the more you know about MS-DOS, the more the computer's power becomes your own.

Using easy-to-understand language and useful examples, this straightforward step-by-step tutorial shows you how to simplify all your computer work:

- Make AUTOEXEC.BAT and CONFIG.SYS do what **you** want to do
- Build *batch files* to combine commands and simplify life
- Control your display and printer with MODE
- Organize your hard disk to find what you need
- Set PATHs so you won't get lost
- Use *Edlin* and *copy con* to edit files
- "Track" your programs with DEBUG
- Create *tree-structured directories* to organize files

...and much, much more.

You can study programming and learn about "ones and zeroes," but to be a real computer professional, *learn the operating system.* With this book, you will master MS-DOS and gain the power that comes with being in control!

David A. Lien, pioneer computer book author, has 37 major titles in 10 languages, and over 2 million sales to his credit. Internationally acclaimed for his ability to teach complex, technical subjects in a non-threatening, easy-to-learn way, Dr. Lien has written best-selling books for nearly all popular microcomputers.

TANDY® COMPUTER PRODUCTS

25-1507

Printed in U.S.A.