Unabridged

The
MS-DOS®
Encyclopedia
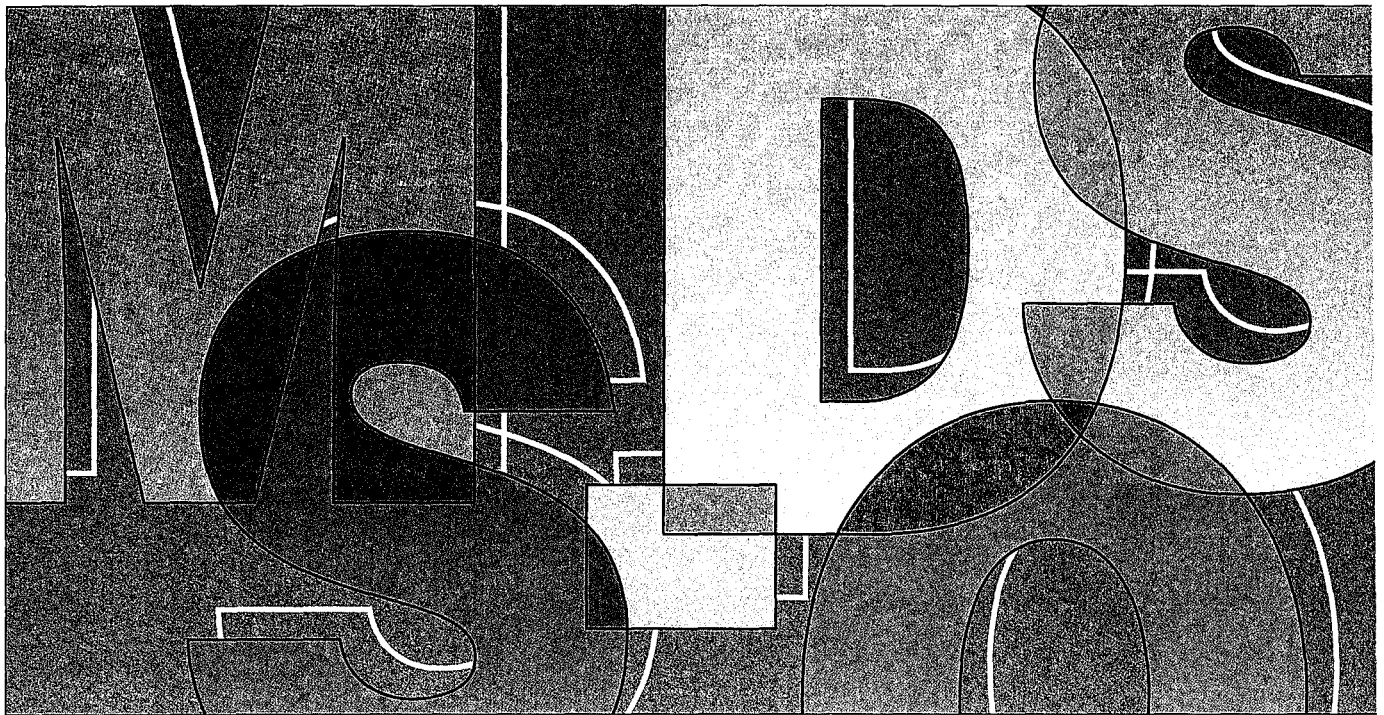
Microsoft
P R E S S

Foreword, Bill Gates
General Editor, Ray Duncan

The

MS-DOS®

Encyclopedia

Encyclopedia Staff

Editor-in-Chief: Susan Lammers

Editorial Director: Patricia Pratt

Senior Editor: Dorothy L. Shattuck

Senior Technical Editor: David L. Rygmyr

Special Projects Editor: Sally A. Brunsman

Editorial Coordinator: Sarah Hersack

Associate Editors and Technical Editors:
Pamela Beason, Ann Becherer, Bob Combs,
Michael Halvorson, Jeff Hinsch, Dean Holmes,
Chris Kinata, Gary Masters, Claudette Moore,
Steve Ross, Roger Shanafelt, Eric Stroo,
Lee Thomas, JoAnne Woodcock

Copy Chief: Brianna Morgan. Proofreaders:
Kathleen Atkins, Julie Carter, Elizabeth
Eisenhood, Matthew Eliot, Patrick Forgette,
Alex Hancock, Richard Isomaki, Shawn Peck,
Alice Copp Smith

Editorial Assistants: Wallis Bolz, Charles Brod,
Stephen Brown, Pat Erickson, Debbie Kem, Susanne
McRhoton, Vihn Nguyen, Cheryl VanGeystel

Index: Shane-Armstrong Information Services

Production: Larry Anderson, Jane Bennett, Rick
Bourgoin, Darcie S. Furlan, Nick Gregoric, Peggy
Herman, Lisa Iversen, Rebecca Johnson, Ruth Pettis,
Russell Steele, Jean Trenary, Joy Ulskey

Marketing and Sales Director: James Brown

Director of Production: Christopher D. Banks

Publisher: Min S. Yee

Contributors

Ray Duncan, General Editor   Duncan received a B.A. in Chemistry from the University of California, Riverside, and an M.D. from the University of California, Los Angeles, and subsequently received specialized training in Pediatrics and Neonatology at the Cedars-Sinai Medical Center in Los Angeles. He has written many articles for personal computing magazines, including *BYTE*, *PC* Magazine, *Dr. Dobb's Journal*, and *Softalk/PC*, and is the author of the Microsoft Press book *Advanced MS-DOS*. He is the founder of Laboratory Microsystems Incorporated, a software house specializing in FORTH interpreters and compilers.

Steve Bostwick   Bostwick holds a B.S. in Physics from the University of California, Los Angeles, and has over 20 years' experience in scientific and commercial data processing. He is president of Query Computing Systems, Inc., a software firm specializing in the creation of systems for applications that interface microcomputers with specialized hardware. He is also an instructor for the UCLA Extension Department of Engineering and Science and helped design their popular Microprocessor Hardware and Software Engineering Certificate Program.

Keith Burgoyne   Born and raised in Orange County, California, Burgoyne began programming in 1974 on IBM 370 mainframes. In 1979, he began developing microcomputer products for Apples, TRS-80s, Ataris, Commodores, and IBM PCs. He is presently Senior Systems Engineer at Local Data of Torrance, California, which is a major producer of IBM 3174/3274 and System 3X protocol conversion products. His previous writing credits include numerous user manuals and tutorials.

Robert A. Byers   Byers is the author of the bestselling *Everyman's Database Primer*. He is presently involved with the Emerald Bay database project with RSPI and Migent, Inc.

Thom Hogan   During 11 years working with personal computers, Hogan has been a software developer, a programmer, a technical writer, a marketing manager, and a lecturer. He has written six books, numerous magazine articles, and four manuals. Hogan is the author of the forthcoming Microsoft Press book *PC Programmer's Sourcebook*.

Jim Kyle   Kyle has 23 years' experience in computing. Since 1967, he has been a systems programmer with strong telecommunications orientation. His interest in microcomputers dates from 1975. He is currently MIS Administrator for BTI Systems, Inc., the OEM Division of BancTec Inc., manufacturers of MICR equipment for the banking industry. He has written 14 books and numerous magazine articles (mostly on ham radio and hobby electronics) and has been primary Forum Administrator for *Computer Language* Magazine's CLMFORUM on CompuServe since early 1985.

Gordon Letwin   Letwin is Chief Architect, Systems Software, Microsoft Corporation. He is the author of *Inside OS/2*, published by Microsoft Press.

Charles Petzold   Petzold holds an M.S. in Mathematics from Stevens Institute of Technology. Before launching his writing career, he worked 10 years in the insurance industry, programming and teaching programming on IBM mainframes and PCs. He is the author of the Microsoft Press book *Programming Windows 2.0*, a contributing editor to *PC* Magazine, and a frequent contributor to the Microsoft *Systems Journal*.

Chip Rabinowitz   Rabinowitz has been a programmer for 11 years. He is presently chief programmer for Productivity Solutions, a microcomputer consulting firm based in Pennsylvania, and has been Forum Administrator for the CompuServe MICROSOFT SIG since 1986.

Jim Tomlin   Tomlin holds a B.S. and an M.S. in Mathematics. He has programmed at Boeing, Microsoft, and Opcon and has taught at Seattle Pacific University. He now heads his own company in Seattle, which specializes in PC systems programming and industrial machine vision applications.

Richard Wilton   Wilton has programmed extensively in PL/1, FORTRAN, FORTH, C, and several assembly languages. He is the author of *Programmer's Guide to PC & PS/2 Video Systems*, published by Microsoft Press.

Van Wolverton   A professional writer since 1963, Wolverton has had bylines as a newspaper reporter, editorial writer, political columnist, and technical writer. He is the author of *Running MS-DOS* and *Supercharging MS-DOS*, both published by Microsoft Press.

William Wong   Wong holds engineering and computer science degrees from Georgia Tech and Rutgers University. He is director of PC Labs and president of Logic Fusion, Inc. His interests include operating systems, computer languages, and artificial intelligence. He has written numerous magazine articles and a book on MS-DOS.

JoAnne Woodcock   Woodcock, a former senior editor at Microsoft Press, has been a writer for *Encyclopaedia Britannica* and a freelance and project editor on marine biological studies at the University of Southern California. She is co-editor (with Michael Halvorson) of *XENIX at Work* and co-author (with Peter Rinearson) of *Microsoft Word Style Sheets*, both published by Microsoft Press.

Special Technical Advisor

Mark Zbikowski

Technical Advisors

Paul Allen	Michael Geary	David Melin	John Pollock
Steve Ballmer	Bob Griffin	Charles Mergentime	Aaron Reynolds
Reuben Borman	Doug Hogarth	Randy Nevin	Darryl Rubin
Rob Bowman	James W. Johnson	Dan Newell	Ralph Ryan
John Butler	Kaamel Kermaani	Tani Newell	Karl Schulmeisters
Chuck Carroll	Adrian King	David Norris	Rajen Shah
Mark Chamberlain	Reed Koch	Mike O'Leary	Barry Shaw
David Chell	James Landowski	Bob O'Rear	Anthony Short
Mike Colee	Chris Larson	Mike Olsson	Ben Slivka
Mike Courtney	Thomas Lennon	Larry Osterman	Jon Smirl
Mike Dryfoos	Dan Lipkie	Ridge Ostling	Betty Stillmaker
Rachel Duncan	Marc McDonald	Sunil Pai	John Stoddard
Kurt Eckhardt	Bruce McKinney	Tim Paterson	Dennis Tillman
Eric Evans	Pascal Martin	Gary Perez	Greg Whitten
Rick Farmer	Estelle Mathers	Chris Peters	Natalie Yount
Bill Gates	Bob Matthews	Charles Petzold	Steve Zeck

Contents

Indexes   1531

Foreword

Microsoft's MS-DOS is the most popular piece of software in the world. It runs on more than 10 million personal computers worldwide and is the foundation for at least 20,000 applications — the largest set of applications in any computer environment. As an industry standard for the family of 8086-based microcomputers, MS-DOS has had a central role in the personal computer revolution and is the most significant and enduring factor in furthering Microsoft's original vision — a computer for every desktop and in every home. The challenge of maintaining a single operating system over the entire range of 8086-based microcomputers and applications is incredible, but Microsoft has been committed to meeting this challenge since the release of MS-DOS in 1981. The true measure of our success in this effort is MS-DOS's continued prominence in the microcomputer industry.

Since MS-DOS's creation, more powerful and much-improved computers have entered the marketplace, yet each new version of MS-DOS reestablishes its position as the foundation for new applications as well as for old. To explain this extraordinary prominence, we must look to the origins of the personal computer industry. The three most significant factors in the creation of MS-DOS were the compatibility revolution, the development of Microsoft BASIC and its widespread acceptance by the personal computer industry, and IBM's decision to build a computer that incorporated 16-bit technology.

The compatibility revolution began with the Intel 8080 microprocessor. This technological breakthrough brought unprecedented opportunities in the emerging microcomputer industry, promising continued improvements in power, speed, and cost of desktop computing. In the minicomputer market, every hardware manufacturer had its own special instruction set and operating system, so software developed for a specific machine was incompatible with the machines of other hardware vendors. This specialization also meant tremendous duplication of effort — each hardware vendor had to write language compilers, databases, and other development tools to fit its particular machine. Microcomputers based on the 8080 microprocessor promised to change all this because different manufacturers would buy the same chip with the same instruction set.

From 1975 to 1981 (the 8-bit era of microcomputing), Microsoft convinced virtually every personal computer manufacturer — Radio Shack, Commodore, Apple, and dozens of others — to build Microsoft BASIC into its machines. For the first time, one common language cut across all hardware vendor lines. The success of our BASIC demonstrated the advantages of compatibility: To their great benefit, users were finally able to move applications from one vendor's machine to another.

Most machines produced during this early period did not have a built-in disk drive. Gradually, however, floppy disks, and later fixed disks, became less expensive and more common, and a number of disk-based programs, including WordStar and dBASE, entered the market. A standard disk operating system that could accommodate these developments became extremely important, leading Lifeboat, Microsoft, and Digital Research all to support CP/M-80, Digital Research's 8080 DOS.

The 8-bit era proved the importance of having a multiple-manufacturer standard that permitted the free interchange of programs. It was important that software designed for the new 16-bit machines have this same advantage. No personal computer manufacturer in 1980 could have predicted with any accuracy how quickly a third-party software industry would grow and get behind a strong standard — a standard that would be the software industry's lifeblood. The intricacies of how MS-DOS became the most common 16-bit operating system, in part through the work we did for IBM, is not the key point here. The key point is that it was inevitable for a popular operating system to emerge for the 16-bit machine, just as Microsoft's BASIC had prevailed on the 8-bit systems.

It was overwhelmingly evident that the personal computer had reached broad acceptance in the market when *Time* in 1982 named the personal computer "Man of the Year." MS-DOS was integral to this acceptance and popularity, and we have continued to adapt MS-DOS to support more powerful computers without sacrificing the compatibility that is essential to keeping it an industry standard. The presence of the 80386 microprocessor guarantees that continued investments in Intel-architecture software will be worthwhile.

Our goal with *The MS-DOS Encyclopedia* is to provide the most thorough and accessible resource available anywhere for MS-DOS programmers. The length of this book is many times greater than the source listing of the first version of MS-DOS — evidence of the growing complexity and sophistication of the operating system. The encyclopedia will be especially useful to software developers faced with preserving continuity yet enhancing the portability of their applications.

Our thriving industry is committed to exploiting the advantages offered by the protected mode introduced with the 80286 microprocessor and the virtual mode introduced with the 80386 microprocessor. MS-DOS will continue to play an integral part in this effort. Faster and more powerful machines running Microsoft OS/2 mean an exciting future of multi-tasking systems, networking, improved levels of data protection, better hardware memory management for multiple applications, stunning graphics systems that can display an innovative graphical user interface, and communication subsystems. MS-DOS version 3, which runs in real mode on 80286-based and 80386-based machines, is a vital link in the Family API of OS/2. Users will continue to benefit from our commitment to improved operating-system performance and usability as the future unfolds.

Bill Gates

Preface

In the space of six years, MS-DOS has become the most widely used computer operating system in the world, running on more than 10 million machines. It has grown, matured, and stabilized into a flexible, easily extendable system that can support networking, graphical user interfaces, nearly any peripheral device, and even CD ROMs containing massive amounts of on-line information. MS-DOS will be with us for many years to come as the platform for applications that run on low-cost, 8086/8088-based machines.

Not surprisingly, the success of MS-DOS has drawn many writers and publishers into its orbit. The number of books on MS-DOS and its commands, languages, and applications dwarfs the list of titles for any other operating system. Why, then, yet another book on MS-DOS? And what can we say about the operating system that has not been said already?

First, we have written and edited *The MS-DOS Encyclopedia* with one audience in mind: the community of working programmers. We have therefore been free to bypass elementary subjects such as the number of bits in a byte and the interpretation of hexadecimal numbers. Instead, we have emphasized detailed technical explanations, working code examples that can be adapted and incorporated into new applications, and a systems view of even the most common MS-DOS commands and utilities.

Second, because we were not subject to size restrictions, we have explored topics in depth that other MS-DOS books mention only briefly, such as exception and error handling, interrupt-driven communications, debugging strategies, memory management, and installable device drivers. We have commissioned definitive articles on the relocatable object modules generated by Microsoft language translators, the operation of the Microsoft Object Linker, and terminate-and-stay-resident utilities. We have even interviewed the key developers of MS-DOS and drawn on their files and bulletin boards to offer an entertaining, illustrated account of the origins of Microsoft's standard-setting operating system.

Finally, by combining the viewpoints and experience of non-Microsoft programmers and writers, the expertise and resources of Microsoft software developers, and the publishing know-how of Microsoft Press, we have assembled a unique and comprehensive reference to MS-DOS services, commands, directives, and utilities. In many instances, the manuscripts have been reviewed by the authors of the Microsoft tools described.

We have made every effort during the creation of this book to ensure that its contents are timely and trustworthy. In a work of this size, however, it is inevitable that errors and omissions will occur. If you discover any such errors, please bring them to our attention so that they can be repaired in future printings and thus aid your fellow programmers. To this end, Microsoft Press has established a bulletin board on MCI Mail for posting corrections and comments. Please refer to page *xvi* for more information.

Ray Duncan

Introduction

The MS-DOS Encyclopedia is the most comprehensive reference work available on Microsoft's industry-standard operating system. Written for experienced microcomputer users and programmers, it contains detailed, version-specific information on all the MS-DOS commands, utilities, and system calls, plus articles by recognized experts in specialized areas of MS-DOS programming. This wealth of material is organized into major topic areas, each with a format suited to its content. Special typographic conventions are also used to clarify the material.

Organization of the Book

The MS-DOS Encyclopedia is organized into five major sections, plus appendixes. Each section has a unique internal organization; explanatory introductions are included where appropriate.

Section I, The Development of MS-DOS, presents the history of Microsoft's standard-setting operating system from its immediate predecessors through version 3.2. Numerous photographs, anecdotes, and quotations are included.

Section II, Programming in the MS-DOS Environment, is divided into five parts: Structure of MS-DOS, Programming for MS-DOS, Customizing MS-DOS, Directions of MS-DOS, and Programming Tools. Each part contains several articles by acknowledged experts on these topics. The articles include numerous figures, tables, and programming examples that provide detail about the subject.

Section III, User Commands, presents all the MS-DOS internal and external commands in alphabetic order, including ANSI.SYS, BATCH, CONFIG.SYS, DRIVER.SYS, EDLIN, RAMDRIVE.SYS, and VDISK.SYS. Each command is presented in a structure that allows the experienced user to quickly review syntax and restrictions on variables; the less-experienced user can refer to the detailed discussion of the command and its uses.

Section IV, Programming Utilities, uses the same format as the User Commands section to present the Microsoft programming aids, including the DEBUG, SYMDEB, and CodeView debuggers. Although some of these utilities are supplied only with Microsoft language products and are not included on the MS-DOS system or supplemental disks, their use is intrinsic to programming for MS-DOS, and they are therefore included to create a comprehensive reference.

Updates to The MS-DOS Encyclopedia

Periodically, the staff of *The MS-DOS Encyclopedia* will publish updates containing clarifications or corrections to the information presented in this current edition. To obtain information about receiving these updates, please check the appropriate box on the business reply card in the back of this book, or send your name and address to: MS-DOS Encyclopedia Update Information, c/o Microsoft Press, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717.

Bulletin Board Service

Microsoft Press is sponsoring a bulletin board on MCI Mail for posting and receiving corrections and comments for *The MS-DOS Encyclopedia*. To use this service, log on to MCI Mail and, after receiving the prompt, type

```
VIEW  <Enter>
```

The *Bulletin Board name:* prompt will be displayed. Then type

```
MSPRESS  <Enter>
```

to connect to the Microsoft Press bulletin board. A list of the individual Microsoft Press bulletin boards will be displayed; simply choose *MSPress DOSENCY* to enter the encyclopedia's bulletin board.

Special Companion Disk Offer

Microsoft Press has created a set of valuable, time saving companion disks to *The MS-DOS Encyclopedia*. They contain the routines and functional programs that are listed throughout this book—thousands of lines of executable code. Conveniently organized, these disks will save you hours of typing time and allow you to start using the code immediately. The companion disks are only available directly from Microsoft Press. To order, use the special bind-in card in the back of the book or send $49.95 for each set of disks, plus sales tax if applicable and $5.50 per disk for domestic postage and handling, $8.00 per disk for foreign orders, to: Microsoft Press, Attn: Companion Disk Offer, 21919 20th Ave. S.E., Box 3011, Bothell, WA 98041-3011. Please specify 5.25-inch or 3.5-inch format. Payment must be in U.S. funds. You may pay by check or money order (payable to Microsoft Press), or by American Express, VISA, or MasterCard; please include your credit card number and expiration date. All domestic orders are shipped 2nd day air upon receipt of order by Microsoft.

CA residents 5% plus local option tax, CT 7.5%, FL 6%, MA 5%, MN 6%, MO 4.225%, NY 4% plus local option tax, WA State 7.8%.

Italic font indicates user-supplied variable names, procedure names in text, parameters whose values are to be supplied by the user, reserved words in the C programming language, messages and return values in text, and, occasionally, emphasis.

A typographic distinction is made between lowercase l and the numeral 1 in both text and program listings.

Cross-references appear in the form SECTION NAME: PART NAME, COMMAND NAME, OR INTERRUPT NUMBER: Article Name or Function Number.

Color indicates user input and program examples.

Terminology

Although not an official IBM name, the term *PC-DOS* in this book means the IBM implementation of MS-DOS. If PC-DOS is referenced and the information differs from that for the related MS-DOS version, the PC-DOS version number is included. To avoid confusion, the term *DOS* is never used without a modifier.

The names of special function keys are spelled as they are shown on the IBM PC keyboard. In particular, the execute key is called Enter, not Return. When *<Enter>* is included in a user-entry line, the user is to press the Enter key at the end of the line.

The common key combinations, such as Ctrl-C and Ctrl-Z, appear in this form when the actual key to be pressed is being discussed but are written as Control-C, Control-Z, and so forth when the resulting code is the true reference. Thus, an article might reference the Control-C handler but state that it is activated when the user presses Ctrl-C.

Unless specifically indicated, hexadecimal numbers are used throughout. These numbers are always followed by the designation *H* (*h* in the code portions of program listings). Ranges of hexadecimal values are indicated with a dash — for example, 07–0AH.

The notation *(more)* appears in italic at the bottom of program listings and tables that are continued on the next page. The complete caption or table title appears on the first page of a continued element and is designated *Continued* on subsequent pages.

Section V, System Calls, documents Interrupts 20H through 27H and Interrupt 2FH. The Interrupt 21H functions are listed in individual entries. This section, like the User Commands and Programming Utilities sections, presents a quick review of usage for the experienced user and also provides extensive notes for the less-experienced programmer.

The 15 appendixes provide quick-reference materials, including a summary of MS-DOS version 3.3, the segmented (new) .EXE file header format, an object file dump utility, and the Intel hexadecimal object file format. Much of this material is organized into tables or bulleted lists for ease of use.

The book includes two indexes — one organized by subject and one organized by command name or system-call number. The subject index provides comprehensive references to the indexed topic; the command index references only the major entry for the command or system call.

Program Listings

The MS-DOS Encyclopedia contains numerous program listings in assembly language, C, and QuickBASIC, all designed to run on the IBM PC family and compatibles. Most of these programs are complete utilities; some are routines that can be incorporated into functioning programs. Vertical ellipses are often used to indicate where additional code would be supplied by the user to create a more functional program. All program listings are heavily commented and are essentially self-documenting.

The programs were tested using the Microsoft Macro Assembler (MASM) version 4.0, the Microsoft C Compiler version 4.0, or the Microsoft QuickBASIC Compiler version 2.0.

The functional programs and larger routines are also available on disk. Instructions for ordering are on the page preceding this introduction and on the mail-in card bound into this volume.

Typography and Terminology

Because *The MS-DOS Encyclopedia* was designed for an advanced audience, the reader generally will be familiar with the notation and typographic conventions used in this volume. However, for ease of use, a few special conventions should be noted.

Typographic conventions

Capital letters are used for MS-DOS internal and external commands in text and syntax lines. Capital letters are also used for filenames in text.

Section I
The Development of MS-DOS
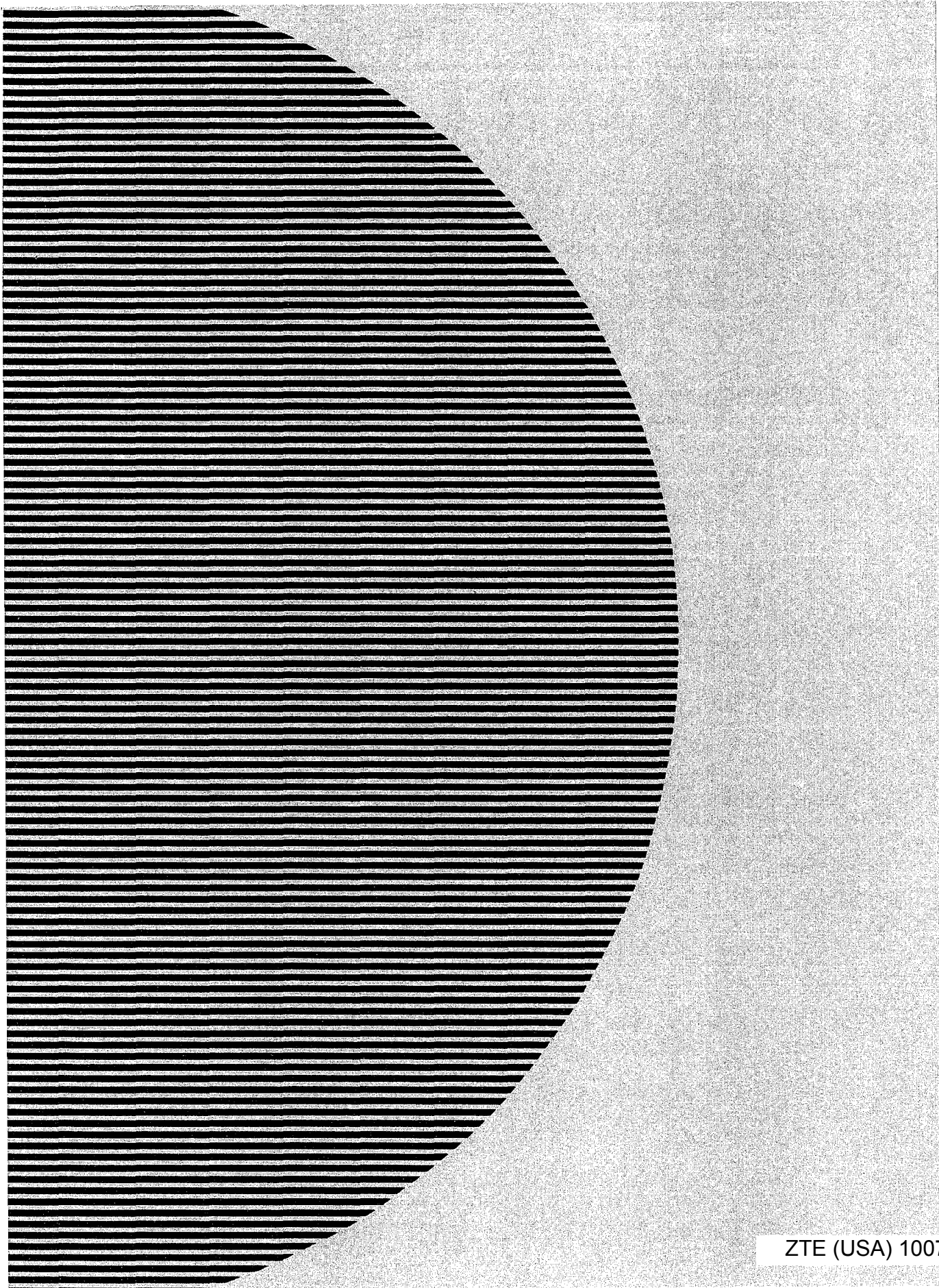
The Development of MS-DOS

To many people who use personal computers, MS-DOS is the key that unlocks the power of the machine. It is their most visible connection to the hardware hidden inside the cabinet, and it is through MS-DOS that they can run applications and manage disks and disk files.

In the sense that it opens the door to doing work with a personal computer, MS-DOS is indeed a key, and the lock it fits is the Intel 8086 family of microprocessors. MS-DOS and the chips it works with are, in fact, closely connected — so closely that the story of MS-DOS is really part of a larger history that encompasses not only an operating system but also a microprocessor and, in retrospect, part of the explosive growth of personal computing itself.

Chronologically, the history of MS-DOS can be divided into three parts. First came the formation of Microsoft and the events preceding Microsoft's decision to develop an operating system. Then came the creation of the first version of MS-DOS. Finally, there is the continuing evolution of MS-DOS since its release in 1981.

Much of the story is based on technical developments, but dates and facts alone do not provide an adequate look at the past. Many people have been involved in creating MS-DOS and directing the lines along which it continues to grow. To the extent that personal opinions and memories are appropriate, they are included here to provide a fuller picture of the origin and development of MS-DOS.

Before MS-DOS

The role of International Business Machines Corporation in Microsoft's decision to create MS-DOS has been well publicized. But events, like inventions, always build on prior accomplishments, and in this respect the roots of MS-DOS reach farther back, to four hardware and software developments of the 1970s: Microsoft's disk-based and stand-alone versions of BASIC, Digital Research's CP/M-80 operating system, the emergence of the 8086 chip, and a disk operating system for the 8086 developed by Tim Paterson at a hardware company called Seattle Computer Products.

Microsoft and BASIC

On the surface, BASIC and MS-DOS might seem to have little in common, but in terms of file management, MS-DOS is a direct descendant of a Microsoft version of BASIC called Stand-alone Disk BASIC.

Before Microsoft even became a company, its founders, Paul Allen and Bill Gates, developed a version of BASIC for a revolutionary small computer named the Altair, which was introduced in January 1975 by Micro Instrumentation Telemetry Systems (MITS) of

The January 1975 cover of Popular Electronics *magazine, featuring the machine that caught the imaginations of thousands of like-minded electronics enthusiasts — among them, Paul Allen and Bill Gates.*

Although it was too limited to serve as the central processor for a general-purpose computer, the 8008 was undeniably the ancestor of the 8080 as far as its architecture and instruction set were concerned. Thus Traf-O-Data's work with the 8008 gave Gates and Allen a head start when they later developed their version of BASIC for the Altair.

Paul Allen learned of the Altair from the cover story in the January 1975 issue of *Popular Electronics* magazine. Allen, then an employee of Honeywell in Boston, convinced Gates, a student at Harvard University, to develop a BASIC for the new computer. The two wrote their version of BASIC for the 8080 in six weeks, and Allen flew to New Mexico to demonstrate the language for MITS. The developers gave themselves the company name of Microsoft and licensed their BASIC to MITS as Microsoft's first product.

Though not a direct forerunner of MS-DOS, Altair BASIC, like the machine for which it was developed, was a landmark product in the history of personal computing. On another level, Altair BASIC was also the first link in a chain that led, somewhat circuitously, to Tim Paterson and the disk operating system he developed for Seattle Computer Products for the 8086 chip.

On the left, Bill Gates's original handwritten notes describing memory configuration for Altair BASIC. On the right, a short bootstrap program written by Gates for Altair users; published in the July 1975 edition of the MITS user newsletter, Computer Notes.

From paper tape to disk

Gates and Allen's early BASIC for the Altair was loaded from paper tape after the bootstrap to load the tape was entered into memory by flipping switches on the front panel of the computer. In late 1975, however, MITS decided to release a floppy-disk system for the Altair — the first retail floppy-disk system on the market. As a result, in February 1976 Allen, by then Director of Software for MITS, asked Gates to write a disk-based version of Altair BASIC. The Altair had no operating system and hence no method of managing files, so the disk BASIC would have to include some file-management routines. It would, in effect, have to function as a rudimentary operating system.

The Altair. Christened one evening shortly before its appearance on the cover of Popular Electronics *magazine, the computer was named for the night's destination of the starship* Enterprise. *The photograph clearly shows the input switches on the front panel of the cabinet.*

Albuquerque, New Mexico. Though it has long been eclipsed by other, more powerful makes and models, the Altair was the first "personal" computer to appear in an environment dominated by minicomputers and mainframes. It was, simply, a metal box with a panel of switches and lights for input and output, a power supply, a motherboard with 18 slots, and two boards. One board was the central processing unit, with the 8-bit Intel 8080 microprocessor at its heart; the other board provided 256 bytes of random-access memory. This miniature computer had no keyboard, no monitor, and no device for permanent storage, but it did possess one great advantage: a price tag of $397.

Now, given the hindsight of a little more than a decade of microcomputing history, it is easy to see that the Altair's combination of small size and affordability was the thin edge of a wedge that, in just a few years, would move everyday computing power away from impersonal monoliths in climate-controlled rooms and onto the desks of millions of people. In 1975, however, the computing environment was still primarily a matter of data processing for specialists rather than personal computing for everyone. Thus when 4 KB
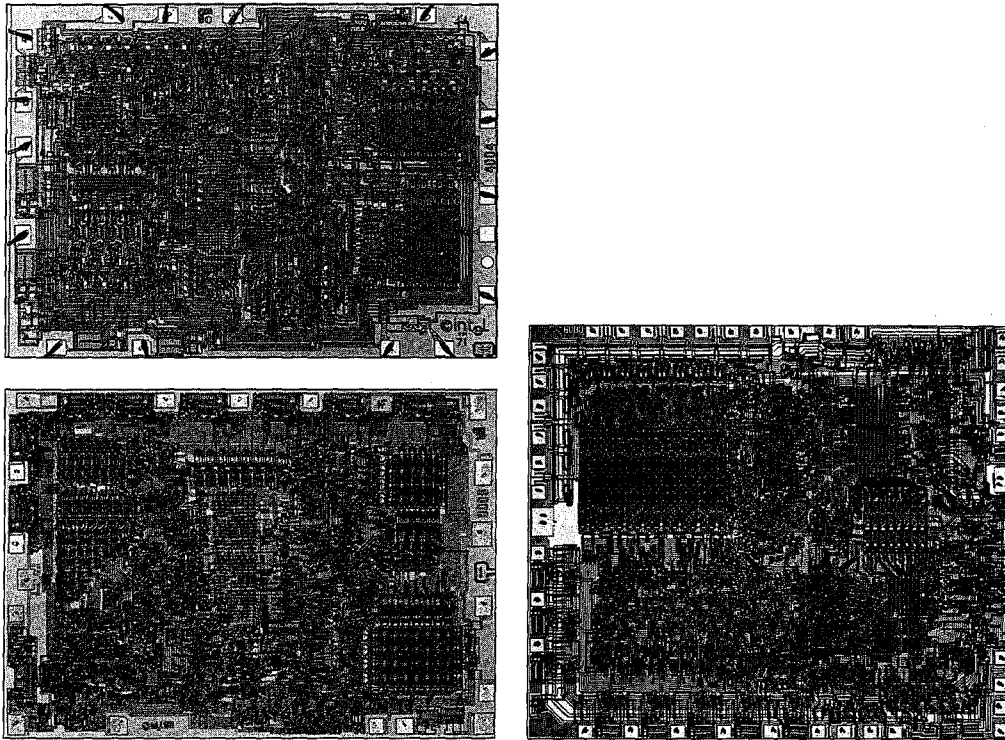
Intel's 4004, 8008, and 8080 chips. At the top left is the 4-bit 4004, which was named for the approximate number of old-fashioned transistors it replaced. At the bottom left is the 8-bit 8008, which addressed 16 KB of memory; this was the chip used in the Traf-O-Data tape-reader built by Paul Gilbert. At the right is the 8080, a faster 8-bit chip that could address 64 KB of memory. The brain of the MITS Altair, the 8080 was, in many respects, the chip on which the personal computing industry was built. The 4004 and 8008 chips were developed early in the 1970s; the 8080 appeared in 1974.

memory expansion boards became available for the Altair, the software needed most by its users was not a word processor or a spreadsheet, but a programming language — and the language first developed for it was a version of BASIC written by Bill Gates and Paul Allen.

Gates and Allen had become friends in their teens, while attending Lakeside School in Seattle. They shared an intense interest in computers, and by the time Gates was in the tenth grade, they and another friend named Paul Gilbert had formed a company called Traf-O-Data to produce a machine that automated the reading of 16-channel, 4-digit, binary-coded decimal (BCD) tapes generated by traffic-monitoring recorders. This machine, built by Gilbert, was based on the Intel 8008 microprocessor, the predecessor of the 8080 in the Altair.

HOW TO "READ" FM TUNER SPECIFICATIONS

Popular Electronics

WORLD'S LARGEST-SELLING ELECTRONICS MAGAZINE   JANUARY 1975/75¢

PROJECT BREAKTHROUGH!

World's First Minicomputer Kit to Rival Commercial Models...

"ALTAIR 8800"   SAVE OVER $1000

ALSO IN THIS ISSUE:

● An Under-$90 Scientific Calculator Project
    ● CCD's—TV Camera Tube Successor?
    ● Thyristor-Controlled Photoflashers

TEST REPORTS:

Technics 200 Speaker System
Pioneer RT-1011 Open-Reel Recorder
Tram Diamond-40 CB AM Transceiver
Edmund Scientific "Kirlian" Photo Kit
Hewlett-Packard 5381 Frequency Counter

The January 1975 cover of Popular Electronics *magazine, featuring the machine that caught the imaginations of thousands of like-minded electronics enthusiasts — among them, Paul Allen and Bill Gates.*

Although it was too limited to serve as the central processor for a general-purpose computer, the 8008 was undeniably the ancestor of the 8080 as far as its architecture and instruction set were concerned. Thus Traf-O-Data's work with the 8008 gave Gates and Allen a head start when they later developed their version of BASIC for the Altair.

Paul Allen learned of the Altair from the cover story in the January 1975 issue of *Popular Electronics* magazine. Allen, then an employee of Honeywell in Boston, convinced Gates, a student at Harvard University, to develop a BASIC for the new computer. The two wrote their version of BASIC for the 8080 in six weeks, and Allen flew to New Mexico to demonstrate the language for MITS. The developers gave themselves the company name of Microsoft and licensed their BASIC to MITS as Microsoft's first product.

Though not a direct forerunner of MS-DOS, Altair BASIC, like the machine for which it was developed, was a landmark product in the history of personal computing. On another level, Altair BASIC was also the first link in a chain that led, somewhat circuitously, to Tim Paterson and the disk operating system he developed for Seattle Computer Products for the 8086 chip.

Storage layout for BASIC

low memory

[TXTTAB]
zero (1 byte)
pointer to next line (2 bytes)
binary line # (2 bytes)
character on line (see note 1)
zero (1 byte)

< Repeat above for each line >
zero (2 bytes)

[VARTAB]
· Simple variables. 6 bytes per variable
  2 bytes give the name
  4 bytes give the value.
  < Repeat for each variable >

[ARYTAB]
Array variables
  2 byte name.
  2 byte length.
  values —

Repeats for each array

[STREND]
lowest location for stack

[STKTOP]
Free space (SP can be in here)
most recent stack entry

[FRETOP]
Stack
bottom of stack / topost location for strings

[FRETOP]
free space
current string usage
STRINGS

[MEMSIZ]
highest machine location.

This scheme allows for simple
table management. Only collector
is for strings which aren't in 4K BASIC.

COMPUTER NOTES/JULY, 1975

Loading Software

Software from MITS will be provided in a checksummed format. There will be a bootstrap loader that you key in manually (less than 25 bytes). This will read a checksum loader (the 'bin' loader) which will be about 120 bytes.

For audio cassette loading the bootstrap and checksum loaders will be longer. All of this will be explained in detail in a cover package that will go out with all software.

For loading non-checksummed paper tapes here is a short program:

```
STKLOC:  DW GETNEW
            (2 bytes-#1 low byte of
                       GETNEW address
                      #2 high byte of
                       GETNEW address)

START:   LXI H,0
GETNEW:  LXI SP, STKLOC
         IN <flag-input channel>
         RAL  ;get input ready bit
         RNZ  ;ready?
         IN <data-input channel>
CHGLOC:  CPI <043 = INX B>
         RNZ
         INR A
         STA CHGLOC
         RET
                         (22 bytes)
```
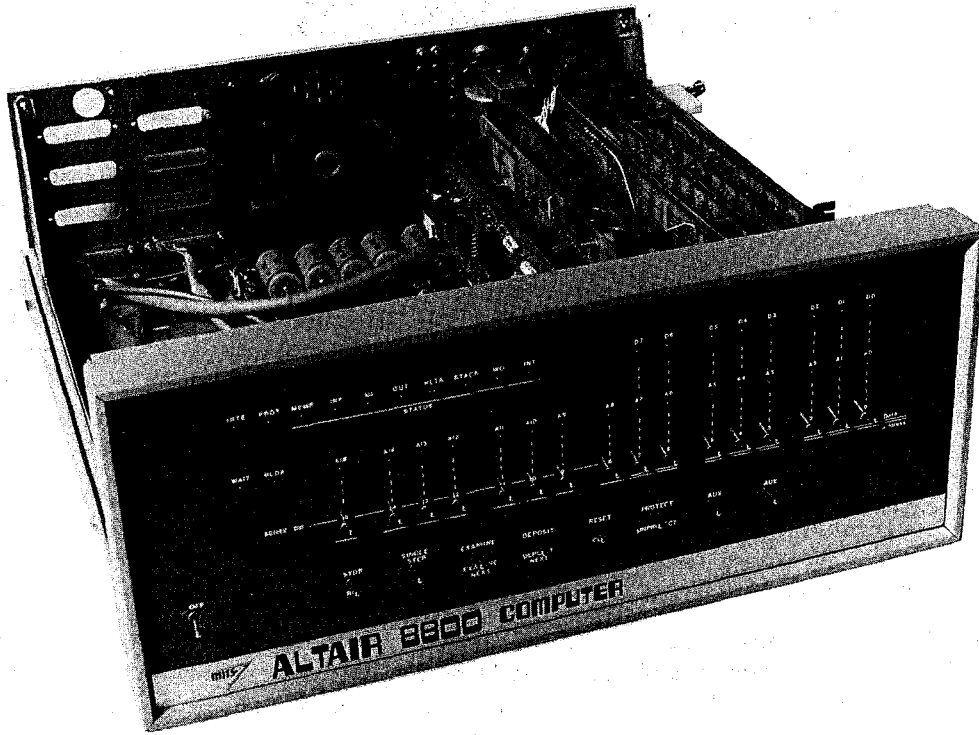
Punch a paper tape with leader, a 043 start byte, the byte to be stored at loc 0, the byte to be stored at 1, - - - etc. Start at START, making sure the memory the loader is in is unprotected. Make sure you don't wipe out the loader by loading on top of it.

To run this again change CHGLOC back to CPI - 376.

On the left, Bill Gates's original handwritten notes describing memory configuration for Altair BASIC. On the right, a short bootstrap program written by Gates for Altair users; published in the July 1975 edition of the MITS user newsletter, Computer Notes.

From paper tape to disk

Gates and Allen's early BASIC for the Altair was loaded from paper tape after the bootstrap to load the tape was entered into memory by flipping switches on the front panel of the computer. In late 1975, however, MITS decided to release a floppy-disk system for the Altair — the first retail floppy-disk system on the market. As a result, in February 1976 Allen, by then Director of Software for MITS, asked Gates to write a disk-based version of Altair BASIC. The Altair had no operating system and hence no method of managing files, so the disk BASIC would have to include some file-management routines. It would, in effect, have to function as a rudimentary operating system.

*Microsoft, 1978, Albuquerque,
New Mexico. Top row, left to right:
Steve Wood, Bob Wallace, Jim Lane.
Middle row, left to right: Bob O'Rear,
Bob Greenberg, Marc McDonald,
Gordon Letwin. Bottom row, left to
right: Bill Gates, Andrea Lewis,
Marla Wood, Paul Allen.*

Gates, still at Harvard University, agreed to write this version of BASIC for MITS. He went to Albuquerque and, as has often been recounted, checked into the Hilton Hotel with a stack of yellow legal pads. Five days later he emerged, yellow pads filled with the code for the new version of BASIC. Arriving at MITS with the code and a request to be left alone, Gates began typing and debugging and, after another five days, had Disk BASIC running on the Altair.

This disk-based BASIC marked Microsoft's entry into the business of languages for personal computers — not only for the MITS Altair, but also for such companies as Data Terminals Corporation and General Electric. Along the way, Microsoft BASIC took on added features, such as enhanced mathematics capabilities, and, more to the point in terms of MS-DOS, evolved into Stand-alone Disk BASIC, produced for NCR in 1977.

Designed and coded by Marc McDonald, Stand-alone Disk BASIC included a file-management scheme called the FAT, or file allocation table that used a linked list for managing disk files. The FAT, born during one of a series of discussions between McDonald and Bill Gates, enabled disk-allocation information to be kept in one location, with "chained" references pointing to the actual storage locations on disk. Fast and flexible, this file-management strategy was later used in a stand-alone version of BASIC for the 8086 chip and eventually, through an operating system named M-DOS, became the basis for the file-handling routines in MS-DOS.

M-DOS

During 1977 and 1978, Microsoft adapted both BASIC and Microsoft FORTRAN for an increasingly popular 8-bit operating system called CP/M. At the end of 1978, Gates and Allen moved Microsoft from Albuquerque to Bellevue, Washington. The company continued to concentrate on programming languages, producing versions of BASIC for the 6502 and the TI9900.

A Microsoft advertisement from the January 1979 issue of Byte *magazine mentioning some products and the machines they ran on. In the lower right corner is an announcement of the company's move to Bellevue, Washington.*

During this same period, Marc McDonald also worked on developing an 8-bit operating system called M-DOS (usually pronounced "Midas" or "My DOS"). Although it never became a real part of the Microsoft product line, M-DOS was a true multitasking operating system modeled after the DEC TOPS-10 operating system. M-DOS provided good performance and, with a more flexible FAT than that built into BASIC, had a better file-handling structure than the up-and-coming CP/M operating system. At about 30 KB, however, M-DOS was unfortunately too big for an 8-bit environment and so ended up being relegated to the back room. As Allen describes it, "Trying to do a large, full-blown operating system on the 8080 was a lot of work, and it took a lot of memory. The 8080 addresses only 64 K, so with the success of CP/M, we finally concluded that it was best not to press on with that."

CP/M

In the volatile microcomputer era of 1976 through 1978, both users and developers of personal computers quickly came to recognize the limitations of running applications on top of Microsoft's Stand-alone Disk BASIC or any other language. MITS, for example, scheduled

a July 1976 release date for an independent operating system for its machine that used the code from the Altair's Disk BASIC. In the same year, Digital Research, headed by Gary Kildall, released its Control Program/Monitor, or CP/M.

CP/M was a typical microcomputer software product of the 1970s in that it was written by one person, not a group, in response to a specific need that had not yet been filled. One of the most interesting aspects of CP/M's history is that the software was developed several years before its release date — actually, several years before the hardware on which it would be a standard became commercially available.

In 1973, Kildall, a professor of computer science at the Naval Postgraduate School in Monterey, California, was working with an 8080-based small computer given him by Intel Corporation in return for some programming he had done for the company. Kildall's machine, equipped with a monitor and paper-tape reader, was certainly advanced for the time, but Kildall became convinced that magnetic-disk storage would make the machine even more efficient than it was.

Trading some programming for a disk drive from Shugart, Kildall first attempted to build a drive controller on his own. Lacking the necessary engineering ability, he contacted a friend, John Torode, who agreed to handle the hardware aspects of interfacing the computer and the disk drive while Kildall worked on the software portion — the refinement of an operating system he had written earlier that year. The result was CP/M.

The version of CP/M developed by Kildall in 1973 underwent several refinements. Kildall enhanced the CP/M debugger and assembler, added a BASIC interpreter, and did some work on an editor, eventually developing the product that, from about 1977 until the appearance of the IBM Personal Computer, set the standard for 8-bit microcomputer operating systems.

Digital Research's CP/M included a command interpreter called CCP (Console Command Processor), which acted as the interface between the user and the operating system itself, and an operations handler called BDOS (Basic Disk Operating System), which was responsible for file storage, directory maintenance, and other such housekeeping chores. For actual input and output — disk I/O, screen display, print requests, and so on — CP/M included a BIOS (Basic Input/Output System) tailored to the requirements of the hardware on which the operating system ran.

For file storage, CP/M used a system of eight-sector allocation units. For any given file, the allocation units were listed in a directory entry that included the filename and a table giving the disk locations of 16 allocation units. If a long file required more than 16 allocation units, CP/M created additional directory entries as required. Small files could be accessed rapidly under this system, but large files with more than a single directory entry could require numerous relatively time-consuming disk reads to find needed information.

At the time, however, CP/M was highly regarded and gained the support of a broad base of hardware and software developers alike. Quite powerful for its size (about 4KB), it was, in all respects, the undisputed standard in the 8-bit world, and remained so until, and even after, the appearance of the 8086.
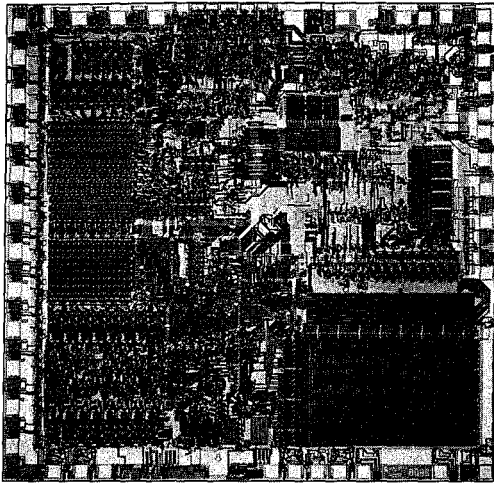
The 16-bit Intel 8086 chip, introduced in 1978. Much faster and far more powerful than its 8-bit predecessor the 8080, the 8086 had the ability to address one megabyte of memory.

The 8086

When Intel released the 8-bit 8080 chip in 1974, the Altair was still a year in the future. The 8080 was designed not to make computing a part of everyday life but to make household appliances and industrial machines more intelligent. By 1978, when Intel introduced the 16-bit 8086, the microcomputer was a reality and the new chip represented a major step ahead in performance and memory capacity. The 8086's full 16-bit buses made it faster than the 8080, and its ability to address one megabyte of random-access memory was a giant step beyond the 8080's 64 KB limit. Although the 8086 was not compatible with the 8080, it was architecturally similar to its predecessor and 8080 source code could be mechanically translated to run on it. This translation capability, in fact, was a major influence on the design of Tim Paterson's operating system for the 8086 and, through Paterson's work, on the first released version of MS-DOS.

When the 8086 arrived on the scene, Microsoft, like other developers, was confronted with two choices: continue working in the familiar 8-bit world or turn to the broader horizons offered by the new 16-bit technology. For a time, Microsoft did both. Acting on Paul Allen's suggestion, the company developed the SoftCard for the popular Apple II, which was based on the 8-bit 6502 microprocessor. The SoftCard included a Z80 microprocessor and a copy of CP/M-80 licensed from Digital Research. With the SoftCard, Apple II users could run any program or language designed to run on a CP/M machine.

It was 16-bit technology, however, that held the most interest for Gates and Allen, who believed that this would soon become the standard for microcomputers. Their optimism was not universal — more than one voice in the trade press warned that industry investment in 8-bit equipment and software was too great to successfully introduce a new standard. Microsoft, however, disregarded these forecasts and entered the 16-bit arena as it had with the Altair: by developing a stand-alone version of BASIC for the 8086.

At the same time and, coincidentally, a few miles south in Tukwila, Washington, a major contribution to MS-DOS was taking place. Tim Paterson, working at Seattle Computer Products, a company that built memory boards, was developing an 8086 CPU card for use in an S-100 bus machine.

86-DOS

Paterson was introduced to the 8086 chip at a seminar held by Intel in June 1978. He had attended the seminar at the suggestion of his employer, Rod Brock of Seattle Computer Products. The new chip sparked his interest because, as he recalls, "all its instructions worked on both 8 and 16 bits, and you didn't have to do everything through the accumulator. It was also real fast — it could do a 16-bit ADD in three clocks."

After the seminar, Paterson — again with Brock's support — began work with the 8086. He finished the design of his first 8086 CPU board in January 1979 and by late spring had developed a working CPU, as well as an assembler and an 8086 monitor. In June, Paterson took his system to Microsoft to try it with Stand-alone BASIC, and soon after, Microsoft BASIC was running on Seattle Computer's new board.

During this period, Paterson also received a call from Digital Research asking whether they could borrow the new board for developing CP/M-86. Though Seattle Computer did not have a board to loan, Paterson asked when CP/M-86 would be ready. Digital's representative said December 1979, which meant, according to Paterson's diary, "we'll have to live with Stand-alone BASIC for a few months after we start shipping the CPU, but then we'll be able to switch to a real operating system."

Early in June, Microsoft and Tim Paterson attended the National Computer Conference in New York. Microsoft had been invited to share Lifeboat Associates' ten-by-ten foot booth, and Paterson had been invited by Paul Allen to show BASIC running on an S-100 8086 system. At that meeting, Paterson was introduced to Microsoft's M-DOS, which he found interesting because it used a system for keeping track of disk files — the FAT developed for Stand-alone BASIC — that was different from anything he had encountered.

After this meeting, Paterson continued working on the 8086 board, and by the end of the year, Seattle Computer Products began shipping the CPU with a BASIC option.

When CP/M-86 had still not become available by April 1980, Seattle Computer Products decided to develop a 16-bit operating system of its own. Originally, three operating systems were planned: a single-user system, a multiuser version, and a small interim product soon informally christened QDOS (for Quick and Dirty Operating System) by Paterson.

Both Paterson (working on QDOS) and Rod Brock knew that a standard operating system for the 8086 was mandatory if users were to be assured of a wide range of application software and languages. CP/M had become the standard for 8-bit machines, so the ability to mechanically translate existing CP/M applications to run on a 16-bit system became one of Paterson's major goals for the new operating system. To achieve this compatibility, the system he developed mimicked CP/M-80's functions and command structure, including its use of file control blocks (FCBs) and its approach to executable files.

An advertisement for the Seattle Computer Products 8086 CPU, with 86-DOS; published in the December 1980 issue of Byte.

At the same time, however, Paterson was dissatisfied with certain elements of CP/M, one of them being its file-allocation system, which he considered inefficient in the use of disk space and too slow in operation. So for fast, efficient file handling, he used a file allocation table, as Microsoft had done with Stand-alone Disk BASIC and M-DOS. He also wrote a translator to translate 8080 code to 8086 code, and he then wrote an assembler in Z80 assembly language and used the translator to translate it.

Four months after beginning work, Paterson had a functioning 6 KB operating system, officially renamed 86-DOS, and in September 1980 he contacted Microsoft again, this time to ask the company to write a version of BASIC to run on his system.

IBM

While Paterson was developing 86-DOS, the third major element leading to the creation of MS-DOS was gaining force at the opposite end of the country. IBM, until then seemingly oblivious to most of the developments in the microcomputer world, had turned its attention to the possibility of developing a low-end workstation for a market it knew well: business and business people.

On August 21, 1980, a study group of IBM representatives from Boca Raton, Florida, visited Microsoft. This group, headed by a man named Jack Sams, told Microsoft of IBM's interest in developing a computer based on a microprocessor. IBM was, however, unsure of microcomputing technology and the microcomputing market. Traditionally, IBM relied on long development cycles — typically four or five years — and was aware that such lengthy design periods did not fit the rapidly evolving microcomputer environment.

One of IBM's solutions — the one outlined by Sams's group — was to base the new machine on products from other manufacturers. All the necessary hardware was available, but the same could not be said of the software. Hence the visit to Microsoft with the question: Given the specifications for an 8-bit computer, could Microsoft write a ROM BASIC for it by the following April?

Microsoft responded positively, but added questions of its own: Why introduce an 8-bit computer? Why not release a 16-bit machine based on Intel's 8086 chip instead? At the end of this meeting — the first of many — Sams and his group returned to Boca Raton with a proposal for the development of a low-end, 16-bit business workstation. The venture was named Project Chess.

One month later, Sams returned to Microsoft asking whether Gates and Allen could, still by April 1981, provide not only BASIC but also FORTRAN, Pascal, and COBOL for the new computer. This time the answer was no because, though Microsoft's BASIC had been designed to run as a stand-alone product, it was unique in that respect — the other languages would need an operating system. Gates suggested CP/M-86, which was then still under development at Digital Research, and in fact made the initial contact for IBM. Digital Research and IBM did not come to any agreement, however.

Microsoft, meanwhile, still wanted to write all the languages for IBM — approximately 400 KB of code. But to do this within the allotted six-month schedule, the company needed some assurances about the operating system IBM was going to use. Further, it needed specific information on the internals of the operating system, because the ROM BASIC would interact intimately with the BIOS.

The turning point

That state of indecision, then, was Microsoft's situation on Sunday, September 28, 1980, when Bill Gates, Paul Allen, and Kay Nishi, a Microsoft vice president and president of ASCII Corporation in Japan, sat in Gates's eighth-floor corner office in the Old National Bank Building in Bellevue, Washington. Gates recalls, "Kay and I were just sitting there at night and Paul was on the couch. Kay said, 'Got to do it, got to do it.' It was only 20 more K

of code at most—actually, it turned out to be 12 more K on top of the 400. It wasn't that big a deal, and once Kay said it, it was obvious. We'd always wanted to do a low-end operating system, we had specs for low-end operating systems, and we knew we were going to do one up on 16-bit."

At that point, Gates and Allen began looking again at Microsoft's proposal to IBM. Their estimated 400 KB of code included four languages, an assembler, and a linker. To add an operating system would require only another 20 KB or so, and they already knew of a working model for the 8086: Tim Paterson's 86-DOS. The more Gates, Allen, and Nishi talked that night about developing an operating system for IBM's new computer, the more possible—even preferable—the idea became.

Allen's first step was to contact Rod Brock at Seattle Computer Products to tell him that Microsoft wanted to develop and market SCP's operating system and that the company had an OEM customer for it. Seattle Computer Products, which was not in the business of marketing software, agreed and licensed 86-DOS to Microsoft. Eventually, SCP sold the operating system to Microsoft for $50,000, favorable language licenses, and a license back from Microsoft to use 86-DOS on its own machines.

In October 1980, with 86-DOS in hand, Microsoft submitted another proposal to IBM. This time the plan included both an operating system and the languages for the new computer. Time was short and the boundaries between the languages and the operating system were unclear, so Microsoft explained that it needed to control the development of the operating system in order to guarantee delivery by spring of 1981. In November, IBM signed the contract.

Creating MS-DOS

At Thanksgiving, a prototype of the IBM machine arrived at Microsoft and Bill Gates, Paul Allen, and, primarily, Bob O'Rear began a schedule of long, sometimes hectic days and total immersion in the project. As O'Rear recalls, "If I was awake, I was thinking about the project."

The first task handled by the team was bringing up 86-DOS on the new machine. This was a challenge because the work had to be done in a constantly changing hardware environment while changes were also being made to the specifications of the budding operating system itself.

As part of the process, 86-DOS had to be compiled and integrated with the BIOS, which Microsoft was helping IBM to write, and this task was complicated by the media. Paterson's 86-DOS—not counting utilities such as EDLIN, CHKDSK, and INIT (later named FORMAT)—arrived at Microsoft as one large assembly-language program on an 8-inch floppy disk. The IBM machine, however, used 5¼-inch disks, so Microsoft needed to determine the format of the new disk and then find a way to get the operating system from the old format to the new.
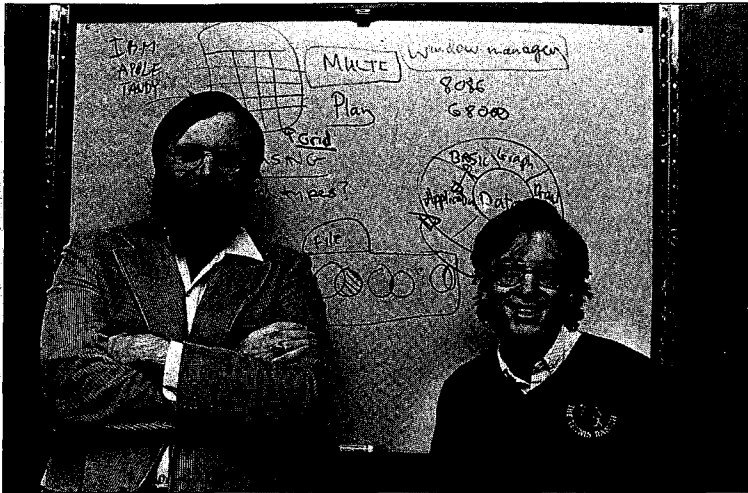
*Paul Allen and
Bill Gates (1982).*

This work, handled by O'Rear, fell into a series of steps. First, he moved a section of code from the 8-inch disk and compiled it. Then, he converted the code to Intel hexadecimal format. Next, he uploaded it to a DEC-2020 and from there downloaded it to a large Intel fixed-disk development system with an In-Circuit Emulator. The DEC-2020 used for this task was also used in developing the BIOS, so there was additional work in downloading the BIOS to the Intel machine, converting it to hexadecimal format, moving it to an IBM development system, and then crossloading it to the IBM prototype.

Defining and implementing the MS-DOS disk format—different from Paterson's 8-inch format—was an added challenge. Paterson's ultimate goal for 86-DOS was logical device independence, but during this first stage of development, the operating system simply had to be converted to handle logical records that were independent of the physical record size.

Paterson, still with Seattle Computer Products, continued to work on 86-DOS and by the end of 1980 had improved its logical device independence by adding functions that streamlined reading and writing multiple sectors and records, as well as records of variable size. In addition to making such refinements of his own, Paterson also worked on dozens of changes requested by Microsoft, from modifications to the operating system's startup messages to changes in EDLIN, the line editor he had written for his own use. Throughout this process, IBM's security restrictions meant that Paterson was never told the name of the OEM and never shown the prototype machines until he left Seattle Computer Products and joined Microsoft in May 1981.

And of course, throughout the process the developers encountered the myriad loose ends, momentary puzzles, bugs, and unforeseen details without which no project is complete. There were, for example, the serial card interrupts that occurred when they should not and, frustratingly, a hardware constraint that the BIOS could not accommodate at first and that resulted in sporadic crashes during early MS-DOS operations.

Bob O'Rear's sketch of the steps involved in moving 86-DOS to the IBM prototype.

Part of Bob O'Rear's "laundry" list of operating-system changes and corrections for early April 1981. Around this time, interim beta copies were shipped to IBM for testing.

"My own IBM computer.
Imagine that."

Presenting the IBM of
Personal Computers.

"Dad, can I use
the IBM computer
tonight?"
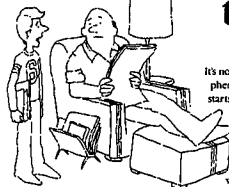
It's not an unusual phenomenon. It starts when your son asks to borrow a tie. Or when your daughter wants to use your metal racquet. Sometimes you let them. Often you don't. But when they start asking to use your IBM Personal Computer, it's better to say yes.

Because learning about computers is a subject your kids can study and enjoy at home.

It's also a fact that the IBM Personal Computer can be as useful in your home as it is in your office. To help plan the family budget, for instance. Or to compute anything from interest paid to calories consumed. You can even tap directly into the Dow Jones data bank with your telephone and an inexpensive adapter.

But as surely as an IBM Personal Computer can help you, it can also help your children. Because just by playing games or drawing

colorful graphics, your son or daughter will discover what makes a computer tick—and what it can do. They can take the same word processing program you use to create business reports to write and edit book reports (and learn how to type in the process.) Your kids might even get so "computer smart," they'll start writing their own programs in BASIC or Pascal.

Ultimately, an IBM Personal Computer can be one of the best investments you make in your family's future. And one of the least expensive. Starting at less than $1600* there's a system that, with the addition of one simple device, hooks up to your home TV and uses your audio cassette recorder.

To introduce your family to the IBM Personal Computer, visit any ComputerLand® store or Sears Business Systems Center. Or see it all at one of our IBM Product Centers. (The IBM National Accounts Division will serve business customers who want to purchase in quantity.)

And remember. When your kids ask to use your IBM Personal Computer, let them. But just make sure you can get it back. After all your son's still wearing that tie.

The IBM Personal Computer   and me.

The 1981 debut of the IBM Personal Computer.

In spite of such difficulties, however, the new operating system ran on the prototype for the first time in February 1981. In the six months that followed, the system was continually refined and expanded, and by the time of its debut in August 1981, MS-DOS, like the IBM Personal Computer on which it appeared, had become a functional product for home and office use.

Version 1

The first release of MS-DOS, version 1.0, was not the operating system Microsoft envisioned as a final model for 16-bit computer systems. According to Bill Gates, "Basically, what we wanted to do was one that was more like MS-DOS 2, with the hierarchical file system and everything...the key thing [in developing version 1.0] was my saying, 'Look, we can come out with a subset first and just go upward from that.'"

This first version — Gates's subset of MS-DOS — was actually a good compromise between the present and the future in two important respects: It enabled Microsoft to meet the development schedule for IBM and it maintained program-translation compatibility with CP/M.

Available only for the IBM Personal Computer, MS-DOS 1.0 consisted of 4000 lines of assembly-language source code and ran in 8 KB of memory. In addition to utilities such as DEBUG, EDLIN, and FORMAT, it was organized into three major files. One file, IBMBIO.COM, interfaced with the ROM BIOS for the IBM PC and contained the disk and character input/output system. A second file, IBMDOS.COM, contained the DOS kernel, including the application-program interface and the disk-file and memory managers. The third file, COMMAND.COM, was the external command processor — the part of MS-DOS most visible to the user.

To take advantage of the existing base of languages and such popular applications as WordStar and dBASE II, MS-DOS was designed to allow software developers to mechanically translate source code for the 8080 to run on the 8086. And because of this link, MS-DOS looked and acted like CP/M-80, at that time still the standard among operating systems for microcomputers. Like its 8-bit relative, MS-DOS used eight-character filenames and three-character extensions, and it had the same conventions for identifying disk drives in command prompts. For the most part, MS-DOS also used the same command language, offered the same file services, and had the same general structure as CP/M. The resemblance was even more striking at the programming level, with an almost one-to-one correspondence between CP/M and MS-DOS in the system calls available to application programs.

New Features

MS-DOS was not, however, a CP/M twin, nor had Microsoft designed it to be inextricably bonded to the IBM PC. Hoping to create a product that would be successful over the long term, Microsoft had taken steps to make MS-DOS flexible enough to accommodate changes and new directions in the hardware technology — disks, memory boards, even microprocessors — on which it depended. The first steps toward this independence from

BUSINESS Digest

Big I.B.M.'s Little Computer

Its Desk-Top Model Brings A New Image

Retail Sales In U.S. Up 1.3% in July

But Analysts Are Dubious of General Upturn

The U.S. Desktop Computer Market

UNIT SHIPMENTS | VALUE OF SHIPMENTS

TANDY 175.4
APPLE 102.1
COMMODORE 61
ATARI 13.3
HEWLETT-PACKARD 139.1
NORTHSTAR 27.4
TI INSTRUMENTS
IBM 97.0
VERTEC DATA 30.8
XIDY SYSTEMS 18.4
VEKTRONIX 44.4
All Others 337.0

IBM's New Line Likely to Shake Up The Market for Personal Computers

By GEORGE ANDERS
Staff Reporter of THE WALL STREET JOURNAL

NEW YORK—International Business Machines Corp. has made its bold entry into the personal-computer market, and experts believe the computer giant could capture the lead in the youthful industry within two years.

Yesterday the company introduced several versions of a small computer designed for use in homes, schools and offices. Prices...

catch-up. The IBM machines operate on an Intel Corp. 8088 microprocessor, a faster and more powerful "chip" than those used in rivals' machines. IBM also has obtained for distribution such popular programs as VisiCalc, a financial forecasting model marketed by Personal Software Inc.

Other programs, or software, for the IBM equipment include the EasyWriter word-processing system, three accounting packages from Peachtree Software Inc. and...

far greater, equivalent to more than 1,000 typewritten pages. The new IBM computers don't use all that capacity, but what they do use will enable them to work with longer programs and more data than competing machines and to display images on their video screens in greater detail.

But the added memory comes at a price. IBM acknowledges that a fully stocked computer will cost $6,000 or more. Its basic $1,565 machine comes with 16,000 characters...

another 64,000
another $540,
ble extras runs analyst at Morr new system's s "It's a nice kage," he says. y little that you here already " ggests that IBM rounded" price quipped model Apple III. the

InfoWorld
News For Microcomputer Users

IBM Announces New Microcomputer System
It's Official; One surprise

By Thom Hogan, IW Staff

NEW YORK, NY—Within a month you should be able to order an IBM Personal Computer. Whether or not that will have any effect on the microcomputing industry remains to be seen.

For those of you who have been reading InfoWorld, there were few surprises in the IBM announcement. Although the actual introduction took place a month later than anticipated, the features of the machine are virtually identical to the information we've already published.

tion of the unit taking place at the IBM facility at Boca Raton, Florida, as reported earlier in InfoWorld. Computerland, the newly created Sears Business Centers and IBM Product Centers will be selling the Personal Computer. IBM is also setting up a special division of the Data Processing Group to market the machine.

The price begins at $1565, slightly higher than we reported earlier. It includes the keyboard unit, an enhanced Microsoft BASIC in...

OUTLOOK

IBM really gets personal.
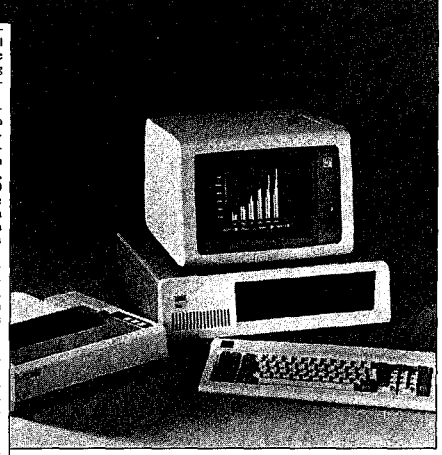
PERSONAL COMPUTERS

PERSONAL COMPUTER FROM IBM

The mainframer's long-awaited entry into the personal computing market aims for corporate as well as home users.

With uncharacteristic but resounding fanfare, IBM ended the summer's most popular guessing game for the industry by introducing its Personal Computer. Highly comparable to offerings from arch-contenders Apple and Radio Shack, the machine represents several new tacks for the leading computer manufacturer as it attempts to hitch its wagon to one of the fastest growing segments of the industry.

The computer, which is designed to appeal to home users as well as corporate professionals, ranges in price from $1,565 for a bare-bones configuration to $6,300 for the full-blown model. It will be sold through

Sears and Computerland computer retail stores as well as directly to large corporate and educational users, IBM says, pointing out that it has set up a special national marketing team to handle such volume orders.

Donald Estridge, the articulate director of IBM's entry systems business who braved strobes and movie lights at the machine's Waldorf-Astoria introduction, declines to say how many personnel have been dedicated to the national marketing effort, but says it will be selling in volumes of 20 machines or more. Several weeks after the unveiling, he said response so far had been "very, very good," with orders being taken but no deliveries to be made before this month.

In addition to the game of Adventure, which Estridge said has been thoroughly exercised by his Boca Raton, Fla., staff, IBM has decked out the machine with an array of packaged applications programs that are expected to make it attractive to the corporate user.

Among these are the popular VisiCalc spreadsheet package from Personal Software, accounting packages from Management Science America's Peachtree Software, and Information Unlimited's EasyWriter word processing system. Although IBM wouldn't say, more independently developed packages are certain to be offered for the computer as well as packages

ently unveiled its first offering in the personal computer market—the IBM Personal Computer. The unit, perhaps surprisingly, plays music and includes game software to say nothing of the standard features available.

The machine is impressive. Its starting price is a mere $1565. For that price the buyer gets the 83-key keyboard, the computer itself, based on an 8088 microprocessor, and 16k of main memory. This minimal configuration can use a tape cassette for mass storage and a television set (with an rf

modulator) for a display. (The machine is fully FCC certified for home operation as a class B computing device.)

IBM is cognizant of the fact that this minimally configured machine probably won't last a serious computerist long before he wants to expand. The company offers upgraded versions of the machine, and will sell them in different configurations. For example, the firm lists a more typical configuration for home or school as 64k of main memory, one disk

A sampling of the headlines and newspaper articles that abounded when IBM announced its Personal Computer.

MICROSOFT
QUARTERLY

This policy is especially advantageous when a large number of programs is distributed using a single copy of the runtime module because only one royalty payment is paid.

(Microsoft still supports the runtime system used with previous versions. If application programmers link the old library to their applications, there is no royalty fee. This applies to versions 5.2 and earlier, too.)

This change in the BASCOM royalty policy reflects Microsoft's wish to increase the number of application packages on the market. This policy change, the addition of CHAIN with COMMON, and the implementation of the runtime module make BASCOM a much more flexible and powerful tool for the application programmer.

BASCOM 5.3 is available now for CP/M systems, including the Apple II with the Microsoft Softcard. Microsoft is committed to supporting BASCOM and the BASIC interpreter on many processors and operating systems, thus assuring that application programs created with BASCOM have, and will continue to have, the broadest possible market.

Paul Allen

IBM Breaks the 16-Bit Barrier

The most important feature of the new IBM Personal Computer is its 8088 CPU. IBM's choice of the 8088 CPU opens up two areas of the industry that have been

Paul Allen, Vice President, Microsoft

on the verge of changing for the past 10 months: first, the industry's hesitancy over a serious 16-bit software commitment has finally been broken; and second, the capabilities of the 16-bit processors are finally being put to some really exciting uses.

A 16-bit processor gives software designers many advantages inherent in an enhanced instruction set. For example, we've taken advantage of the expanded addressing in our MS-LINK, a linker for Pascal or FORTRAN programs that are up to a megabyte in size. In 96K of memory, the Microsoft 8086 BASIC interpreter can execute a 64K program, almost double the size executable on an 8-bit runtime. Applications programs can be more sophisticated in their features, human engineering factors, and in solving problems that involve larger amounts of data.

The larger number of registers with the 8086/8088 processors also means that com-

plex operations, such as floating point and graphics routines, execute much faster. The speed of the graphics primitives in MBASIC-86 makes it very easy to construct a graphics application without machine language.

With the IBM announcement of the Personal Computer, it looks as though the industry is finally gearing up for serious 16-bit software support. In addition to the Microsoft software already provided for the IBM Personal Computer, we're planning a full line of 16-bit languages and end-user software tools. Application packages are rapidly being adapted to the 16-bit environment, especially those programs already written in Microsoft BASIC.

The "linch pin" of Microsoft's new 16-bit product line for the 8086/8088 is our compact, flexible operating system, MS-DOS. MS-DOS is the primary operating system on the IBM Personal Computer. We've maintained compatibility with existing CP/M 2.x operating system calls, so it's a straightforward process to convert 8080 and Z80 programs to run under MS-DOS. MS-DOS also provides a future upgrade path to the XENIX multi-user, multitasking environment. Other important features of MS-DOS include error recovery, device independent I/O, and built-in variable length disk reads and writes. What is now the standard operating system for the IBM Personal Computer will no doubt become an industry standard.

Now that the 16-bit software barrier has been crossed and the technical capabilities of the 16-bit processors are being appreciated, Microsoft expects to see many 16-bit personal computers. It's an industry move we've anticipated for quite some time and, given the momentum of IBM, it should soon be in full swing.

Microsoft COBOL Passes GSA Validation

Microsoft is always concerned about standards for all its products. The United States government, the largest user of computer equipment and software in the world, has developed tests for compliance with and implementation of standards for compilers. Testing of compilers, called validation, is performed by government inspectors, who are independent of software developers.

Microsoft submitted its COBOL compiler (under the CP/M operating system) for validation. The General Services Administration (GSA) performed the validation tests and validated Microsoft COBOL as a low-intermediate implementation of the 1974 ANSI standard for COBOL.

Why is Microsoft concerned about standards, and why did we submit Microsoft COBOL for validation? Mike Orr, COBOL product manager, offered the following reasons:
(continued on back)

A page from Microsoft's third-quarter report for 1981.

specific hardware configurations appeared in MS-DOS version 1.0 in the form of device-independent input and output, variable record lengths, relocatable program files, and a replaceable command processor.

MS-DOS made input and output device-independent by treating peripheral devices as if they were files. To do this, it assigned a reserved filename to each of the three devices it recognized: CON for the console (keyboard and display), PRN for the printer, and AUX for the auxiliary serial ports. Whenever one of these reserved names appeared in the file control block of a file named in a command, all operations were directed to the device, rather than to a disk file. (A file control block, or FCB, is a 37-byte housekeeping record located in an application's portion of the memory space. It includes, among other things, the filename, the extension, and information about the size and starting location of the file on disk.)

Such device independence benefited both application developers and computer users. On the development side, it meant that applications could use one set of read and write calls, rather than a number of different calls for different devices, and it meant that an application did not have to be modified if new devices were added to the system. From the

user's point of view, device independence meant greater flexibility. For example, even if a program had been designed for disk I/O only, the user could still use a file for input or direct output to the printer.

Variable record lengths provided another step toward logical independence. In CP/M, logical and physical record lengths were identical: 128 bytes. Files could be accessed only in units of 128 bytes and file sizes were always maintained in multiples of 128 bytes. With MS-DOS, however, physical sector sizes were of no concern to the user. The operating system maintained file lengths to the exact size in bytes and could be relied on to support logical records of any size desired.

Another new feature in MS-DOS was the relocatable program file. Unlike CP/M, MS-DOS had the ability to load two different types of program files, identified by the extensions .COM and .EXE. Program files ending with .COM mimicked the binary files in CP/M. They were more compact than .EXE files and loaded somewhat faster, but the combined program code, stack, and data could be no larger than 64 KB. A .EXE program, on the other hand, could be much larger because the file could contain multiple segments, each of which could be up to 64KB. Once the segments were in memory, MS-DOS then used part of the file header, the relocation table, to automatically set the correct addresses for each segment reference.

In addition to supporting .EXE files, MS-DOS made the external command processor, COMMAND.COM, more adaptable by making it a separate relocatable file just like any other program. It could therefore be replaced by a custom command processor, as long as the new file was also named COMMAND.COM.

Performance

Everyone familiar with the IBM PC knows that MS-DOS eventually became the dominant operating system on 8086-based microcomputers. There were several reasons for this, not least of which was acceptance of MS-DOS as the operating system for IBM's phenomenally successful line of personal computers. But even though MS-DOS was the only operating system available when the first IBM PCs were shipped, positioning alone would not necessarily have guaranteed its ability to outstrip CP/M-86, which appeared six months later. MS-DOS also offered significant advantages to the user in a number of areas, including the allocation and management of storage space on disk.

Like CP/M, MS-DOS shared out disk space in allocation units. Unlike CP/M, however, MS-DOS mapped the use of these allocation units in a central file allocation table — the FAT — that was always in memory. Both operating systems used a directory entry for recording information about each file, but whereas a CP/M directory entry included an allocation map — a list of sixteen 1 KB allocation units where successive parts of the file were stored — an MS-DOS directory entry pointed only to the first allocation unit in the FAT and each entry in the table then pointed to the next unit associated with the file. Thus, CP/M might require several directory entries (and more than one disk access) to load a file

larger than 16 KB, but MS-DOS retained a complete in-memory list of all file components and all available disk space without having to access the disk at all. As a result, MS-DOS's ability to find and load even very long files was extremely rapid compared with CP/M's.

Two other important features — the ability to read and write multiple records with one operating-system call and the transient use of memory by the MS-DOS command processor — provided further efficiency for both users and developers.

The independence of the logical record from the physical sector laid the foundation for the ability to read and write multiple sectors. When reading multiple records in CP/M, an application had to issue a read function call for each sector, one at a time. With MS-DOS, the application could issue one read function call, giving the operating system the beginning record and the number of records to read, and MS-DOS would then load all of the corresponding sectors automatically.

Another innovative feature of MS-DOS version 1.0 was the division of the command processor, COMMAND.COM, into a resident portion and a transient portion. (There is also a third part, an initialization portion, which carries out the commands in an AUTOEXEC batch file at startup. This part of COMMAND.COM is discarded from memory when its work is finished.) The reason for creating resident and transient portions of the command processor had to do with maximizing the efficiency of MS-DOS for the user: On the one hand, the programmers wanted COMMAND.COM to include commonly requested functions, such as DIR and COPY, for speed and ease of use; on the other hand, adding these commands meant increasing the size of the command processor, with a resulting decrease in the memory available to application programs. The solution to this trade-off of speed versus utility was to include the extra functions in a transient portion of COMMAND.COM that could be overwritten by any application requiring more memory. To maintain the integrity of the functions for the user, the resident part of COMMAND.COM was given the job of checking the transient portion for damage when an application terminated. If necessary, this resident portion would then load a new copy of its transient partner into memory.

Ease of Use

In addition to its moves toward hardware independence and efficiency, MS-DOS included several services and utilities designed to make life easier for users and application developers. Among these services were improved error handling, automatic logging of disks, date and time stamping of files, and batch processing.

MS-DOS and the IBM PC were targeted at a nontechnical group of users, and from the beginning IBM had stressed the importance of data integrity. Because data is most likely to be lost when a user responds incorrectly to an error message, an effort was made to include concise yet unambiguous messages in MS-DOS. To further reduce the risks of misinterpretation, Microsoft used these messages consistently across all MS-DOS functions and utilities and encouraged developers to use the same messages, where appropriate, in their applications.

```
   ○    Package Contents

                1 diskette, with the following files:
                     COMMAND.COM
                     MSDOS.COM
                     EDLIN.COM
                     DEBUG.COM
                     FILCOM.COM
                1 MS-DOS Disk Operating System Manual



        System Requirements

        The MS-DOS Operating System requires 8K bytes of memory.

   ○


   ○
```

```
   ○    Contents

        Introduction

                Features and Benefits of MS-DOS
                Using This Manual
                Syntax Notation
                MS-DOS Structure and Characteristics

        Chapter  1      General MS-DOS Commands
                 1.1    Control Function Characters
                 1.2    Special Editing Commands
                 1.3    Disk Errors

        Chapter  2      COMMAND.COM

                 2.1    Prompt
                 2.2    Filenames
                 2.3    Commands
                 2.3.1      Internal Commands
                 2.3.2      External Commands

   ○    Chapter  3      EDLIN

                 3.1    Invoking EDLIN
                 3.2    Commands
                 3.2.1      Command Parameters
                 3.2.2      Interline Commands
                 3.3    Error Messages

        Chapter  4      DEBUG

                 4.1    Invoking DEBUG
                 4.2    Commands
                 4.2.1      Command Parameters
                 4.2.2      Command Descriptions
                 4.3    Error Messages

        Chapter  5      FILCOM

                 5.1    Invoking FILCOM
                 5.2    Commands
                 5.2.1      Filenames
                 5.2.2      Switches
                 5.3    Examples

   ○    Chapter  6      Instructions for Single Disk Drive Users
```

Two pages from Microsoft's MS-DOS version 1.0 manual. On the left, the system's requirements — 8 KB of memory; on the right, the 118-page manual's complete table of contents.

In a further attempt to safeguard data, MS-DOS also trapped hard errors — such as critical hardware errors — that had previously been left to the hardware-dependent logic. Now the hardware logic could simply report the nature of the error and the operating system would handle the problem in a consistent and systematic way. MS-DOS could also trap the Control-C break sequence so that an application could either protect against accidental termination by the user or provide a graceful exit when appropriate.

To reduce errors and simplify use of the system, MS-DOS also automatically updated memory information about the disk when it was changed. In CP/M, users had to log new disks as they changed them — a cumbersome procedure on single-disk systems or when data was stored on multiple disks. In MS-DOS, new disks were automatically logged as long as no file was currently open.

Another new feature — one visible with the DIR command — was date and time stamping of disk files. Even in its earliest forms, MS-DOS tracked the system date and displayed it at every startup, and now, when it turned out that only the first 16 bytes of a directory entry

were needed for file-header information, the MS-DOS programmers decided to use some of the remaining 16 bytes to record the date and time of creation or update (and the size of the file) as well.

Batch processing was originally added to MS-DOS to help IBM. IBM wanted to run scripts — sequences of commands or other operations — one after the other to test various functions of the system. To do this, the testers needed an automated method of calling routines sequentially. The result was the batch processor, which later also provided users with the convenience of saving and running MS-DOS commands as batch files.

Finally, MS-DOS increased the options available to a program when it terminated. For example, in less sophisticated operating systems, applications and other programs remained in memory only as long as they were active; when terminated, they were removed from memory. MS-DOS, however, added a terminate-and-stay-resident function that enabled a program to be locked into memory and, in effect, become part of the operating-system environment until the computer system itself was shut down or restarted.

The Marketplace

When IBM announced the Personal Computer, it said that the new machine would run three operating systems: MS-DOS, CP/M-86, and SofTech Microsystem's p-System. Of the three, only MS-DOS was available when the IBM PC shipped. Nevertheless, when MS-DOS was released, nine out of ten programs on the *InfoWorld* bestseller list for 1981 ran under CP/M-80, and CP/M-86, which became available about six months later, was the operating system of choice to most writers and reviewers in the trade press.

Understandably, MS-DOS was compared with CP/M-80 and, later, CP/M-86. The main concern was compatibility: To what extent was Microsoft's new operating system compatible with the existing standard? No one could have foreseen that MS-DOS would not only catch up with but supersede CP/M. Even Bill Gates now recalls that "our most optimistic view of the number of machines using MS-DOS wouldn't have matched what really ended up happening."

To begin with, the success of the IBM PC itself surprised many industry watchers. Within a year, IBM was selling 30,000 PCs per month, thanks in large part to a business community that was already comfortable with IBM's name and reputation and, at least in retrospect, was ready for the leap to personal computing. MS-DOS, of course, benefited enormously from the success of the IBM PC — in large part because IBM supplied all its languages and applications in MS-DOS format.

But, at first, writers in the trade press still believed in CP/M and questioned the viability of a new operating system in a world dominated by CP/M-80. Many assumed, incorrectly, that a CP/M-86 machine could run CP/M-80 applications. Even before CP/M-86 was available, *Future Computing* referred to the IBM PC as the "CP/M Record Player" — presumably in anticipation of a vast inventory of CP/M applications for the new computer — and led its readers to assume that the PC was actually a CP/M machine.

Microsoft, meanwhile, held to the belief that the success of IBM's machine or any other 16-bit microcomputer depended ultimately on the emergence of an industry standard for a 16-bit operating system. Software developers could not afford to develop software for even two or three different operating systems, and users could (or would) not pay the prices the developers would have to charge if they did. Furthermore, users would almost certainly rebel against the inconvenience of sharing data stored under different operating-system formats. There had to be one operating system, and Microsoft wanted MS-DOS to be the one.

The company had already taken the first step toward a standard by choosing hardware independent designs wherever possible. Machine independence meant portability, and portability meant that Microsoft could sell one version of MS-DOS to different hardware manufacturers who, in turn, could adapt it to their own equipment. Portability alone, however, was no guarantee of industry-wide acceptance. To make MS-DOS the standard, Microsoft needed to convince software developers to write programs for MS-DOS. And in 1981, these developers were a little confused about IBM's new operating system.

An operating system by any other name...

A tangle of names gave rise to one point of confusion about MS-DOS. Tim Paterson's "Quick and Dirty Operating System" for the 8086 was originally shipped by Seattle Computer Products as 86-DOS. After Microsoft purchased 86-DOS, the name remained for a while, but by the time the PC was ready for release, the new system was known as MS-DOS. Then, after the IBM PC reached the market, IBM began to refer to the operating system as the IBM Personal Computer DOS, which the trade press soon shortened to PC-DOS. IBM's version contained some utilities, such as DISKCOPY and DISKCOMP, that were not included in MS-DOS, the generic version available for license by other manufacturers. By calling attention to these differences, publications added to the confusion about the distinction between the Microsoft and IBM releases of MS-DOS.

Further complications arose when Lifeboat Associates agreed to help promote MS-DOS but decided to call the operating system Software Bus 86. MS-DOS thus became one of a line of trademarked Software Bus products, another of which was a product called SB-80, Lifeboat's version of CP/M-80.

Finally, some of the first hardware companies to license MS-DOS also wanted to use their own names for the operating system. Out of this situation came such additional names as COMPAQ-DOS and Zenith's Z-DOS.

Given this confusing host of names for a product it believed could become the industry standard, Microsoft finally took the lead and, as developer, insisted that the operating system was to be called MS-DOS. Eventually, everyone but IBM complied.

Developers and MS-DOS

Early in its career, MS-DOS represented just a small fraction of Microsoft's business — much larger revenues were generated by BASIC and other languages. In addition, in the first two years after the introduction of the IBM PC, the growth of CP/M-86 and other

environments nearly paralleled that of MS-DOS. So Microsoft found itself in the unenviable position of giving its support to MS-DOS while also selling languages to run on CP/M-86, thereby contributing to the growth of software for MS-DOS's biggest competitor.

Given the uncertain outcome of this two-horse race, some other software developers chose to wait and see which way the hardware manufacturers would jump. For their part, the hardware manufacturers were confronting the issue of compatibility between operating systems. Specifically, they needed to be convinced that MS-DOS was not a maverick — that it could perform as well as CP/M-86 as a base for applications that had been ported from the CP/M-80 environment for use on 16-bit computers.

Microsoft approached the problem by emphasizing four related points in its discussions with hardware manufacturers:

- First, one of Microsoft's goals in developing the first version of MS-DOS had always been translation compatibility from CP/M-80 to MS-DOS software.
- Second, translation was possible only for software written in 8080 or Z80 assembly language; thus, neither MS-DOS nor CP/M-86 could run programs written for other 8-bit processors, such as the 6800 or the 6502.
- Third, many applications were written in a high-level language, rather than in assembly language.
- Fourth, most of those high-level languages were Microsoft products and ran on MS-DOS.

Thus, even though some people had originally believed that only CP/M-86 would automatically make the installed base of CP/M-80 software available to the IBM PC and other 16-bit computers, Microsoft convinced the hardware manufacturers that MS-DOS was, in actuality, as flexible as CP/M-86 in its compatibility with existing — and appropriate — CP/M-80 software.

MS-DOS was put at a disadvantage in one area, however, when Digital Research convinced several manufacturers to include both 8080 and 8086 chips in their machines. With 8-bit and 16-bit software used on the same machine, the user could rely on the same disk format for both types of software. Because MS-DOS used a different disk format, CP/M had the edge in these dual-processor machines — although, in fact, it did not seem to have much effect on the survival of CP/M-86 after the first year or so.

Although making MS-DOS the operating system of obvious preference was not as easy as simply convincing hardware manufacturers to offer it, Microsoft's list of MS-DOS customers grew steadily from the time the operating system was introduced. Many manufacturers continued to offer CP/M-86 along with MS-DOS, but by the end of 1983 the technical superiority of MS-DOS (bolstered by the introduction of such products as Lotus 1-2-3) carried the market. For example, when DEC, a longtime holdout, decided to make MS-DOS the primary operating system for its Rainbow computer, the company mentioned the richer set of commands and "dramatically" better disk performance of MS-DOS as reasons for its choice over CP/M-86.

Additional MS-DOS Features and Benefits

- **Written Entirely In 8086 Assembly Language**

  This provides significant speed improvements over operating systems that are largely translated from their 8-bit counterparts.

- **Fast Efficient File Structure**

  The format eliminates the need for "extents," minimizes access to the directory track, and provides for duplicate directory information and verify after write.

- **No Need to Log In Disks**

  As long as no file is currently open, there is no need to log in a new disk by typing Control-C. This greatly improves usability for single disk system users and for people who like to store their data on separate diskettes.

- **No Physical File/Disk Size Limitation**

  Unlike users of operating systems that are limited to 8 megabytes, MS-DOS users would not have to break a 24 megabyte hard disk into three separate drives.

- **No Overhead for Non-128-Byte Physical Sectors**

  One does not have to worry about different physical sector sizes when writing a BIOS.

- **Time/Date Stamps**

  This alleviates, for instance, the need to recompile a file if the time on the relocatable file is more recent than on the source file.

- **Lifeboat Associates**

  The world's largest independent distributor of microcomputer software has chosen to support MS-DOS as its low-end 16-bit operating system. Recognizing the important migration path from the 8-bit level to XENIX OS, Lifeboat will be offering a wide range of software for the MS-DOS environment.

- **100% IBM Compatible**

  IBM is offering software running under MS-DOS. IBM has announced Microsoft BASIC and Microsoft Pascal, along with accounting, financial planning, and word processing software running under MS-DOS.

MS-DOS

Standard Operating System for 8086 Micros

MS-DOS is a disk operating system from Microsoft for 8086/8088 microprocessors. International Business Machines Corp. chose MS-DOS (called IBM Personal Computer DOS) to be its operating system of choice for its Personal Computer. Microsoft's agreements with IBM and several other major computer manufacturers indicate that end-user systems running MS-DOS will be widely available in the near future, making MS-DOS the standard low-end operating system for 8086 micros. Why is MS-DOS becoming popular? MS-DOS is an important advance in microcomputer operating systems.

What Makes MS-DOS Important?

All of Microsoft's languages (BASIC Interpreter, BASIC Compiler, FORTRAN, COBOL, Pascal) are available immediately under MS-DOS. Users of MS-DOS are assured that their operating system will be the first that Microsoft will support when any new products or major releases are announced. In addition, the 8-bit versions of Microsoft's languages are upward compatible with the 16-bit versions. Thus, application programs written in 8-bit Microsoft languages can be run under MS-DOS with little or no modification. Microsoft wants to encourage both the transporting of 8-bit to 16-bit software, and the development of new 16-bit software.

Here are the major features that make MS-DOS the operating system people want to use on 8086 machines:

- **Easy Conversion from 8080 to 8086**

  MS-DOS allows as much transportability of 8-bit machine language software as is possible. MS-DOS emulates system calls to CP/M-80. By simply running assembly language source code through the Intel conversion program, almost all 8080 programs will work without modification. In most cases, a conversion to MS-DOS is easier than conversion to other operating systems.

- **Device Independent I/O**

  MS-DOS simplifies I/O to different devices on the UNIX concept. A single set of I/O calls treats all devices alike from the user's perspective. There is no need to rewrite programs when a new device is added to the system. Simply OPEN the device and READ or WRITE. Also, device independent I/O assures that different control characters (specifically TAB) are handled the same by the different devices.

- **Advanced Error Recovery Procedures**

  MS-DOS doesn't simply fade away when errors occur. If a disk error occurs at any time during any program, MS-DOS will retry the operation three times. If the operation cannot be completed successfully, MS-DOS will return an error message, then wait for the user to enter a response. The user can attempt recovery rather than reboot the operating system.

- **Complete Program Relocatability**

  MS-DOS is a truly relocatable operating system. Not only can the Microsoft relocatable linking loader provide for separate segments, but also the COMMAND program in MS-DOS relocates the modules during loading rather than loading them to preset addresses. Thus, MS-DOS does not have the 64K program space limitation of other operating systems.

- **Powerful, Flexible File Characteristics**

  MS-DOS has no practical limit on file or disk size. MS-DOS uses 4-byte XENIX OS compatible logical pointers for file and disk capacity up to 4 gigabytes.

  Within a single diskette, the user of MS-DOS can have files of different logical record lengths. MS-DOS is designed to block and deblock its own physical sectors; 128 is not a sacred number in MS-DOS.

  MS-DOS remembers the exact end of file marker. Thus, should one open a file with a logical record length other than the physical record length, MS-DOS remembers exactly where the file ends to the byte, rather than rounded to 128 bytes. This alleviates the need for forcing Control-Z's or the like at the end of a file.

MICROSOFT

Microsoft, Inc.
10800 NE Eighth, Suite 819
Bellevue, WA 98004
206-455-8080   Telex 328945

The Future of MS-DOS

Microsoft plans to enhance MS-DOS. The additional addressing space of the 8086 processor makes multi-tasking a particularly attractive enhancement. An upward migration path to the XENIX operating system through XENIX compatible system calls, "pipes," and "forking" is another planned enhancement.

Plans for MS-DOS also include disk buffering, graphics and cursor positioning, kanji support, multi-user and hard disk support, and networking.

A Microsoft original equipment manufacturer (OEM) marketing brochure describing the strengths of MS-DOS.

Version 2

After the release of PC-specific version 1.0 of MS-DOS, Microsoft worked on an update that contained some bug fixes. Version 1.1 was provided to IBM to run on the upgraded PC released in 1982 and enabled MS-DOS to work with double-sided, 320 KB floppy disks. This version, referred to as 1.25 by all but IBM, was the first version of MS-DOS shipped by other OEMs, including COMPAQ and Zenith.

Even before these intermediate releases were available, however, Microsoft began planning for future versions of MS-DOS. In developing the first version, the programmers had had two primary goals: running translated CP/M-80 software and keeping MS-DOS small. They had neither the time nor the room to include more sophisticated features, such as those typical of Microsoft's UNIX-based multiuser, multitasking operating system, XENIX. But when IBM informed Microsoft that the next major edition of the PC would be the Personal Computer XT with a 10-megabyte fixed disk, a larger, more powerful version of MS-DOS — one closer to the operating system Microsoft had envisioned from the start — became feasible.

There were three particular areas that interested Microsoft: a new, hierarchical file system, installable device drivers, and some type of multitasking. Each of these features contributed to version 2.0, and together they represented a major change in MS-DOS while still maintaining compatibility with version 1.0.

The File System

Primary responsibility for version 2.0 fell to Paul Allen, Mark Zbikowski, and Aaron Reynolds, who wrote (and rewrote) most of the version 2.0 code. The major design issue confronting the developers, as well as the most visible example of its difference from versions 1.0, 1.1, and 1.25, was the introduction of a hierarchical file system to handle the file-management needs of the XT's fixed disk.

Version 1.0 had a single directory for all the files on a floppy disk. That system worked well enough on a disk of limited capacity, but on a 10-megabyte fixed disk a single directory could easily become unmanageably large and cumbersome.

CP/M had approached the problem of high-capacity storage media by using a partitioning scheme that divided the fixed disk into 10 user areas equivalent to 10 separate floppy-disk drives. On the other hand, UNIX, which had traditionally dealt with larger systems, used a branching, hierarchical file structure in which the user could create directories and subdirectories to organize files and make them readily accessible. This was the file-management system implemented in XENIX, and it was the MS-DOS team's choice for handling files on the XT's fixed disk.

/
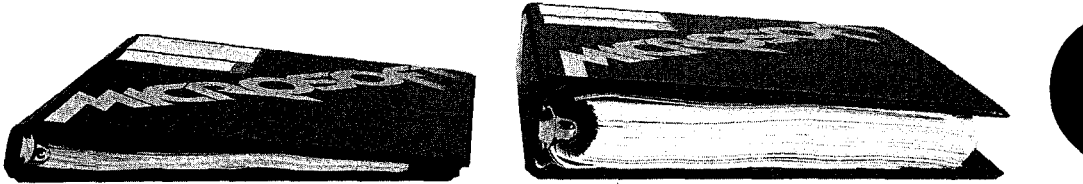
The MS-DOS version 1.0 manual next to the version 2.0 manual.

Partitioning, IBM's initial choice, had the advantages of familiarity, size, and ease of implementation. Many small-system users — particularly software developers — were already familiar with partitioning, if not overly fond of it, from their experience with CP/M. Development time was also a major concern, and the code needed to develop a partitioning scheme would be minimal compared with the code required to manage a hierarchical file system. Such a scheme would also take less time to implement.

However, partitioning had two inherent disadvantages. First, its functionality would decrease as storage capacity increased, and even in 1982, Microsoft was anticipating substantial growth in the storage capacity of disk-based media. Second, partitioning depended on the physical device. If the size of the disk changed, either the number or the size of the partitions must also be changed in the code for both the operating system and the application programs. For Microsoft, with its commitment to hardware independence, partitioning would have represented a step in the wrong direction.

A hierarchical file structure, on the other hand, could be independent of the physical device. A disk could be partitioned logically, rather than physically. And because these partitions (directories) were controlled by the user, they were open-ended and enabled the individual to determine the best way of organizing a disk.

Ultimately, it was a hierarchical file system that found its way into MS-DOS 2.0 and eventually convinced everyone that it was, indeed, the better and more flexible solution to the problem of supporting a fixed disk. The file system was logically consistent with the XENIX file structure, yet physically consistent with the file access incorporated in versions 1.x, and was based on a root, or main, directory under which the user could create a system of subdirectories and sub-subdirectories to hold files. Each file in the system was identified by the directory path leading to it, and the number of subdirectories was limited only by the length of the pathname, which could not exceed 64 characters.

In this file structure, all the subdirectories and the filename in a path were separated from one another by backslash characters, which represented the only anomaly in the XENIX/MS-DOS system of hierarchical files. XENIX used a forward slash as a separator, but versions 1.x of MS-DOS, borrowing from the tradition of DEC operating systems, already used the forward slash for switches in the command line, so Microsoft, at IBM's request, decided to use the backslash as the separator instead. Although the backslash

character created no practical problems, except on keyboards that lacked a backslash, this decision did introduce inconsistency between MS-DOS and existing UNIX-like operating systems. And although Microsoft solved the keyboard problem by enabling the user to change the switch character from a slash to a hyphen, the solution itself created compatibility problems for people who wished to exchange batch files.

Another major change in the file-management system was related to the new directory structure: In order to fully exploit a hierarchical file system, Microsoft had to add a new way of calling file services.

Versions 1.x of MS-DOS used CP/M-like structures called file control blocks, or FCBs, to maintain compatibility with older CP/M-80 programs. The FCBs contained all pertinent information about the size and location of a file but did not allow the user to specify a file in a different directory. Therefore, version 2.0 of MS-DOS needed the added ability to access files by means of handles, or descriptors, that could operate across directory lines.

In this added step toward logical device independence, MS-DOS returned a handle whenever an MS-DOS program opened a file. All further interaction with the file involved only this handle. MS-DOS made all necessary adjustments to an internal structure — different from an FCB — so that the program never had to deal directly with information about the file's location in memory. Furthermore, even if future versions of MS-DOS were to change the structure of the internal control units, program code would not need to be rewritten — the file handle would be the only referent needed, and this would not change.

Putting the internal control units under the supervision of MS-DOS and substituting handles for FCBs also made it possible for MS-DOS to redirect a program's input and output. A system function was provided that enabled MS-DOS to divert the reads or writes directed to one handle to the file or device assigned to another handle. This capability was used by COMMAND.COM to allow output from a file to be redirected to a device, such as a printer, or to be piped to another program. It also allowed system cleanup on program terminations.

Installable Device Drivers

At the time Microsoft began developing version 2.0 of MS-DOS, the company also realized that many third-party peripheral devices were not working well with one another. Each manufacturer had its own way of hooking its hardware into MS-DOS and if two third-party devices were plugged into a computer at the same time, they would often conflict or fail.

One of the hallmarks of IBM's approach to the PC was open architecture, meaning that users could simply slide new cards into the computer whenever new input/output devices, such as fixed disks or printers, were added to the system. Unfortunately, version 1.0 of MS-DOS did not have a corresponding open architecture built into it — the BIOS

contained all the code that permitted the operating system to run the hardware. If independent hardware manufacturers wanted to develop equipment for use with a computer manufacturer's operating system, they would have to either completely rewrite the device drivers or write a complicated utility to read the existing drivers, alter them, add the code to support the new device, and produce a working set of drivers. If the user installed more than one device, these patches would often conflict with one another. Furthermore, they would have to be revised each time the computer manufacturer updated its version of MS-DOS.

By the time work began on version 2.0, the MS-DOS team knew that the ability to install any device driver at run time was vital. They implemented installable device drivers by making the drivers more modular. Like the FAT, IO.SYS (IBMBIO.COM in PC-DOS) became, in effect, a linked list — this time, of device drivers — that could be expanded through commands in the CONFIG.SYS file on the system boot disk. Manufacturers could now write a device driver that the user could install at run time by including it in the CONFIG.SYS file. MS-DOS could then add the device driver to the linked list.

By extension, this ability to install device drivers also added the ability to supersede a previously installed driver — for example, the ANSI.SYS console driver that supports the ANSI standard escape codes for cursor positioning and screen control.

Print Spooling

At IBM's request, version 2.0 of MS-DOS also possessed the undocumented ability to perform rudimentary background processing — an interim solution to a growing awareness of the potentials of multitasking.

Background print spooling was sufficient to meet the needs of most people in most situations, so the print spooler, PRINT.COM, was designed to run whenever MS-DOS had nothing else to do. When the parent application became active, PRINT.COM would be interrupted until the next lull. This type of background processing, though both limited and extremely complex, was exploited by a number of applications, such as SideKick.

Loose Ends and a New MS-DOS

Hierarchical files, installable device drivers, and print spooling were the major design decisions in version 2.0. But there were dozens of smaller changes, too.

For example, with the fixed disk it was necessary to modify the code for automatic logging of disks. This modification meant that MS-DOS had to access the disk more often, and file access became much slower as a result. In trying to find a solution to this problem, Chris Peters reasoned that, if MS-DOS had just checked the disk, there was some minimum time
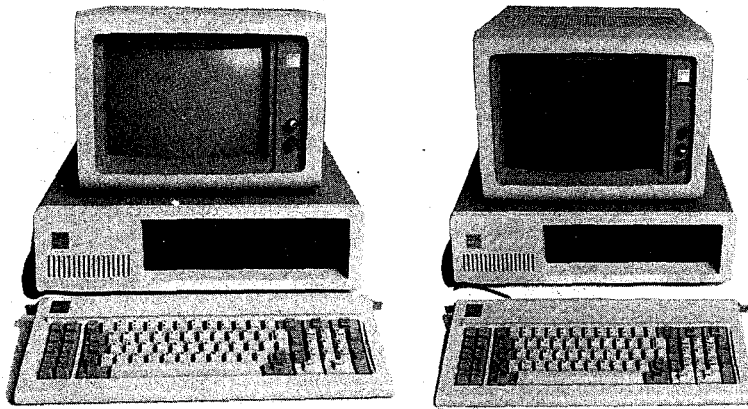
Two members of the IBM line of personal computers for which versions 1 and 2 of MS-DOS were developed. On the left, the original IBM PC (version 1.0 of MS-DOS); on the right, the IBM PC/XT (version 2.0).

a user would need to physically change disks. If that minimum time had not elapsed, the current disk information in RAM — whether for a fixed disk or a floppy — was probably still good.

Peters found that the fastest anyone could physically change disks, even if the disks were damaged in the process, was about two seconds. Reasoning from this observation, he had MS-DOS check to see how much time had gone by since the last disk access. If less than two seconds had elapsed, he had MS-DOS assume that a new disk had not been inserted and that the disk information in RAM was still valid. With this little trick, the speed of file handling in MS-DOS version 2.0 increased considerably.

Version 2.0 was released in March 1983, the product of a surprisingly small team of six developers, including Peters, Mani Ulloa, and Nancy Panners in addition to Allen, Zbikowski, and Reynolds. Despite its complex new features, version 2.0 was only 24 KB of code. Though it maintained its compatibility with versions 1.x, it was in reality a vastly different operating system. Within six months of its release, version 2.0 gained widespread public acceptance. In addition, popular application programs such as Lotus 1-2-3 took advantage of the features of this new version of MS-DOS and thus helped secure its future as the industry standard for 8086 processors.

Versions 2.1 and 2.25

The world into which version 2.0 of MS-DOS emerged was considerably different from the one in which version 1.0 made its debut. When IBM released its original PC, the business market for microcomputers was as yet undefined — if not in scope, at least in terms of who and what would dominate the field. A year and a half later, when the PC/XT came on the scene, the market was much better known. It had, in fact, been heavily influenced by IBM itself. There were still many MS-DOS machines, such as the Tandy 2000 and the Hewlett Packard HP150, that were hardware incompatible with the IBM, but manufacturers of new computers knew that IBM was a force to consider and many chose to compete with the IBM PC by emulating it. Software developers, too, had gained an understanding of business computing and were confident they could position their software accurately in the enormous MS-DOS market.

In such an environment, concerns about the existing base of CP/M software faded as developers focused their attention on the fast-growing business market and MS-DOS quickly secured its position as an industry standard. Now, with the obstacles to MS-DOS diminished, Microsoft found itself with a new concern: maintaining the standard it had created. Henceforth, MS-DOS had to be many things to many people. IBM had requirements; other OEMs had requirements. And sometimes these requirements conflicted.

Hardware Developers

When version 2.0 was released, IBM was already planning to introduce its PCjr. The PCjr would have the ability to run programs from ROM cartridges and, in addition to using half-height 5¼-inch drives, would employ a slightly different disk-controller architecture. Because of these differences from the standard PC line, IBM's immediate concern was for a version 2.1 of MS-DOS modified for the new machine.

For the longer term, IBM was also planning a faster, more powerful PC with a 20-megabyte fixed disk. This prospect meant Microsoft needed to look again at its file-management system, because the larger storage capacity of the 20-megabyte disk stretched the size limitations for the file allocation table as it worked in version 2.0.

However, IBM's primary interest for the next major release of MS-DOS was networking. Microsoft would have preferred to pursue multitasking as the next stage in the development of MS-DOS, but IBM was already developing its IBM PC Network Adapter, a plug-in card with an 80188 chip to handle communications. So as soon as version 2.0 was released, the MS-DOS team, again headed by Zbikowski and Reynolds, began work on a networking version (3.0) of the operating system.

Meanwhile ...

The international market for MS-DOS was not significant in the first few years after the release of the IBM PC and version 1.0 of MS-DOS. IBM did not, at first, ship its Personal Computer to Europe, so Microsoft was on its own there in promoting MS-DOS. In 1982, the company gained a significant advantage over CP/M-86 in Europe by concluding an agreement with Victor, a software company that was very successful in Europe and had already licensed CP/M-86. Working closely with Victor, Microsoft provided special development support for its graphics adaptors and eventually convinced the company to offer its products only on MS-DOS. In Japan, the most popular computers were Z80 machines, and given the country's huge installed base of 8-bit machines, 16-bit computers were not taking hold. Mitsubishi, however, offered a 16-bit computer. Although CP/M-86 was Mitsubishi's original choice for an operating system, Microsoft helped get Multiplan and FORTRAN running on the CP/M-86 system, and eventually won the manufacturer's support for MS-DOS.

DOS 3.0

Irresistible DOS 3.0

International support, file-sharing capabilities, and many other features in DOS 3.0 result in a significantly enhanced operating system.

The Ascent of DOS

● Hands On: Operating Systems

MS-DOS 2.00: A Hands-On Tutorial

Tom Sheldon

Although the announcement of the IBM Personal Computer XT grabbed the headlines after its unveiling, the latest version of Microsoft's Disk Operating System (DOS 2.00), introduced on the same day, marks a significant extension of the capabilities available to all PC users for managing the flow of data between the PC's processor and peripheral devices. This article takes a close look at some of those enhancements, especially the tree-structured filing system and new batch file subcommands.

Even before the latest set of changes, MS-DOS was one of the best buys for the PC. For $40 version 1.10 of this package combines an editor, a file-keeping system, batch processing, a linker and debug program, and much more. Many users touch only the surface of this package. Most of their time is spent in the applications environment of a prepackaged program. The typical word processing operator, for example, rarely uses any DOS commands besides FORMAT and COPY. Some users touch on batch processing and do elaborate directory and copy commands using wild cards or global characters.

The avid computer user, on the other hand, has scoured the manual looking for new and interesting commands and procedures. DOS version 2.00 promises to be a stimulating package for these users, and considering all the new features you get for only $60, it would be a bargain at twice the price.

The Forest of Files
DOS 2.00 utilizes a tree-structured filing system. In this type of arrangement a root, or base, directory holds a certain number of files. Some of these files are themselves directories; they are actually subdirectories of the root directory and can contain files and subdirectories themselves.

170   *Volume 1, Number 3*

nd therefore can access only a maximum of 1MB of memory.
DOS 3.0 is upwardly compatible with DOS 2.1 but does not replace it, one reason being that DOS 3.0 is substantially bigger. Because DOS 3.0 requires at least 36KB of memory (DOS 2.1 requires 24KB), IBM recommends that minimum user memory be 96KB—or 128KB with a fixed disk.
Because its size has been increased so dramatically, DOS is now supplied on two double-sided diskettes as opposed to the single-sided diskettes used for previous DOS releases. One reason for the increase in size is that many parts of the operating system appear to have been rewritten in the C language. Also, much of the network support promised for DOS 3.1 is already installed, including file sharing and file locking.

PC TECH JOURNAL

y changes of ... were acter. Version ... only. erent charac- drives unro- tortable PC. to date that ommands least one hough it was n new ver- extended ilities of the eficient step in ng system for e 8028b-based . Apart from 2MB floppy ally obvious ty to specify a d name pre- ively misused

eature of ersion is that a path format nd file to that e, typing each for the directory of and there.

TECH JOURNAL

A sample of the reviews that appeared with each new version of MS-DOS.

In the software arena, by the time development was underway on the 2.x releases of MS-DOS, Microsoft's other customers were becoming more vocal about their own needs. Several wanted a networking capability, adding weight to IBM's request, but a more urgent need for many — a need *not* shared by IBM at the time — was support for international products. Specifically, these manufacturers needed a version of MS-DOS that could be sold in other countries — a version of MS-DOS that could display messages in other languages and adapt to country-specific conventions, such as date and time formats.

Microsoft, too, wanted to internationalize MS-DOS, so the MS-DOS team, while modifying the operating system to support the PCjr, also added functions and a COUNTRY command that allowed users to set the date and time formats and other country-dependent variables in the CONFIG.SYS file.

```
NEC PC-9800 Series Personal Computer

マイクロソフト MS-DOS バージョン 3.10
Copyright 1981,1985 Microsoft Corp. / NEC Corporation

  連文節変換が使用可能です
  辞書は、カレントドライブの NECDIC .SYS です

COMMAND.バージョン 3.10

A>DIR /W
  ドライブ A: のディスクのボリュームラベルは KAWAI_RYU
  ディレクトリは A:¥BIN

.                    ..      ASSIGN  COM   ATTRIB  EXE   BACKUP  EXE
CHKDSK   EXE   COPY2   COM   COPYA   COM   DISKCOPY COM  MOUSE   SYS
FC       EXE   FIND    EXE   FORMAT  EXE   KEY     COM   LABEL   EXE
MORE     COM   SPEED   COM   SWITCH  COM   SYS     EXE   SORT    COM
      20 個のファイルがあります.
   3604480 バイトが使用可能です.

A>マイクロソフト株式会社

R【かな】    漢字MS－DOS                               <
```

A Kanji screen with the MS-DOS copyright message.

At about the same time, another international requirement appeared. The Japanese market for MS-DOS was growing, and the question of supporting 7000 Kanji characters (ideograms) arose. The difficulty with Kanji is that it requires dual-byte characters. For English and most European character sets, one byte corresponds to one character. Japanese characters, however, sometimes use one byte, sometimes two. This variability creates problems in parsing, and as a result MS-DOS had to be modified to parse a string from the beginning, rather than back up one character at a time.

This support for individual country formats and Kanji appeared in version 2.01 of MS-DOS. IBM did not want this version, so support for the PCjr, developed by Zbikowski, Reynolds, Ulloa, and Eric Evans, appeared separately in version 2.1, which went only to IBM and did not include the modifications for international MS-DOS.

Different customers, different versions

As early as version 1.25, Microsoft faced the problem of trying to satisfy those OEM customers that wanted to have the same version of MS-DOS as IBM. Some, such as COMPAQ, were in the business of selling 100-percent compatibility with IBM. For them, any difference between their version of the operating system and IBM's introduced the possibility of incompatibility. Satisfying these requests was difficult, however, and it was not until version 3.1 that Microsoft was able to supply a system that other OEMs agreed was identical with IBM's.

Before then, to satisfy the OEM customers, Microsoft combined versions 2.1 and 2.01 to create version 2.11. Although IBM did not accept this because of the internationalization code, version 2.11 became the standard version for all non-IBM customers running any form of MS-DOS in the 2.x series. Version 2.11 was sold worldwide and translated into about 10 different languages. Two other intermediate versions provided support for Hangeul (the Korean character set) and Chinese Kanji.

Software Concerns

After the release of version 2.0, Microsoft also gained an appreciation of the importance —
and difficulty — of supporting the people who were developing software for MS-DOS.

Software developers worried about downward compatibility. They also worried about
upward compatibility. But despite these concerns, they sometimes used programming
practices that could guarantee neither. When this happened and the resulting programs
were successful, it was up to Microsoft to ensure compatibility.

For example, because the information about the internals of the BIOS and the ROM inter-
face had been published, software developers could, and often did, work directly with the
hardware in order to get more speed. This meant sidestepping the operating system for
some operations. However, by choosing to work at the lower levels, these developers lost
the protection provided by the operating system against hardware changes. Thus, when
low-level changes were made in the hardware, their programs either did not work or did
not run cooperatively with other applications.

Another software problem was the continuing need for compatibility with CP/M. For
example, in CP/M, programmers would call a fixed address in low memory in order to re-
quest a function; in MS-DOS, they would request operating-system services by executing a
software interrupt. To support older software, the first version of MS-DOS allowed a pro-
gram to request functions by either method. One of the CP/M-based programs supported
in this fashion was the very popular WordStar. Since Microsoft could not make changes in
MS-DOS that would make it impossible to run such a widely used program, each new ver-
sion of MS-DOS had to continue supporting CP/M-style calls.

A more pervasive CP/M-related issue was the use of FCB-style calls for file and record
management. The version 1.x releases of MS-DOS had used FCB-style calls exclusively, as
had CP/M. Version 2.0 introduced the more efficient and flexible handle calls, but Microsoft
could not simply abolish the old FCB-style calls, because so many popular programs used
them. In fact, some of Microsoft's own languages used them. So, MS-DOS had to support
both types of calls in the version 2.x series. To encourage the use of the new handle calls,
however, Microsoft made it easy for MS-DOS users to upgrade to version 2.0. In addition,
the company convinced IBM to require version 2.0 for the PC/XT and also encouraged
software developers to require 2.0 for their applications.

At first, both software developers and OEM customers were reluctant to require 2.0
because they were concerned about problems with the installed user base of 1.0
systems — requiring version 2.0 meant supporting both sets of calls. Applications also
needed to be able to detect which version of the operating system the user was running.
For versions 1.x, the programs would have to use FCB calls; for versions 2.x, they would
use the file handles to exploit the flexibility of MS-DOS more fully.

All told, it was an awkward period of transition, but by the time Microsoft began work on
version 3.0 and the support for IBM's upcoming 20-megabyte fixed disk, it had become
apparent that the change had been in everyone's best interest.

Version 3

The types of issues that began to emerge as Microsoft worked toward version 3.0, MS-DOS for networks, exaggerated the problems of compatibility that had been encountered before.

First, networking, with or without a multitasking capability, requires a level of cooperation and compatibility among programs that had never been an issue in earlier versions of MS-DOS. As described by Mark Zbikowski, one of the principals involved in the project, "there was a very long period of time between 2.1 and 3.0 — almost a year and a half. During that time, we believed we understood all the problems involved in making DOS a networking product. [But] as time progressed, we realized that we didn't fully understand it, either from a compatibility standpoint or from an operating-system standpoint. We knew very well how it [DOS] ran in a single-tasking environment, but we started going to this new environment and found places where it came up short."

In fact, the great variability in programs and programming approaches that MS-DOS supported eventually proved to be one of the biggest obstacles to the development of a sophisticated networking system and, in the longer term, to the addition of true multitasking.

Further, by the time Microsoft began work on version 3.0, the programming style of the MS-DOS team had changed considerably. The team was still small, with a core group of just five people: Zbikowski, Reynolds, Peters, Evans, and Mark Bebic. But the concerns for maintainability that had dominated programming in larger systems had percolated down to the MS-DOS environment. Now, the desire to use tricks to optimize for speed had to be tempered by the need for clarity and maintainability, and the small package of tightly written code that was the early MS-DOS had to be sacrificed for the same reasons.

Version 3.0

All told, the work on version 3.0 of MS-DOS proved to be long and difficult. For a year and a half, Microsoft grappled with problems of software incompatibility, remote file management, and logical device independence at the network level. Even so, when IBM was ready to announce its new Personal Computer AT, the network software for MS-DOS was not quite ready, so in August 1984, Microsoft released version 3.0 to IBM without network software.

Version 3.0 supported the AT's larger fixed disk, its new CMOS clock, and its high-capacity 1.2-megabyte floppy disks. It also provided the same international support included earlier in versions 2.01 and 2.11. These features were made available to Microsoft's other OEM customers as version 3.05.

ROM

ROM

ROM

ROM

NewDIR

GETEOF

NEXTSFC

ALLOCATE
Release
RELBLKS

FNDCLUS
SKPCLP
OPTIMIZE

misc. DISK-RESET ──── ←─ "IGNORE" FAIL
Rename RENAME
finfo Set file ATTR
CTRLC Reset Envir ──── ←─ "IGNORE" FAIL
Close CLOSE
Delete DELETE

BUFWR

ISEARCH Do

c+2
GETPATH
SETDIRSRCH

FINDEntry
NEXT ENTRY

UNPACK
(A)

PACK

MAPCLUST

NETBUF

FLUSHBUF → CHECKFLUSH → BUFWR
NETFLUSHBUF NET

FromBuf
DIR
FAT
DATA

NSR READ
WRITE ....→

DISKWRITE
DATA
ARI

DWRITE

Net Marks
its buffers
DATA

123

Aaron Reynolds's diagram of version 3.0's network support, sketched out to enable him to add the fail option to Interrupt 24 and find all places where existing parts of MS-DOS were affected. Even after networking had become a reality, Reynolds kept this diagram pinned to his office wall simply because "it was so much work to put together."
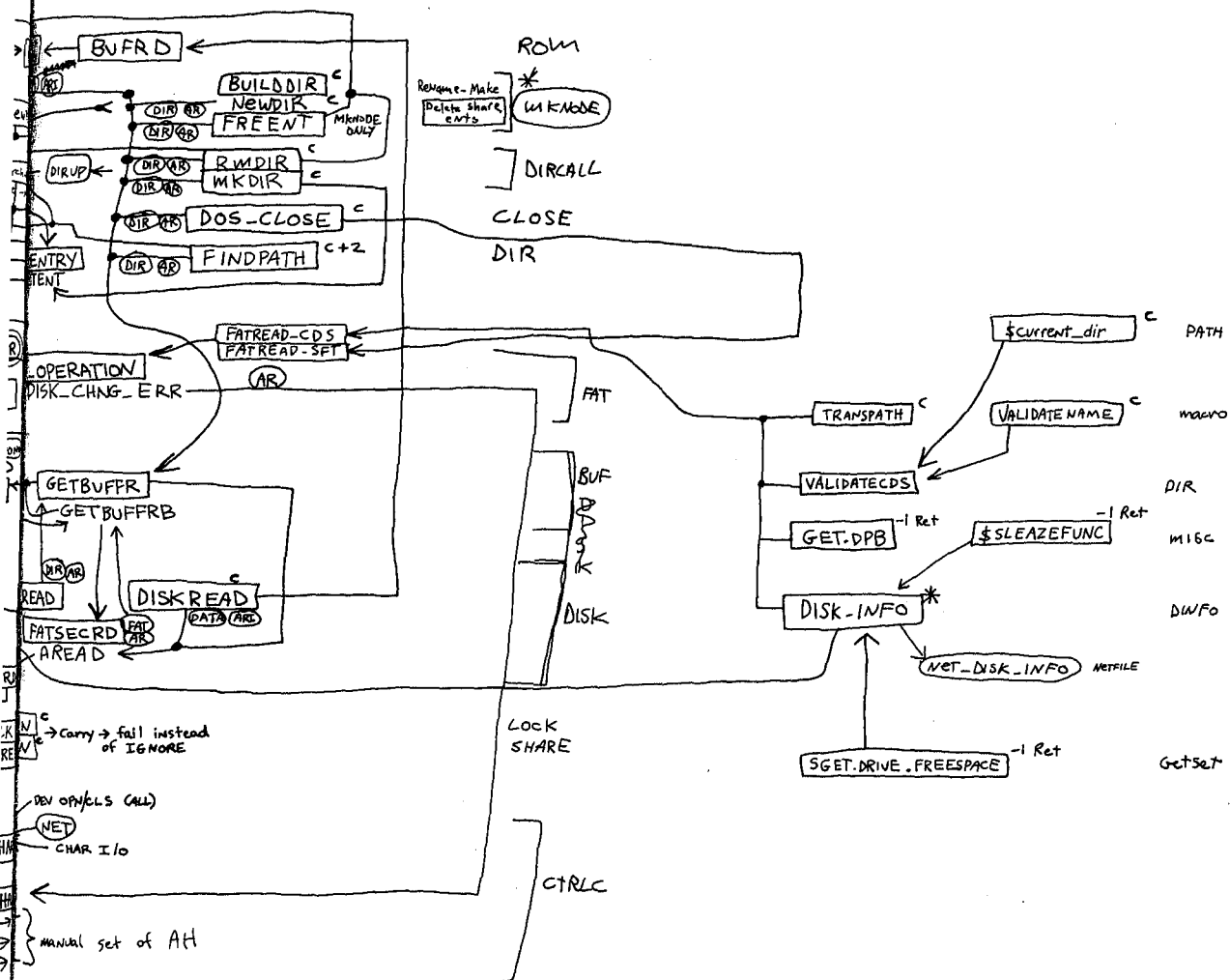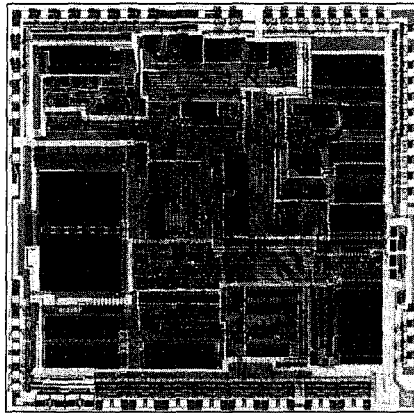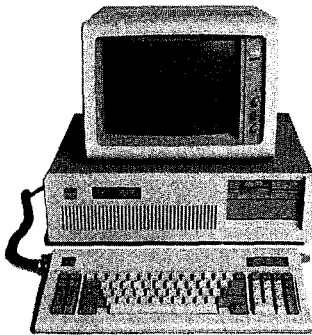
The Intel 80286 micro-processor, the chip at the heart of the IBM PC/AT, which is shown beside it. Version 3.0 of MS-DOS, developed for this machine, offered support for networks and the PC/AT's 1.2-megabyte floppy disk drive and built-in CMOS clock.

But version 3.0 was not a simple extension of version 2.0. In laying the foundation for net-working, the MS-DOS team had completely redesigned and rewritten the DOS kernel.

Different as it was from version 1.0, version 2.0 had been built on top of the same structure. For example, whereas file requests in MS-DOS 1.0 used FCBs, requests in version 2.0 used file handles. However, the version 2.0 handle calls would simply parse the pathname and then use the underlying FCB calls in the same way as version 1.0. The redirected input and output in version 2.0 further complicated the file-system requests. When a program used one of the CP/M-compatible calls for character input or output, MS-DOS 2.0 first opened a handle and then turned it back into an FCB call at a lower level. Version 3.0 eliminated this redundancy by eliminating the old FCB input/output code of versions 1 and 2, replacing it with a standard set of I/O calls that could be called directly by both FCB calls and handle calls. The look-alike calls for CP/M-compatible character I/O were included as part of the set of handle calls. As a result of this restructuring, these calls were distinctly faster in version 3.0 than in version 2.0.

More important than the elimination of inefficiencies, however, was the fact that this new structure made it easier to handle network requests under the ISO Open System Intercon-nect model Microsoft was using for networking. The ISO model describes a number of protocol layers, ranging from the application-to-application interface at the top level down to the physical link — plugging into the network — at the lowest level. In the middle is the transport layer, which manages the actual transfer of data. The layers above the transport layer belong to the realm of the operating system; the layers below the transport layer are traditionally the domain of the network software or hardware.

On the IBM PC network, the transport layer and the server functions were handled by IBM's Network Adapter card and the task of MS-DOS was to support this hardware. For its other OEM customers, however, Microsoft needed to supply both the transport and the server functions as software. Although version 3.0 did not provide this general-purpose networking software, it did provide the basic support for IBM's networking hardware.

The support for IBM consisted of redirector and sharer software. MS-DOS used an ap-proach to networking in which remote requests were routed by a redirector that was able

to interact with the transport layer of the network. The transport layer was composed of the device drivers that could reliably transfer data from one part of the network to another. Just before a call was sent to the newly designed low-level file I/O code, the operating system determined whether the call was local or remote. A local call would be allowed to fall through to the local file I/O code; a remote call would be passed to the redirector which, working with the operating system, would make the resources on a remote machine appear as if they were local.

Version 3.1

Both the redirector and the sharer interfaces for IBM's Network Adapter card were in place in version 3.0 when it was delivered to IBM, but the redirector itself wasn't ready. Version 3.1, completed by Zbikowski and Reynolds and released three months later, completed this network support and made it available in the form of Microsoft Networks for use on non-IBM network cards.

Microsoft Networks was built on the concept of "services" and "consumers." Services were provided by a file server, which was part of the Networks application and ran on a computer dedicated to the task. Consumers were programs on various network machines. Requests for information were passed at a high level to the file server; it was then the responsibility of the file server to determine where to find the information on the disk. The requesting programs — the consumers — did not need any knowledge of the remote machine, not even what type of file system it had.

This ability to pass a high-level request to a remote server without having to know the details of the server's file structure allowed another level of generalization of the system. In MS-DOS 3.1, different types of file systems could be accessed on the same network. It was possible, for example, to access a XENIX machine across the network from an MS-DOS machine and to read data from XENIX files.

Microsoft Networks was designed to be hardware independent. Yet the variability of the classes of programs that would be using its structures was a major problem in developing a networking system that would be transparent to the user. In evaluating this variability, Microsoft identified three types of programs:

- First were the MS-DOS-compatible programs. These used only the documented software-interrupt method of requesting services from the operating system and would run on any MS-DOS machine without problems.
- Second were the MS-DOS-based programs. These would run on IBM-compatible computers but not necessarily on all MS-DOS machines.
- Third were the programs that used undocumented features of MS-DOS or that addressed the hardware directly. These programs tended to have the best performance but were also the most difficult to support.

Of these, Microsoft officially encouraged the writing of MS-DOS-compatible programs for use on the network.

Network concerns

The file-access module was changed in version 3.0 to simplify file management on the network, but this did not solve all the problems. For instance, MS-DOS still needed to handle FCB requests from programs that used them, but many programs would open an FCB and never close it. One of the functions of the server was to keep track of all open files on the network, and it ran into difficulties when an FCB was opened 50 or 100 times and never closed. To solve this problem, Microsoft introduced an FCB cache in version 3.1 that allowed only four FCBs to be open at any one time. If a fifth FCB was opened, the least recently used one was closed automatically and released. In addition, an FCBS command was added in the CONFIG.SYS file to allow the user or network manager to change the maximum number of FCBs that could be open at any one time and to protect some of the FCBs from automatic closure.

In general, the logical device independence that had been a goal of MS-DOS acquired new meaning — and generated new problems — with networking. One problem concerned printers on the network. Commonly, networks are used to allow several people to share a printer. The network could easily accommodate a program that would open the printer, write to it, and close it again. Some programs, however, would try to use the direct IBM BIOS interface to access the printer. To handle this situation, Microsoft's designers had to develop a way for MS-DOS to intercept these BIOS requests and filter out the ones the server could not handle. Once this was accomplished, version 3.1 was able to handle most types of printer output on the network in a transparent manner.

Version 3.2

In January 1986, Microsoft released another revision of MS-DOS, version 3.2, which supported 3½-inch floppy disks. Version 3.2 also moved the formatting function for a device out of the FORMAT utility routine and into the device driver, eliminating the need for a special hardware-dependent program in addition to the device driver. It included a sample installable-block-device driver and, finally, benefited the users and manufacturers of IBM-compatible computers by including major rewrites of the MS-DOS utilities to increase compatibility with those of IBM.

The Future

Since its appearance in 1981, MS-DOS has taken and held an enviable position in the microcomputer environment. Not only has it "taught" millions of personal computers "how to think," it has taught equal millions of people how to use computers. Many highly sophisticated computer users can trace their first encounter with these machines to the original IBM PC and version 1.0 of MS-DOS. The MS-DOS command interface is the one with which they are comfortable and it is the MS-DOS file structure that, in one way or another, they wander through with familiarity.

Microsoft has stated its commitment to ensuring that, for the foreseeable future, MS-DOS will continue to evolve and grow, changing as it has done in the past to satisfy the needs of its millions of users. In the long term, MS-DOS, the product of a surprisingly small group of gifted people, will undoubtedly remain the industry standard for as long as 8086-based (and to some extent, 80286-based) microcomputers exist in the business world. The story of MS-DOS will, of course, remain even longer. For this operating system has earned its place in microcomputing history.

JoAnne Woodcock

Section II
Programming in the MS-DOS Environment

Part A
Structure of MS-DOS

Article 1
An Introduction to MS-DOS

An operating system is a set of interrelated supervisory programs that manage and control computer processing. In general, an operating system provides

● Storage management
● Processing management
● Security
● Human interface

Existing operating systems for microcomputers fall into three major categories: ROM monitors, traditional operating systems, and operating environments. The general characteristics of the three categories are listed in Table 1-1.

Table 1-1. Characteristics of the Three Major Types of Operating Systems.

	ROM Monitor	Traditional Operating System	Operating Environment
Complexity	Low	Medium	High
Built on	Hardware	BIOS	Operating system
Delivered on	ROM	Disk	Disk
Programs on	ROM	Disk	Disk
Peripheral support	Physical	Logical	Logical
Disk access	Sector	File system	File system
Example	PC ROM BIOS	MS-DOS	Microsoft Windows

A ROM monitor is the simplest type of operating system. It is designed for a particular hardware configuration and provides a program with basic — and often direct — access to peripherals attached to the computer. Programs coupled with a ROM monitor are often used for dedicated applications such as controlling a microwave oven or controlling the engine of a car.

A traditional microcomputer operating system is built on top of a ROM monitor, or BIOS (basic input/output system), and provides additional features such as a file system and logical access to peripherals. (Logical access to peripherals allows applications to run in a hardware-independent manner.) A traditional operating system also stores programs in files on peripheral storage devices and, on request, loads them into memory for execution. MS-DOS is a traditional operating system.

An operating environment is built on top of a traditional operating system. The operating environment provides additional services, such as common menu and forms support, that

simplify program operation and make the user interface more consistent. Microsoft Windows is an operating environment.

MS-DOS System Components

The Microsoft Disk Operating System, MS-DOS, is a traditional microcomputer operating system that consists of five major components:

● The operating-system loader
● The MS-DOS BIOS
● The MS-DOS kernel
● The user interface (shell)
● Support programs

Each of these is introduced briefly in the following pages. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: The Components of MS-DOS.

The operating-system loader

The operating-system loader brings the operating system from the startup disk into RAM.

The complete loading process, called bootstrapping, is often complex, and multiple loaders may be involved. (The term *bootstrapping* came about because each level pulls up the next part of the system, like pulling up on a pair of bootstraps.) For example, in most standard MS-DOS-based microcomputer implementations, the ROM loader, which is the first program the microcomputer executes when it is turned on or restarted, reads the disk bootstrap loader from the first (boot) sector of the startup disk and executes it. The disk bootstrap loader, in turn, reads the main portions of MS-DOS — MSDOS.SYS and IO.SYS (IBMDOS.COM and IBMBIO.COM with PC-DOS) — from conventional disk files into memory. The special module SYSINIT within MSDOS.SYS then initializes MS-DOS's tables and buffers and discards itself. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

(The term loader is also used to refer to the portion of the operating system that brings application programs into memory for execution. This loader is different from the ROM loader and the operating-system loader.)

The MS-DOS BIOS

The MS-DOS BIOS, loaded from the file IO.SYS during system initialization, is the layer of the operating system that sits between the operating-system kernel and the hardware. An application performs input and output by making requests to the operating-system kernel, which, in turn, calls the MS-DOS BIOS routines that access the hardware directly. *See* SYSTEM CALLS. This division of function allows application programs to be written in a hardware-independent manner.

The MS-DOS BIOS consists of some initialization code and a collection of device drivers. (A device driver is a specialized program that provides support for a specific device such as

a display or serial port.) The device drivers are responsible for hardware access and for the interrupt support that allows the associated devices to signal the microprocessor that they need service.

The device drivers contained in the file IO.SYS, which are always loaded during system initialization, are sometimes referred to as the resident drivers. With MS-DOS versions 2.0 and later, additional device drivers, called installable drivers, can optionally be loaded during system initialization as a result of DEVICE directives in the system's configuration file. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: Customizing ms-dos: Installable Device Drivers; USER COMMANDS: config.sys:device.

The MS-DOS kernel

The services provided to application programs by the MS-DOS kernel include

- Process control
- Memory management
- Peripheral support
- A file system

The MS-DOS kernel is loaded from the file MSDOS.SYS during system initialization.

Process control

Process, or task, control includes program loading, task execution, task termination, task scheduling, and intertask communication.

Although MS-DOS is not a multitasking operating system, it can have multiple programs residing in memory at the same time. One program can invoke another, which then becomes the active (foreground) task. When the invoked task terminates, the invoking program again becomes the foreground task. Because these tasks never execute simultaneously, this stack-like operation is still considered to be a single-tasking operating system.

MS-DOS does have a few "hooks" that allow certain programs to do some multitasking on their own. For example, terminate-and-stay-resident (TSR) programs such as PRINT use these hooks to perform limited concurrent processing by taking control of system resources while MS-DOS is "idle," and the Microsoft Windows operating environment adds support for nonpreemptive task switching.

The traditional intertask communication methods include semaphores, queues, shared memory, and pipes. Of these, MS-DOS formally supports only pipes. (A pipe is a logical, unidirectional, sequential stream of data that is written by one program and read by another.) The data in a pipe resides in memory or in a disk file, depending on the implementation; MS-DOS uses disk files for intermediate storage of data in pipes because it is a single-tasking operating system.

Memory management

Because the amount of memory a program needs varies from program to program, the traditional operating system ordinarily provides memory-management functions. Memory

requirements can also vary during program execution, and memory management is especially necessary when two or more programs are present in memory at the same time.

MS-DOS memory management is based on a pool of variable-size memory blocks. The two basic memory-management actions are to allocate a block from the pool and to return an allocated block to the pool. MS-DOS allocates program space from the pool when the program is loaded; programs themselves can allocate additional memory from the pool. Many programs perform their own memory management by using a local memory pool, or heap — an additional memory block allocated from the operating system that the application program itself divides into blocks for use by its various routines. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Memory Management.

Peripheral support

The operating system provides peripheral support to programs through a set of operating-system calls that are translated by the operating system into calls to the appropriate device driver.

Peripheral support can be a direct logical-to-physical-device translation or the operating system can interject additional features or translations. Keyboards, displays, and printers usually require only logical-to-physical-device translations; that is, the data is transferred between the application program and the physical device with minimal alterations, if any, by the operating system. The data provided by clock devices, on the other hand, must be transformed to operating-system-dependent time and date formats. Disk devices — and block devices in general — have the greatest number of features added by the operating system. *See* The File System below.

As stated earlier, an application need not be concerned with the details of peripheral devices or with any special features the devices might have. Because the operating system takes care of all the logical-to-physical-device translations, the application program need only make requests of the operating system.

The file system

The file system is one of the largest portions of an operating system. A file system is built on the storage medium of a block device (usually a floppy disk or a fixed disk) by mapping a directory structure and files onto the physical unit of storage. A file system on a disk contains, at a minimum, allocation information, a directory, and space for files. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

The file allocation information can take various forms, depending on the operating system, but all forms basically track the space used by files and the space available for new data. The directory contains a list of the files stored on the device, their sizes, and information about where the data for each file is located.

Several different approaches to file allocation and directory entries exist. MS-DOS uses a particular allocation method called a file allocation table (FAT) and a hierarchical directory

structure. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: Structure of ms-dos: MS-DOS Storage Devices; Programming for ms-dos: Disk Directories and Volume Labels.

The file granularity available through the operating system also varies depending on the implementation. Some systems, such as MS-DOS, have files that are accessible to the byte level; others are restricted to a fixed record size.

File systems are sometimes extended to map character devices as if they were files. These device "files" can be opened, closed, read from, and written to like normal disk files, but all transactions occur directly with the specified character device. Device files provide a useful consistency to the environment for application programs; MS-DOS supports such files by assigning a reserved logical name (such as CON or PRN) to each character device.

The user interface

The user interface for an operating system, also called a shell or command processor, is generally a conventional program that allows the user to interact with the operating system itself. The default MS-DOS user interface is a replaceable shell program called COMMAND.COM.

One of the fundamental tasks of a shell is to load a program into memory on request and pass control of the system to the program so that the program can execute. When the program terminates, control returns to the shell, which prompts the user for another command. In addition, the shell usually includes functions for file and directory maintenance and display. In theory, most of these functions could be provided as programs, but making them resident in the shell allows them to be accessed more quickly. The tradeoff is memory space versus speed and flexibility. Early microcomputer-based operating systems provided a minimal number of resident shell commands because of limited memory space; modern operating systems such as MS-DOS include a wide variety of these functions as internal commands.

Support programs

The MS-DOS software includes support programs that provide access to operating-system facilities not supplied as resident shell commands built into COMMAND.COM. Because these programs are stored as executable files on disk, they are essentially the same as application programs and MS-DOS loads and executes them as it would any other program.

The support programs provided with MS-DOS, often referred to as external commands, include disk utilities such as FORMAT and CHKDSK and more general support programs such as EDLIN (a line-oriented text editor) and PRINT (a TSR utility that allows files to be printed while another program is running). *See* USER COMMANDS.

MS-DOS releases

MS-DOS and PC-DOS have been released in a number of forms, starting in 1981. *See* THE DEVELOPMENT OF MS-DOS. The major MS-DOS and PC-DOS implementations are summarized in the following table.

Version	Date	Special Characteristics
PC-DOS 1.0	1981	First operating system for the IBM PC Record-oriented files
PC-DOS 1.1	1982	Double-sided-disk support
MS-DOS 1.25	1982	First OEM release of MS-DOS
MS-DOS/PC-DOS 2.0	1983	Operating system for the IBM PC/XT UNIX/XENIX-like file system Installable device drivers Byte-oriented files Support for fixed disks
PC-DOS 2.1		Operating system for the IBM PCjr
MS-DOS 2.11		Internationalization support 2.0x bug fixes
MS-DOS/PC-DOS 3.0	1984	Operating system for the IBM PC/AT Support for 1.2 MB floppy disks Support for large fixed disks Support for file and record locking Application control of print spooler
MS-DOS/PC-DOS 3.1	1984	Support for MS Networks
MS-DOS/PC-DOS 3.2	1986	3.5-inch floppy-disk support Disk track formatting support added to device drivers
MS-DOS/PC-DOS 3.3	1987	Support for the IBM PS/2 Enhanced internationalization support Improved file-system performance Partitioning support for disks with capacity above 32 MB

PC-DOS version 1.0 was the first commercial version of MS-DOS. It was developed for the original IBM PC, which was typically shipped with 64 KB of memory or less. MS-DOS and PC-DOS versions 1.x were similar in many ways to CP/M, the popular operating system for 8-bit microcomputers based on the Intel 8080 (the predecessor of the 8086). These versions of MS-DOS used a single-level file system with no subdirectory support and did not support installable device drivers or networks. Programs accessed files using file control blocks (FCBs) similar to those found in CP/M programs. File operations were record oriented, again like CP/M, although record sizes could be varied in MS-DOS.

Although they retained compatibility with versions 1.x, MS-DOS and PC-DOS versions 2.x represented a major change. In addition to providing support for fixed disks, the new versions switched to a hierarchical file system like that found in UNIX/XENIX and to file-handle access instead of FCBs. (A file handle is a 16-bit number used to reference an internal table that MS-DOS uses to keep track of currently open files; an application program has no access to this internal table.) The UNIX/XENIX-style file functions allow files to be treated as a byte stream instead of as a collection of records. Applications can read or write 1 to 65535 bytes in a single operation, starting at any byte offset within the file. Filenames

ZTE (USA) 1007, Page 71

used for opening a file are passed as text strings instead of being parsed into an FCB. Installable device drivers were another major enhancement.

MS-DOS and PC-DOS versions 3.x added a number of valuable features, including support for the added capabilities of the IBM PC/AT, for larger-capacity disks, and for file-locking and record-locking functions. Network support was added by providing hooks for a redirector (an additional operating-system module that has the ability to redirect local system service requests to a remote system by means of a local area network).

With all these changes, MS-DOS remains a traditional single-tasking operating system. It provides a large number of system services in a transparent fashion so that, as long as they use only the MS-DOS-supplied services and refrain from using hardware-specific operations, applications developed for one MS-DOS machine can usually run on another.

Basic MS-DOS Requirements

Foremost among the requirements for MS-DOS is an Intel 8086-compatible microprocessor. *See* Specific Hardware Requirements below.

The next requirement is the ROM bootstrap loader and enough RAM to contain the MS-DOS BIOS, kernel, and shell and an application program. The RAM must start at address 0000:0000H and, to be managed by MS-DOS, must be contiguous. The upper limit for RAM is the limit placed upon the system by the 8086 family — 1 MB.

The final requirement for MS-DOS is a set of devices supported by device drivers, including at least one block device, one character device, and a clock device. The block device is usually the boot disk device (the disk device from which MS-DOS is loaded); the character device is usually a keyboard/display combination for interaction with the user; the clock device, required for time-of-day and date support, is a hardware counter driven in a submultiple of one second.

Specific hardware requirements

MS-DOS uses several hardware components and has specific requirements for each. These components include

- An 8086-family microprocessor
- Memory
- Peripheral devices
- A ROM BIOS (PC-DOS only)

The microprocessor

MS-DOS runs on any machine that uses a microprocessor that executes the 8086/8088 instruction set, including the Intel 8086, 80C86, 8088, 80186, 80188, 80286, and 80386 and the NEC V20, V30, and V40.

The 80186 and 80188 are versions of the 8086 and 8088, integrated in a single chip with direct memory access, timer, and interrupt support functions. PC-DOS cannot usually run on the 80186 or 80188 because these chips have internal interrupt and interface register addresses that conflict with addresses used by the PC ROM BIOS. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Hardware Interrupt Handlers. MS-DOS, however, does not have address requirements that conflict with those interrupt and interface areas.

The 80286 has an extended instruction set and two operating modes: real and protected. Real mode is compatible with the 8086/8088 and runs MS-DOS. Protected mode, used by operating systems like UNIX/XENIX and MS OS/2, is partially compatible with real mode in terms of instructions but provides access to 16 MB of memory versus only 1 MB in real mode (the limit of the 8086/8088).

The 80386 adds further instructions and a third mode called virtual 86 mode. The 80386 instructions operate in either a 16-bit or a 32-bit environment. MS-DOS can run on the 80386 in real or virtual 86 mode, although the latter requires additional support in the form of a virtual machine monitor such as Windows /386.

Memory requirements

At a minimum, MS-DOS versions 1.x require 64 KB of contiguous RAM from the base of memory to do useful work; versions 2.x and 3.x need at least 128 KB. The maximum is 1 MB, although most MS-DOS machines have a 640 KB limit for IBM PC compatibility. MS-DOS can use additional noncontiguous RAM for a RAMdisk if the proper device driver is included. (Other uses for noncontiguous RAM include buffers for video displays, fixed disks, and network adapters.)

PC-DOS has the same minimum memory requirements but has an upper limit of 640 KB on the initial contiguous RAM, which is generally referred to as conventional memory. This limit was imposed by the architecture of the original IBM PC, with the remaining area above 640 KB reserved for video display buffers, fixed disk adapters, and the ROM BIOS. Some of the reserved areas include

Base Address	Size (bytes)	Description
A000:0000H	10000H (64 KB)	EGA video buffer
B000:0000H	1000H (4 KB)	Monochrome video buffer
B800:0000H	4000H (16 KB)	Color/graphics video buffer
C800:0000H	4000H (16 KB)	Fixed-disk ROM
F000:0000H	10000H (64 KB)	PC ROM BIOS and ROM BASIC

The bottom 1024 bytes of system RAM (locations 00000-003FFH) are used by the microprocessor for an interrupt vector table — that is, a list of addresses for interrupt handler routines. MS-DOS uses some of the entries in this table, such as the vectors for interrupts 20H through 2FH, to store addresses of its own tables and routines and to provide linkage to its services for application programs. The IBM PC ROM BIOS and IBM PC BASIC use many additional vectors for the same purposes.

Peripheral devices

MS-DOS can support a wide variety of devices, including floppy disks, fixed disks, CD ROMs, RAMdisks, and digital tape drives. The required peripheral support for MS-DOS is provided by the MS-DOS BIOS or by installable device drivers.

Five logical devices are provided in a basic MS-DOS system:

Device Name	Description
CON	Console input and output
PRN	Printer output
AUX	Auxiliary input and output
CLOCK$	Date and time support
Varies (A – E)	One block device

These five logical devices can be implemented with a BIOS supporting a minimum of three physical devices: a keyboard and display, a timer or clock/calendar chip that can provide a hardware interrupt at regular intervals, and a block storage device. In such a minimum case, the printer and auxiliary device are simply aliases for the console device. However, most MS-DOS systems support several additional logical and physical devices. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output.

The MS-DOS kernel provides one additional device: the NUL device. NUL is a "bit bucket" — that is, anything written to NUL is simply discarded. Reading from NUL always returns an end-of-file marker. One common use for the NUL device is as the redirected output device of a command or application that is being run in a batch file; this redirection prevents screen clutter and disruption of the batch file's menus and displays.

The ROM BIOS

MS-DOS requires no ROM support (except that most bootstrap loaders reside in ROM) and does not care whether device-driver support resides in ROM or is part of the MS-DOS IO.SYS file loaded at initialization. PC-DOS, on the other hand, uses a very specific ROM BIOS. The PC ROM BIOS does not provide device drivers; rather, it provides support routines used by the device drivers found in IBMBIO.COM (the PC-DOS version of IO.SYS). The support provided by a PC ROM BIOS includes

- Power-on self test (POST)
- Bootstrap loader
- Keyboard
- Displays (monochrome and color/graphics adapters)
- Serial ports 1 and 2
- Parallel printer ports 1, 2, and 3
- Clock
- Print screen

The PC ROM BIOS loader routine searches the ROM space above the PC-DOS 640 KB limit for additional ROMs. The IBM fixed-disk adapter and enhanced graphics adapter (EGA) contain such ROMs. (The fixed-disk ROM also includes an additional loader routine that allows the system to start from the fixed disk.)

Summary

MS-DOS is a widely accepted traditional operating system. Its consistent and well-defined interface makes it one of the easier operating systems to adapt and program.

MS-DOS is also a growing operating system — each version has added more features yet made the system easier to use for both end-users and programmers. In addition, each version has included more support for different devices, from 5.25-inch floppy disks to high-density 3.5-inch floppy disks. As the hardware continues to evolve and user needs become more sophisticated, MS-DOS too will continue to evolve.

William Wong

Article 2
The Components of MS-DOS

MS-DOS is a modular operating system consisting of multiple components with special-ized functions. When MS-DOS is copied into memory during the loading process, many of its components are moved, adjusted, or discarded. However, when it is running, MS-DOS is a relatively static entity and its components are predictable and easy to study. Therefore, this article deals first with MS-DOS in its running state and later with its loading behavior.

The Major Elements

MS-DOS consists of three major modules:

Module	MS-DOS Filename	PC-DOS Filename
MS-DOS BIOS	IO.SYS	IBMBIO.COM
MS-DOS kernel	MSDOS.SYS	IBMDOS.COM
MS-DOS shell	COMMAND.COM	COMMAND.COM

During system initialization, these modules are loaded into memory, in the order given, just above the interrupt vector table located at the beginning of memory. All three modules remain in memory until the computer is reset or turned off. (The loader and system initial-ization modules are omitted from this list because they are discarded as soon as MS-DOS is running. *See* Loading MS-DOS below.)

The MS-DOS BIOS is supplied by the original equipment manufacturer (OEM) that distributes MS-DOS, usually for a particular computer. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: An Introduction to MS-DOS. The kernel is supplied by Microsoft and is the same across all OEMs for a particular version of MS-DOS — that is, no modifications are made by the OEM. The shell is a replaceable module that can be supplied by the OEM or replaced by the user; the default shell, COMMAND.COM, is supplied by Microsoft.

The MS-DOS BIOS

The file IO.SYS contains the MS-DOS BIOS and the MS-DOS initialization module, SYSINIT. The MS-DOS BIOS is customized for a particular machine by an OEM. SYSINIT is supplied by Microsoft and is put into IO.SYS by the OEM when the file is created. *See* Loading MS-DOS below.

The MS-DOS BIOS consists of a list of resident device drivers and an additional initialization module created by the OEM. The device drivers appear first in IO.SYS because they remain resident after IO.SYS is initialized; the MS-DOS BIOS initialization routine and SYSINIT are usually discarded after initialization.

The minimum set of resident device drivers is CON, PRN, AUX, CLOCK$, and the driver for one block device. The resident character-device drivers appear in the driver list before the resident block-device drivers; installable character-device drivers are placed ahead of the resident device drivers in the list; installable block-device drivers are placed after the resident device drivers in the list. This sequence allows installable character-device drivers to supersede resident drivers. The NUL device driver, which must be the first driver in the chain, is contained in the MS-DOS kernel.

Device driver code can be split between IO.SYS and ROM. For example, most MS-DOS systems and all PC-DOS-compatible systems have a ROM BIOS that contains primitive device support routines. These routines are generally used by resident and installable device drivers to augment routines contained in RAM. (Placing the entire driver in RAM makes the driver dependent on a particular hardware configuration; placing part of the driver in ROM allows the MS-DOS BIOS to be paired with a particular ROM interface that remains constant for many different hardware configurations.)

The IO.SYS file is an absolute program image and does not contain relocation information. The routines in IO.SYS assume that the CS register contains the segment at which the file is loaded. Thus, IO.SYS has the same 64 KB restriction as a .COM file. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: Programming for ms-dos: Structure of an Application Program. Larger IO.SYS files are possible, but all device driver headers must lie in the first 64 KB and the code must rely on its own segment arithmetic to access routines outside the first 64 KB.

The MS-DOS kernel

The MS-DOS kernel is the heart of MS-DOS and provides the functions found in a traditional operating system. It is contained in a single proprietary file, MSDOS.SYS, supplied by Microsoft Corporation. The kernel provides its support functions (referred to as system functions) to application programs in a hardware-independent manner and, in turn, is isolated from hardware characteristics by relying on the driver routines in the MS-DOS BIOS to perform physical input and output operations.

The MS-DOS kernel provides the following services through the use of device drivers:

●   File and directory management
●   Character device input and output
●   Time and date support

It also provides the following non-device-related functions:

●   Memory management
●   Task and environment management
●   Country-specific configuration

Programs access system functions using software interrupt (INT) instructions. MS-DOS reserves Interrupts 20H through 3FH for this purpose. The MS-DOS interrupts are

Interrupt	Name
20H	Terminate Program
21H	MS-DOS Function Calls
22H	Terminate Routine Address
23H	Control-C Handler Address
24H	Critical Error Handler Address
25H	Absolute Disk Read
26H	Absolute Disk Write
27H	Terminate and Stay Resident
28H–2EH	Reserved
2FH	Multiplex
30H–3FH	Reserved

Interrupt 21H is the main source of MS-DOS services. The Interrupt 21H functions are implemented by placing a function number in the AH register, placing any necessary parameters in other registers, and issuing an INT 21H instruction. (MS-DOS also supports a call instruction interface for CP/M compatibility. The function and parameter registers differ from the interrupt interface. The CP/M interface was provided in MS-DOS version 1.0 solely to assist in movement of CP/M-based applications to MS-DOS. New applications should use Interrupt 21H functions exclusively.)

MS-DOS version 2.0 introduced a mechanism to modify the operation of the MS-DOS BIOS and kernel: the CONFIG.SYS file. CONFIG.SYS is a text file containing command options that modify the size or configuration of internal MS-DOS tables and cause additional device drivers to be loaded. The file is read when MS-DOS is first loaded into memory. *See* USER COMMANDS: CONFIG.SYS.

The MS-DOS shell

The shell, or command interpreter, is the first program started by MS-DOS after the MS-DOS BIOS and kernel have been loaded and initialized. It provides the interface between the kernel and the user. The default MS-DOS shell, COMMAND.COM, is a command-oriented interface; other shells may be menu-driven or screen-oriented.

COMMAND.COM is a replaceable shell. A number of commercial products can be used as COMMAND.COM replacements, or a programmer can develop a customized shell. The new shell program is installed by renaming the program to COMMAND.COM or by using the SHELL command in CONFIG.SYS. The latter method is preferred because it allows initialization parameters to be passed to the shell program.

COMMAND.COM can execute a set of internal (built-in) commands, load and execute programs, or interpret batch files. Most of the internal commands support file and directory operations and manipulate the program environment segment maintained by COMMAND.COM. The programs executed by COMMAND.COM are .COM or .EXE files loaded from a block device. The batch (.BAT) files supported by COMMAND.COM provide a limited programming language and are therefore useful for performing small, frequently used series of MS-DOS commands. In particular, when it is first loaded by MS-DOS, COMMAND.COM searches for the batch file AUTOEXEC.BAT and interprets it, if found, before taking any other action. COMMAND.COM also provides default terminate, Control-C and critical error handlers whose addresses are stored in the vectors for Interrupts 22H, 23H, and 24H. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

COMMAND.COM's split personality

COMMAND.COM is a conventional .COM application with a slight twist. Ordinarily, a .COM program is loaded into a single memory segment. COMMAND.COM starts this way but then copies the nonresident portion of itself into high memory and keeps the resident portion in low memory. The memory above the resident portion is released to MS-DOS.

The effect of this split is not apparent until after an executed program has terminated and the resident portion of COMMAND.COM regains control of the system. The resident portion then computes a checksum on the area in high memory where the nonresident portion should be, to determine whether it has been overwritten. If the checksum matches a stored value, the nonresident portion is assumed to be intact; otherwise, a copy of the nonresident portion is reloaded from disk and COMMAND.COM continues its normal operation.

This "split personality" exists because MS-DOS was originally designed for systems with a limited amount of RAM. The nonresident portion of COMMAND.COM, which contains the built-in commands and batch-file-processing routines that are not essential to regaining control and reloading itself, is much larger than the resident portion, which is responsible for these tasks. Thus, permitting the nonresident portion to be overwritten frees additional RAM and allows larger application programs to be run.

Command execution

COMMAND.COM interprets commands by first checking to see if the specified command matches the name of an internal command. If so, it executes the command; otherwise, it searches for a .COM, .EXE, or .BAT file (in that order) with the specified name. If a .COM or .EXE program is found, COMMAND.COM uses the MS-DOS EXEC function (Interrupt 21H Function 4BH) to load and execute it; COMMAND.COM itself interprets .BAT files. If no file is found, the message *Bad command or file name* is displayed.

Although a command is usually simply a filename without the extension, MS-DOS versions 3.0 and later allow a command name to be preceded by a full pathname. If a path is not explicitly specified, the COMMAND.COM search mechanism uses the contents of the

PATH environment variable, which can contain a list of paths to be searched for commands. The search starts with the current directory and proceeds through the directories specified by PATH until a file is found or the list is exhausted. For example, the PATH specification

```
PATH C:\BIN;D:\BIN;E:\
```

causes COMMAND.COM to search the current directory, then C:\BIN, then D:\BIN, and finally the root directory of drive E. COMMAND.COM searches each directory for a matching .COM, .EXE, or .BAT file, in that order, before moving to the next directory.

MS-DOS environments

Version 2.0 introduced the concept of environments to MS-DOS. An environment is a paragraph-aligned memory segment containing a concatenated set of zero-terminated (ASCIIZ) variable-length strings of the form

variable=value

that provide such information as the current search path used by COMMAND.COM to find executable files, the location of COMMAND.COM itself, and the format of the user prompt. The end of the set of strings is marked by a null string — that is, a single zero byte. A specific environment is associated with each program in memory through a pointer contained at offset 2CH in the 256-byte program segment prefix (PSP). The maximum size of an environment is 32 KB; the default size is 160 bytes.

If a program uses the EXEC function to load and execute another program, the contents of the new program's environment are provided to MS-DOS by the initiating program — one of the parameters passed to the MS-DOS EXEC function is a pointer to the new program's environment. The default environment provided to the new program is a copy of the initiating program's environment.

A program that uses the EXEC function to load and execute another program will not itself have access to the new program's environment, because MS-DOS provides a pointer to this environment only to the new program. Any changes made to the new program's environment during program execution are invisible to the initiating program because a child program's environment is always discarded when the child program terminates.

The system's master environment is normally associated with the shell COMMAND.COM. COMMAND.COM creates this set of environment strings within itself from the contents of the CONFIG.SYS and AUTOEXEC.BAT files, using the SET, PATH, and PROMPT commands. *See* USER COMMANDS: AUTOEXEC.BAT; CONFIG.SYS. In MS-DOS version 3.2, the initial size of COMMAND.COM's environment can be controlled by loading COMMAND.COM with the /E parameter, using the SHELL directive in CONFIG.SYS. For example, placing the line

```
SHELL=COMMAND.COM /E:2048 /P
```

in CONFIG.SYS sets the initial size of COMMAND.COM's environment to 2 KB. (The /P option prevents COMMAND.COM from terminating, thus causing it to remain in memory until the system is turned off or restarted.)

The SET command is used to display or change the COMMAND.COM environment contents. SET with no parameters displays the list of all the environment strings in the environment. A typical listing might show the following settings:

```
COMSPEC=A:\COMMAND.COM
PATH=C:\;A:\;B:\
PROMPT=$p  $d  $t$_$n$g
TMP=C:\TEMP
```

The following is a dump of the environment segment containing the previous environment example:

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   43 4F 4D 53 50 45 43 3D-41 3A 5C 43 4F 4D 4D 41   COMSPEC=A:\COMMA
0010   4E 44 2E 43 4F 4D 00 50-41 54 48 3D 43 3A 5C 3B   ND.COM.PATH=C:\;
0020   41 3A 5C 3B 42 3A 5C 00-50 52 4F 4D 50 54 3D 24   A:\;B:\.PROMPT=$
0030   70 20 20 24 64 20 20 24-74 24 5F 24 6E 24 67 00   p  $d  $t$_$n$g.
0040   54 4D 50 3D 43 3A 5C 54-45 4D 50 00 00 00 00 00   TMP=C:\TEMP.....
```

A SET command that specifies a variable but does not specify a value for it deletes the variable from the environment.

A program can ignore the contents of its environment; however, use of the environment can add a great deal to the flexibility and configurability of batch files and application programs.

Batch files

Batch files are text files with a .BAT extension that contain MS-DOS user and batch commands. Each line in the file is limited to 128 bytes. *See* USER COMMANDS: batch. Batch files can be created using most text editors, including EDLIN, and short batch files can even be created using the COPY command:

```
C>COPY CON SAMPLE.BAT  <Enter>
```

The CON device is the system console; text entered from the keyboard is echoed on the screen as it is typed. The copy operation is terminated by pressing Ctrl-Z (or the F6 key on IBM-compatible machines), followed by the Enter key.

Batch files are interpreted by COMMAND.COM one line at a time. In addition to the standard MS-DOS commands, COMMAND.COM's batch-file interpreter supports a number of special batch commands:

Command	Meaning
ECHO *	Display a message.
FOR *	Execute a command for a list of files.

(more)

Command	Meaning
GOTO*	Transfer control to another point.
IF*	Conditionally execute a command.
PAUSE	Wait for any key to be pressed.
REM	Insert comment line.
SHIFT*	Access more than 10 parameters.

* MS-DOS versions 2.0 and later

Execution of a batch file can be terminated before completion by pressing Ctrl-C or Ctrl-Break, causing COMMAND.COM to display the prompt

```
Terminate batch job? (Y/N)
```

I/O redirection

I/O redirection was introduced with MS-DOS version 2.0. The redirection facility is implemented within COMMAND.COM using the Interrupt 21H system functions Duplicate File Handle (45H) and Force Duplicate File Handle (46H). COMMAND.COM uses these functions to provide both redirection at the command level and a UNIX/XENIX-like pipe facility.

Redirection is transparent to application programs, but to take advantage of redirection, an application program must make use of the standard input and output file handles. The input and output of application programs that directly access the screen or keyboard or use ROM BIOS functions cannot be redirected.

Redirection is specified in the command line by prefixing file or device names with the special characters >, >>, and <. Standard output (default = CON) is redirected using > and >> followed by the name of a file or character device. The former character creates a new file (or overwrites an existing file with the same name); the latter appends text to an existing file (or creates the file if it does not exist). Standard input (default = CON) is redirected with the < character followed by the name of a file or character device. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: Customizing ms-dos: Writing MS-DOS Filters.

The redirection facility can also be used to pass information from one program to another through a "pipe." A pipe in MS-DOS is a special file created by COMMAND.COM. COMMAND.COM redirects the output of one program into this file and then redirects this file as the input to the next program. The pipe symbol, a vertical bar (|), separates the program names. Multiple program names can be piped together in the same command line:

```
C>DIR *.* | SORT | MORE  <Enter>
```

This command is equivalent to

```
C>DIR *.* > PIPE0  <Enter>
C>SORT < PIPE0 > PIPE1  <Enter>
C>MORE < PIPE1  <Enter>
```

The concept of pipes came from UNIX/XENIX, but UNIX/XENIX is a multitasking operating system that actually runs the programs simultaneously. UNIX/XENIX uses memory buffers to connect the programs, whereas MS-DOS loads one program at a time and passes information through a disk file.

Loading MS-DOS

Getting MS-DOS up to the standard A> prompt is a complex process with a number of variations. This section discusses the complete process normally associated with MS-DOS versions 2.0 and later. (MS-DOS versions 1.x use the same general steps but lack support for various system tables and installable device drivers.)

MS-DOS is loaded as a result of either a "cold boot" or a "warm boot." On IBM-compatible machines, a cold boot is performed when the computer is first turned on or when a hardware reset occurs. A cold boot usually performs a power-on self test (POST) and determines the amount of memory available, as well as which peripheral adapters are installed. The POST is ordinarily reserved for a cold boot because it takes a noticeable amount of time. For example, an IBM-compatible ROM BIOS tests all conventional and extended RAM (RAM above 1 MB on an 80286-based or 80386-based machine), a procedure that can take tens of seconds. A warm boot, initiated by simultaneously pressing the Ctrl, Alt, and Del keys, bypasses these hardware checks and begins by checking for a bootable disk.

A bootable disk normally contains a small loader program that loads MS-DOS from the same disk. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices. The body of MS-DOS is contained in two files: IO.SYS and MSDOS.SYS (IBMBIO.COM and IBMDOS.COM with PC-DOS). IO.SYS contains the Microsoft system initialization module, SYSINIT, which configures MS-DOS using either default values or the specifications in the CONFIG.SYS file, if one exists, and then starts up the shell program (usually COMMAND.COM, the default). COMMAND.COM checks for an AUTOEXEC.BAT file and interprets the file if found. (Other shells might not support such batch files.) Finally, COMMAND.COM prompts the user for a command. (The standard MS-DOS prompt is A> if the system was booted from a floppy disk and C> if the system was booted from a fixed disk.) Each of these steps is discussed in detail below.

The ROM BIOS, POST, and bootstrapping

All 8086/8088-compatible microprocessors begin execution with the CS:IP set to FFFF:0000H, which typically contains a jump instruction to a destination in the ROM BIOS that contains the initialization code for the machine. (This has nothing to do with MS-DOS; it is a feature of the Intel microprocessors.) On IBM-compatible machines, the ROM BIOS occupies the address space from F000:0000H to this jump instruction. Figure 2-1 shows the location of the ROM BIOS within the 1 MB address space. Supplementary ROM support can be placed before (at lower addresses than) the ROM BIOS.

All interrupts are disabled when the microprocessor starts execution and it is up to the initialization routine to set up the interrupt vectors at the base of memory.

```
┌──────────────────┐ ◄── FFFF:000FH(1 MB)
│    ROM BIOS      │ ◄── FFFF:0000H
├──────────────────┤ ◄── F000:0000H
│                  │
│ Other ROM and RAM│
│                  │
├──────────────────┤ ◄── Top of RAM
│                  │      (A000:0000H for IBM PC)
│                  │
│                  │
│    Free RAM      │
│                  │
│                  │
│                  │
│                  │
└──────────────────┘ ◄── 0000:0000H
```
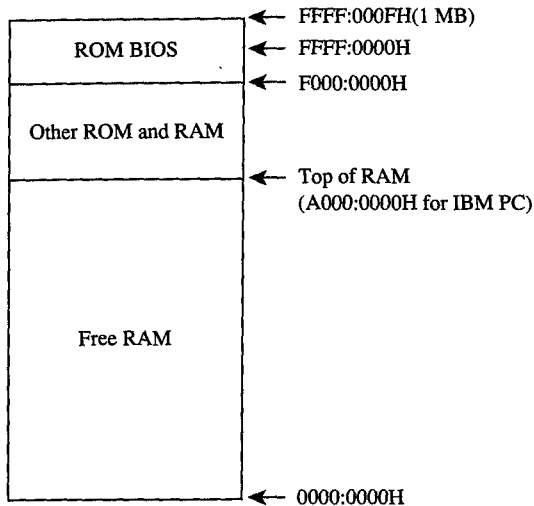
Figure 2-1. Memory layout at startup.

The initialization routine in the ROM BIOS — the POST procedure — typically deter-
mines what devices are installed and operational and checks conventional memory (the
first 1 MB) and, for 80286-based or 80386-based machines, extended memory (above 1
MB). The devices are tested, where possible, and any problems are reported using a series
of beeps and display messages on the screen.

When the machine is found to be operational, the ROM BIOS sets it up for normal opera-
tion. First, it initializes the interrupt vector table at the beginning of memory and any inter-
rupt controllers that reference the table. The interrupt vector table area is located from
0000:0000H to 0000:03FFH. On IBM-compatible machines, some of the subsequent mem-
ory (starting at address 0000:0400H) is used for table storage by various ROM BIOS rou-
tines (Figure 2-2). The beginning load address for the MS-DOS system files is usually in
the range 0000:0600H to 0000:0800H.

Next, the ROM BIOS sets up any necessary hardware interfaces, such as direct memory
access (DMA) controllers, serial ports, and the like. Some hardware setup may be done
before the interrupt vector table area is set up. For example, the IBM PC DMA controller
also provides refresh for the dynamic RAM chips and RAM cannot be used until the
refresh DMA is running; therefore, the DMA must be set up first.

Some ROM BIOS implementations also check to see if additional ROM BIOSs are installed
by scanning the memory from A000:0000H to F000:0000H for a particular sequence of sig-
nature bytes. If additional ROM BIOSs are found, their initialization routines are called to
initialize the associated devices. Examples of additional ROMs for the IBM PC family are
the PC/XT's fixed-disk ROM BIOS and the EGA ROM BIOS.

The ROM BIOS now starts the bootstrap procedure by executing the ROM loader routine.
On the IBM PC, this routine checks the first floppy-disk drive to see if there is a bootable

```
┌─────────────────────────┐  ◄── FFFF:000FH(1 MB)
│        ROM BIOS         │  ◄── FFFF:0000H
├─────────────────────────┤  ◄── F000:0000H
│                         │
│    Other ROM and RAM    │
│                         │  ◄── Top of RAM
├─────────────────────────┤      (A000:0000H for IBM PC)
│                         │
│                         │
│        Free RAM         │
│                         │
│                         │
├─────────────────────────┤  ◄── 0000:0600H
│     ROM BIOS tables     │  ◄── 0000:0400H
├─────────────────────────┤
│    Interrupt vectors    │
└─────────────────────────┘  ◄── 0000:0000H
```
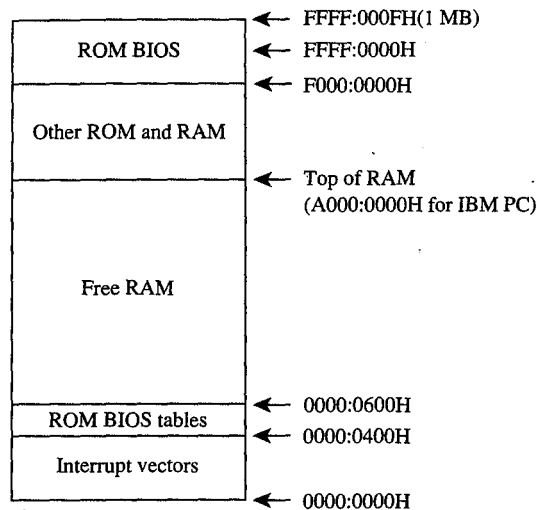
Figure 2-2. The interrupt vector table and the ROM BIOS table.

disk in it. If there is not, the routine then invokes the ROM associated with another bootable device to see if that device contains a bootable disk. This procedure is repeated until a bootable disk is found or until all bootable devices have been checked without success, in which case ROM BASIC is enabled.

Bootable devices can be detected by a number of proprietary means. The IBM PC ROM BIOS reads the first sector on the disk into RAM (Figure 2-3) and checks for an 8086-family short or long jump at the beginning of the sector and for AA55H in the last word of the sector. This signature indicates that the sector contains the operating-system loader. Data disks — those disks not set up with the MS-DOS system files — usually cause the ROM loader routine to display a message indicating that the disk is not a bootable system disk. The customary recovery procedure is to display a message asking the user to insert another disk (with the operating system files on it) and press a key to try the load operation again. The ROM loader routine is then typically reexecuted from the beginning so that it can repeat its normal search procedure.

When it finds a bootable device, the ROM loader routine loads the operating-system loader and transfers control to it. The operating-system loader then uses the ROM BIOS services through the interrupt table to load the next part of the operating system into low memory.

Before it can proceed, the operating-system loader must know something about the configuration of the system boot disk (Figure 2-4). MS-DOS-compatible disks contain a data structure that contains this information. This structure, known as the BIOS parameter block (BPB), is located in the same sector as the operating-system loader. From the contents of the BPB, the operating-system loader calculates the location of the root directory
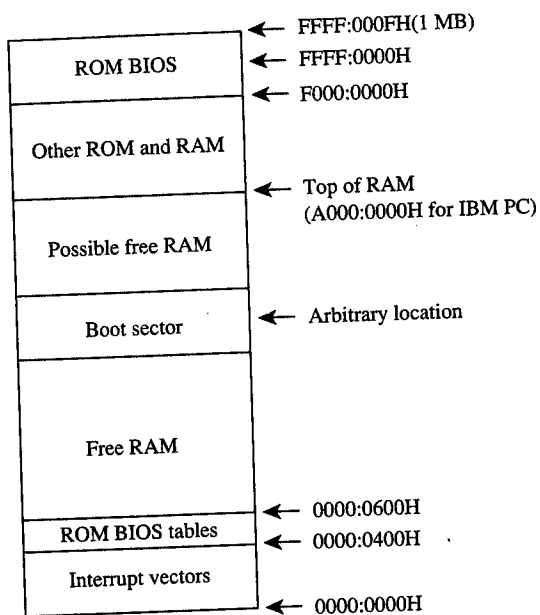
Figure 2-3. A loaded boot sector.
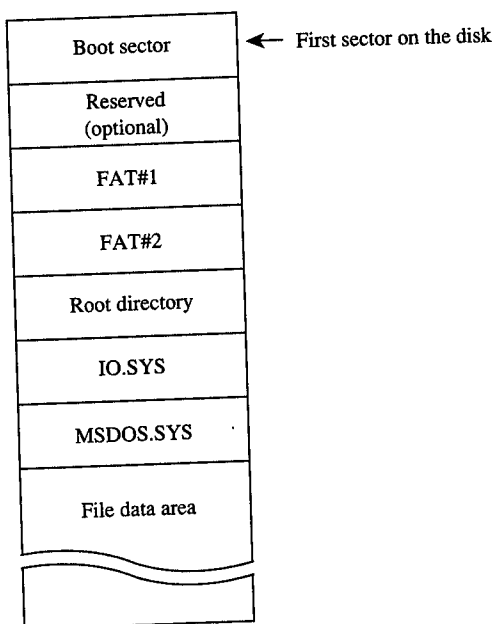
Figure 2-4. Boot-disk configuration.

for the boot disk so that it can verify that the first two entries in the root directory are IO.SYS and MSDOS.SYS. For versions of MS-DOS through 3.2, these files must also be the first two files in the file data area, and they must be contiguous. (The operating-system loader usually does not check the file allocation table [FAT] to see if IO.SYS and MSDOS.SYS are actually stored in contiguous sectors.) *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

Next, the operating-system loader reads the sectors containing IO.SYS and MSDOS.SYS into contiguous areas of memory just above the ROM BIOS tables (Figure 2-5). (An alternative method is to take advantage of the operating-system loader's final jump to the entry point in IO.SYS and include routines in IO.SYS that allow it to load MSDOS.SYS.)

Finally, assuming the file was loaded without any errors, the operating-system loader transfers control to IO.SYS, passing the identity of the boot device. The operating-system loader is no longer needed and its RAM is made available for other purposes.

ROM BIOS	← FFFF:000FH(1 MB)
	← F000:0000H
Other ROM and RAM	
	← Top of RAM (A000:0000H for IBM PC)
Possible free RAM	
Boot sector	← Arbitrary location
Free RAM	
MSDOS.SYS	
IO.SYS	← SYSINIT ← MS-DOS BIOS (resident device drivers) ← 0000:0600H
ROM BIOS tables	← 0000:0400H
Interrupt vectors	
	← 0000:0000H
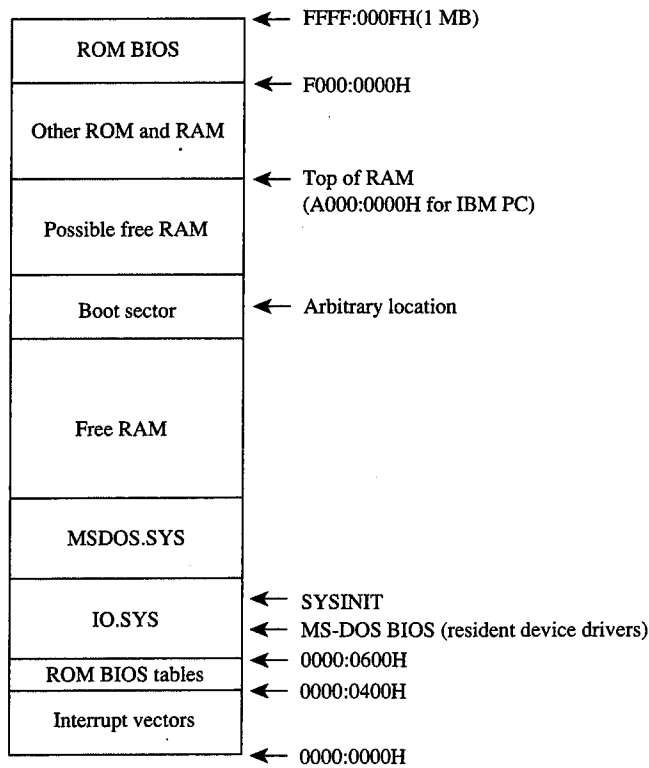
Figure 2-5. IO.SYS and MSDOS.SYS loaded.

MS-DOS system initialization (SYSINIT)

MS-DOS system initialization begins after the operating-system loader has loaded IO.SYS and MSDOS.SYS and transferred control to the beginning of IO.SYS. To this point, there has been no standard loading procedure imposed by MS-DOS, although the IBM PC loading procedure outlined here has become the de facto standard for most MS-DOS machines. When control is transferred to IO.SYS, however, MS-DOS imposes its standards.

The IO.SYS file is divided into three modules:

- The resident device drivers
- The basic MS-DOS BIOS initialization module
- The MS-DOS system initialization module, SYSINIT

The two initialization modules are usually discarded as soon as MS-DOS is completely initialized and the shell program is running; the resident device drivers remain in memory while MS-DOS is running and are therefore placed in the first part of the IO.SYS file, before the initialization modules.

The MS-DOS BIOS initialization module ordinarily displays a sign-on message and the copyright notice for the OEM that created IO.SYS. On IBM-compatible machines, it then examines entries in the interrupt table to determine what devices were found by the ROM BIOS at POST time and adjusts the list of resident device drivers accordingly. This adjustment usually entails removing those drivers that have no corresponding installed hardware. The initialization routine may also modify internal tables within the device drivers. The device driver initialization routines will be called later by SYSINIT, so the MS-DOS BIOS initialization routine is now essentially finished and control is transferred to the SYSINIT module.

SYSINIT locates the top of RAM and copies itself there. It then transfers control to the copy and the copy proceeds with system initialization. The first step is to move MSDOS.SYS, which contains the MS-DOS kernel, to a position immediately following the end of the resident portion of IO.SYS, which contains the resident device drivers. This move overwrites the original copy of SYSINIT and usually all of the MS-DOS BIOS initialization routine, which are no longer needed. The resulting memory layout is shown in Figure 2-6.

SYSINIT then calls the initialization routine in the newly relocated MS-DOS kernel. This routine performs the internal setup for the kernel, including putting the appropriate values into the vectors for Interrupts 20H through 3FH.

The MS-DOS kernel initialization routine then calls the initialization function of each resident device driver to set up vectors for any external hardware interrupts used by the device. Each block-device driver returns a pointer to a BPB for each drive that it supports; these BPBs are inspected by SYSINIT to find the largest sector size used by any of the drivers. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices. The kernel initialization routine then allocates a sector buffer the size of the largest sector found and places the NUL device driver at the head of the device driver list.
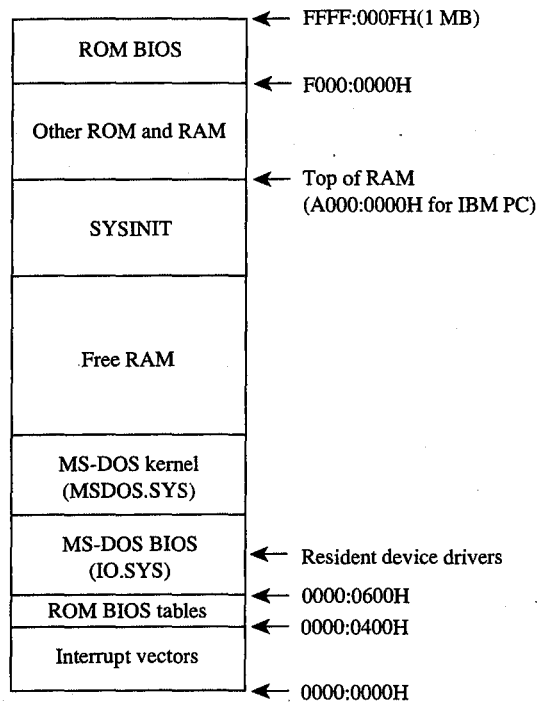
Figure 2-6. SYSINIT and MSDOS.SYS relocated.

The kernel initialization routine's final operation before returning to SYSINIT is to display the MS-DOS copyright message. The loading of the system portion of MS-DOS is now complete and SYSINIT can use any MS-DOS function in conjunction with the resident set of device drivers.

SYSINIT next attempts to open the CONFIG.SYS file in the root directory of the boot drive. If the file does not exist, SYSINIT uses the default system parameters; if the file is opened, SYSINIT reads the entire file into high memory and converts all characters to uppercase. The file contents are then processed to determine such settings as the number of disk buffers, the number of entries in the file tables, and the number of entries in the drive translation table (depending on the specific commands in the file), and these structures are allocated following the MS-DOS kernel (Figure 2-7).

Then SYSINIT processes the CONFIG.SYS text sequentially to determine what installable device drivers are to be implemented and loads the installable device driver files into memory after the system disk buffers and the file and drive tables. Installable device driver files can be located in any directory on any drive whose driver has already been loaded. Each installable device driver initialization function is called after the device driver file is loaded into memory. The initialization procedure is the same as for resident device drivers, except that SYSINIT uses an address returned by the device driver itself to determine where the next device driver is to be placed. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

```
                        ◄─── FFFF:000FH(1 MB)
┌──────────────────┐
│    ROM BIOS      │
│                  │    ◄─── F000:0000H
├──────────────────┤
│                  │
│Other ROM and RAM │
│                  │    ◄─── Top of RAM
├──────────────────┤         (A000:0000H for IBM PC)
│                  │
│     SYSINIT      │
│                  │
├──────────────────┤
│                  │
│                  │
│    Free RAM      │
│                  │
│                  │
│                  │
├──────────────────┤
│   Installable    │
│  device drivers  │
├──────────────────┤
│                  │
│File control blocks│
│                  │
├──────────────────┤
│                  │
│   Disk buffers   │
│                  │
├──────────────────┤
│                  │
│  MS-DOS tables   │
│                  │
├──────────────────┤
│  MS-DOS kernel   │
│   (MSDOS.SYS)    │
├──────────────────┤
│   MS-DOS BIOS    │    ◄─── Resident device drivers
│    (IO.SYS)      │
├──────────────────┤    ◄─── 0000:0600H
│  ROM BIOS tables │    ◄─── 0000:0400H
├──────────────────┤
│ Interrupt vectors│
└──────────────────┘    ◄─── 0000:0000H
```
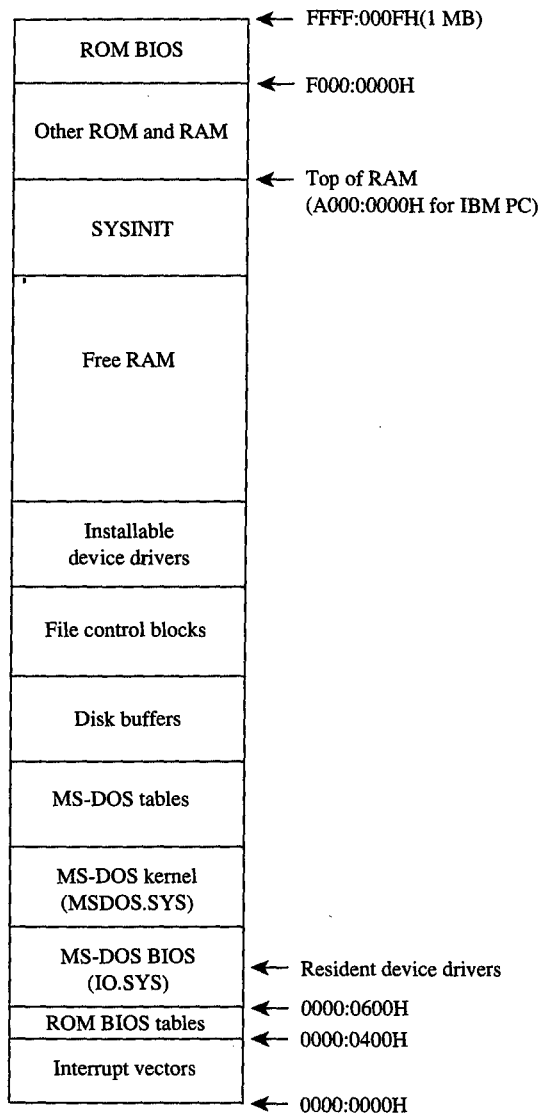
Figure 2-7. Tables allocated and installable device drivers loaded.

Like resident device drivers, installable device drivers can be discarded by SYSINIT if the device driver initialization routine determines that a device is inoperative or nonexistent. A discarded device driver is not included in the list of device drivers. Installable character-device drivers supersede resident character-device drivers with the same name; installable block-device drivers cannot supersede resident block-drivers and are assigned drive letters *following* those of the resident block-device drivers.

SYSINIT now closes all open files and then opens the three character devices CON, PRN, and AUX. The console (CON) is used as standard input, standard output, and standard error; the standard printer port is PRN (which defaults to LPT1); the standard auxiliary port is AUX (which defaults to COM1). Installable device drivers with these names will replace any resident versions.

Starting the shell

SYSINIT's last function is to load and execute the shell program by using the MS-DOS EXEC function. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: The MS-DOS EXEC Function. The SHELL statement in CONFIG.SYS specifies both the name of the shell program and its initial parameters; the default MS-DOS shell is COMMAND.COM. The shell program is loaded at the start of free memory after the installable device drivers or after the last internal MS-DOS file control block if there are no installable device drivers (Figure 2-8).

COMMAND.COM

COMMAND.COM consists of three parts:

- A resident portion
- An initialization module
- A transient portion

The resident portion contains support for termination of programs started by COMMAND.COM and presents critical-error messages. It is also responsible for re-loading the transient portion when necessary.

The initialization module is called once by the resident portion. First, it moves the transient portion to high memory. (Compare Figures 2-8 and 2-9.) Then it processes the parameters specified in the SHELL command in the CONFIG.SYS file, if any. *See* USER COMMANDS: COMMAND. Next, it processes the AUTOEXEC.BAT file, if one exists, and finally, it transfers control back to the resident portion, which frees the space used by the initialization module and transient portion. The relocated transient portion then displays the MS-DOS user prompt and is ready to accept commands.

The transient portion gets a command from either the console or a batch file and executes it. Commands are divided into three categories:

- Internal commands
- Batch files
- External commands

Internal commands are routines contained within COMMAND.COM and include operations like COPY or ERASE. Execution of an internal command does not overwrite the transient portion. Internal commands consist of a keyword, sometimes followed by a list of command-specific parameters.

ROM BIOS	← FFFF:000FH(1 MB)
	← F000:0000H
Other ROM and RAM	
	← Top of RAM (A000:0000H for IBM PC)
SYSINIT	
Free RAM	
COMMAND.COM (transient)	
COMMAND.COM (initialization)	
COMMAND.COM (resident)	
Installable device drivers	
File control blocks	
Disk buffers	
MS-DOS tables	
MS-DOS kernel (MSDOS.SYS)	
MS-DOS BIOS (IO.SYS)	← Resident device drivers
ROM BIOS tables	← 0000:0600H ← 0000:0400H
Interrupt vectors	← 0000:0000H

Figure 2-8. COMMAND.COM loaded.

```
                              ◄─── FFFF:000FH(1 MB)
┌─────────────────────┐
│      ROM BIOS       │
│                     │      ◄─── F000:0000H
├─────────────────────┤
│  Other ROM and RAM  │
│                     │      ◄─── Top of RAM
├─────────────────────┤           (A000:0000H for IBM PC)
│   COMMAND.COM       │
│   (transient)       │
├─────────────────────┤
│                     │
│      Free RAM       │
│                     │
│                     │
├─────────────────────┤
│   COMMAND.COM       │
│   (resident)        │
├─────────────────────┤
│    Installable      │
│   device drivers    │
├─────────────────────┤
│  File control blocks│
│                     │
├─────────────────────┤
│    Disk buffers     │
│                     │
├─────────────────────┤
│   MS-DOS tables     │
│                     │
├─────────────────────┤
│   MS-DOS kernel     │
│   (MSDOS.SYS)       │
├─────────────────────┤
│   MS-DOS BIOS       │      ◄─── Resident device drivers
│   (IO.SYS)          │
├─────────────────────┤      ◄─── 0000:0600H
│  ROM BIOS tables    │      ◄─── 0000:0400H
├─────────────────────┤
│  Interrupt vectors  │
└─────────────────────┘      ◄─── 0000:0000H
```
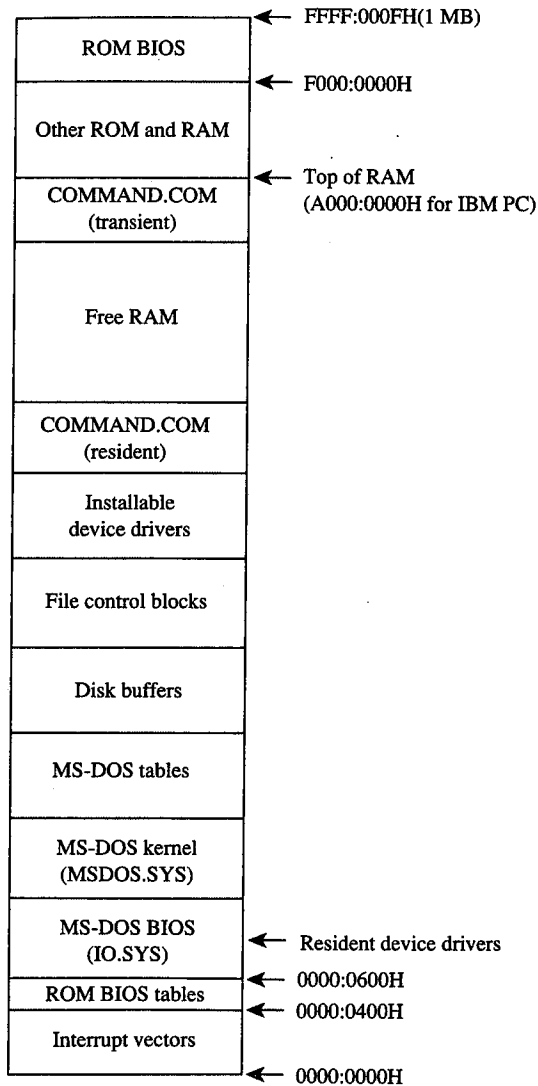
Figure 2-9. COMMAND.COM after relocation.

Batch files are text files that contain internal commands, external commands, batch-file directives, and nonexecutable comments. *See* USER COMMANDS: BATCH.

External commands, which are actually executable programs, are stored in separate files with .COM and .EXE extensions and are included on the MS-DOS distribution disks. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. These programs are invoked with the name of the file without the extension. (MS-DOS versions 3.x allow the complete pathname of the external command to be specified.)

External commands are loaded by COMMAND.COM by means of the MS-DOS EXEC function. The EXEC function loads a program into the free memory area, also called the transient program area (TPA), and then passes it control. Control returns to COMMAND.COM when the new program terminates. Memory used by the program is released unless it is a terminate-and-stay-resident (TSR) program, in which case some of the memory is retained for the resident portion of the program. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities.

After a program terminates, the resident portion of COMMAND.COM checks to see if the transient portion is still valid, because if the program was large, it may have overwritten the transient portion's memory space. The validity check is done by computing a checksum on the transient portion and comparing it with a stored value. If the checksums do not match, the resident portion loads a new copy of the transient portion from the COMMAND.COM file.

Just as COMMAND.COM uses the EXEC function to load and execute a program, programs can load and execute other programs until the system runs out of memory. Figure 2-10 shows a typical memory configuration for multiple applications loaded at the same time. The active task — the last one executed — ordinarily has complete control over the system, with the exception of the hardware interrupt handlers, which gain control whenever a hardware interrupt needs to be serviced.

MS-DOS is not a multitasking operating system, so although several programs can be resident in memory, only one program can be active at a time. The stack-like nature of the system is apparent in Figure 2-10. The top program is the active one; the next program down will continue to run when the top program exits, and so on until control returns to COMMAND.COM. RAM-resident programs that remain in memory after they have terminated are the exception. In this case, a program lower in memory than another program can become the active program, although the one-active-process limit is still in effect.

A custom shell program

The SHELL directive in the CONFIG.SYS file can be used to replace the system's default shell, COMMAND.COM, with a custom shell. Nearly any program can be used as a system shell as long as it supplies default handlers for the Control-C and critical error exceptions. For example, the program in Figure 2-11 can be used to make any application program appear to be a shell program — if the application program terminates, SHELL.COM restarts it, giving the appearance that the application program is the shell program.

SHELL.COM sets up the segment registers for operation as a .COM file and reduces the program segment size to less than 1 KB. It then initializes the segment values in the parameter table for the EXEC function, because .COM files cannot set up segment values within a program. The Control-C and critical error interrupt handler vectors are set to the address of the main program loop, which tries to load the new shell program. SHELL.COM prints a message if the EXEC operation fails. The loop continues forever and SHELL.COM will never return to the now-discarded SYSINIT that started it.

```
                              ◄── FFFF:000FH(1 MB)
┌─────────────────┐
│    ROM BIOS     │
├─────────────────┤          ◄── F000:0000H
│                 │
│ Other ROM and RAM│
│                 │          ◄── Top of RAM
├─────────────────┤              (A000:0000H for IBM PC)
│  COMMAND.COM    │
│   (transient)   │
├─────────────────┤
│                 │
│    Free RAM     │
│                 │
├─────────────────┤
│   Program #3    │
│    (active)     │
├─────────────────┤
│                 │
│   Program #2    │
│                 │
├─────────────────┤
│                 │
│   Program #1    │
│                 │
├─────────────────┤
│  COMMAND.COM    │
│   (resident)    │
├─────────────────┤
│   Installable   │
│ device drivers  │
├─────────────────┤
│                 │
│File control blocks│
│                 │
├─────────────────┤
│                 │
│  Disk buffers   │
│                 │
├─────────────────┤
│                 │
│  MS-DOS tables  │
│                 │
├─────────────────┤
│                 │
│  MS-DOS kernel  │
│  (MSDOS.SYS)    │
├─────────────────┤
│  MS-DOS BIOS    │          ◄── Resident device drivers
│   (IO.SYS)      │
├─────────────────┤          ◄── 0000:0600H
│ ROM BIOS tables │          ◄── 0000:0400H
├─────────────────┤
│ Interrupt vectors│
└─────────────────┘          ◄── 0000:0000H
```

Figure 2-10. Multiple programs loaded.

```
; SHELL.ASM   A simple program to run an application as an
;             MS-DOS shell program. The program name and
;             startup parameters must be adjusted before
;             SHELL is assembled.
;
; Written by William Wong
;
; To create SHELL.COM:
;
;             C>MASM SHELL;
;             C>LINK SHELL;
;             C>EXE2BIN SHELL.EXE SHELL.COM


stderr  equ 2                  ; standard error
cr      equ 0dh                ; ASCII carriage return
lf      equ 0ah                ; ASCII linefeed
cseg    segment para public 'CODE'
;
; --  Set up DS, ES, and SS:SP to run as .COM  --
;
        assume  cs:cseg
start   proc    far
        mov     ax,cs          ; set up segment registers
        add     ax,10h         ; AX = segment after PSP
        mov     ds,ax
        mov     ss,ax          ; set up stack pointer
        mov     sp,offset stk
        mov     ax,offset shell
        push    cs             ; push original CS
        push    ds             ; push segment of shell
        push    ax             ; push offset of shell
        ret                    ; jump to shell
start   endp
;
; --  Main program running as .COM  --
;
; CS, DS, SS = cseg
; Original CS value on top of stack
;
        assume  cs:cseg,ds:cseg,ss:cseg
seg_size equ (((offset last) - (offset start)) + 10fh)/16
shell   proc    near
        pop     es             ; ES = segment to shrink
        mov     bx,seg_size    ; BX = new segment size
        mov     ah,4ah         ; AH = modify memory block
        int     21h            ; free excess memory
        mov     cmd_seg,ds     ; setup segments in
        mov     fcb1_seg,ds    ; parameter block for EXEC
        mov     fcb2_seg,ds
        mov     dx,offset main_loop
        mov     ax,2523h       ; AX = set Control-C handler
```

Figure 2-11. A simple program to run an application as an MS-DOS shell.   *(more)*

```
           int     21h              ; set handler to DS:DX
           mov     dx,offset main_loop
           mov     ax,2524h         ; AX = set critical error handler
           int     21h              ; set handler to DS:DX
                                    ; Note: DS is equal to CS
main_loop:
           push    ds               ; save segment registers
           push    es
           mov     cs:stk_seg,ss    ; save stack pointer
           mov     cs:stk_off,sp
           mov     dx,offset pgm_name
           mov     bx,offset par_blk
           mov     ax,4b00h         ; AX = EXEC/run program
           int     21h              ; carry = EXEC failed
           mov     ss,cs:stk_seg    ; restore stack pointer
           mov     sp,cs:stk_off
           pop     es               ; restore segment registers
           pop     ds
           jnc     main_loop        ; loop if program run
           mov     dx,offset load_msg
           mov     cx,load_msg_length
           call    print            ; display error message
           mov     ah,08h           ; AH = read without echo
           int     21h              ; wait for any character
           jmp     main_loop        ; execute forever
shell   endp
;
; --  Print string  --
;
; DS:DX = address of string
; CX    = size
;
print   proc    near
           mov     ah,40h           ; AH = write to file
           mov     bx,stderr        ; BX = file handle
           int     21h              ; print string
           ret
print   endp
;
; --  Message strings  --
;
load_msg db cr,lf
         db 'Cannot load program.',cr,lf
         db 'Press any key to try again.',cr,lf
load_msg_length equ $-load_msg
;
; --  Program data area  --
;
stk_seg  dw      0                ; stack segment pointer
stk_off  dw      0                ; save area during EXEC
pgm_name db      '\NEWSHELL.COM',0 ; any program will do
```

Figure 2-11. Continued.   *(more)*

```
par_blk  dw    0                      ; use current environment
         dw    offset cmd_line ; command-line address
cmd_seg  dw    0                      ; fill in at initialization
         dw    offset fcb1     ; default FCB #1
fcb1_seg dw    0                      ; fill in at initialization
         dw    offset fcb2     ; default FCB #2
fcb2_seg dw    0                      ; fill in at initialization
cmd_line db    0,cr                   ; actual command line
fcb1     db    0
         db    11 dup (' ')
         db    25 dup ( 0 )
fcb2     db    0
         db    11 dup (' ')
         db    25 dup ( 0 )
         dw    200 dup ( 0 )   ; program stack area
stk      dw    0
last     equ   $                      ; last address used
cseg     ends
         end   start
```

Figure 2-11. Continued.

SHELL.COM is very short and not too smart. It needs to be changed and rebuilt if the name
of the application program changes. A simple extension to SHELL — call it XSHELL —
would be to place the name of the application program and any parameters in the com-
mand line. XSHELL would then have to parse the program name and the contents of the
two FCBs needed for the EXEC function. The CONFIG.SYS line for starting this shell
would be

```
SHELL=XSHELL \SHELL\DEMO.EXE PARAM1 PARAM2 PARAM3
```

SHELL.COM does not set up a new environment but simply uses the one passed to it.

William Wong

Article 3
MS-DOS Storage Devices

Application programs access data on MS-DOS storage devices through the MS-DOS file-system support that is part of the MS-DOS kernel. The MS-DOS kernel accesses these storage devices, also called block devices, through two types of device drivers: resident block-device drivers contained in IO.SYS and installable block-device drivers loaded from individual files when MS-DOS is loaded. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: The Components of MS-DOS; CUSTOMIZING MS-DOS: Installable Device Drivers.

MS-DOS can handle almost any medium, recording method, or other variation for a storage device as long as there is a device driver for it. MS-DOS needs to know only the sector size and the maximum number of sectors for the device; the appropriate translation between logical sector number and physical location is made by the device driver. Information about the number of heads, tracks, and so on is required only for those partitioning programs that allocate logical devices along these boundaries. *See* Layout of a Partition below.

The floppy-disk drive is perhaps the best-known block device, followed by its faster cousin, the fixed-disk drive. Other MS-DOS media include RAMdisks, nonvolatile RAMdisks, removable hard disks, tape drives, and CD ROM drives. With the proper device driver, MS-DOS can place a file system on any of these devices (except read-only media such as CD ROM).

This article discusses the structure of the file system on floppy and fixed disks, starting with the physical layout of a disk and then moving on to the logical layout of the file system. The scheme examined is for the IBM PC fixed disk.

Structure of an MS-DOS Disk

The structure of an MS-DOS disk can be viewed in a number of ways:

● Physical device layout
● Logical device layout
● Logical block layout
● MS-DOS file system

The physical layout of a disk is expressed in terms of sectors, tracks, and heads. The logical device layout, also expressed in terms of sectors, tracks, and heads, indicates how a logical device maps onto a physical device. A partitioned physical device contains multiple logical devices; a physical device that cannot be partitioned contains only one. Each logical device

has a logical block layout used by MS-DOS to implement a file system. These various views of an MS-DOS disk are discussed below. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management; Disk Directories and Volume Labels.

Layout of a physical block device

The two major block-device implementations are solid-state RAMdisks and rotating magnetic media such as floppy or fixed disks. Both implementations provide a fixed amount of storage in a fixed number of randomly accessible same-size sectors.

RAMdisks

A RAMdisk is a block device that has sectors mapped sequentially into RAM. Thus, the RAMdisk is viewed as a large set of sequentially numbered sectors whose addresses are computed by simply multiplying the sector number by the sector size and adding the base address of the RAMdisk sector buffer. Access is fast and efficient and the access time to any sector is fixed, making the RAMdisk the fastest block device available. However, there are significant drawbacks to RAMdisks. First, they are volatile; their contents are irretrievably lost when the computer's power is turned off (although a special implementation of the RAMdisk known as a nonvolatile RAMdisk includes a battery backup system that ensures that its contents are not lost when the computer's power is turned off). Second, they are usually not portable.

Physical disks

Floppy-disk and fixed-disk systems, on the other hand, store information on revolving platters coated with a special magnetic material. The disk is rotated in the drive at high speeds — approximately 300 revolutions per minute (rpm) for floppy disks and 3600 rpm for fixed disks. (The term "fixed" refers to the fact that the medium is built permanently into the drive, not to the motion of the medium.) Fixed disks are also referred to as "hard" disks, because the disk itself is usually made from a rigid material such as metal or glass; floppy disks are usually made from a flexible material such as plastic.

A transducer element called the read/write head is used to read and write tiny magnetic regions on the rotating magnetic medium. The regions act like small bar magnets with north and south poles. The magnetic regions of the medium can be logically oriented toward one or the other of these poles — orientation toward one pole is interpreted as a specific binary state (1 or 0) and orientation toward the other pole is interpreted as the opposite binary state. A change in the direction of orientation (and hence a change in the binary value) between two adjacent regions is called a flux reversal, and the density of a particular disk implementation can be measured by the number of regions per inch reliably capable of flux reversal. Higher densities of these regions yield higher-capacity disks. The flux density of a particular system depends on the drive mechanics, the characteristics of the read/write head, and the magnetic properties of the medium.

The read/write head can encode digital information on a disk using a number of recording techniques, including frequency modulation (FM), modified frequency modulation (MFM),

run length limited (RLL) encoding, and advanced run length limited (ARLL) encoding. Each technique offers double the data encoding density of the previous one. The associated control logic is more complex for the denser techniques.

Tracks

A read/write head reads data from or writes data to a thin section of the disk called a track, which is laid out in a circular fashion around the disk (Figure 3-1). Standard 5.25-inch floppy disks contain either 40 (0–39) or 80 (0–79) tracks per side. Like-numbered tracks on either side of a double-sided disk are distinguished by the number of the read/write head used to access the track. For example, track 1 on the top of the disk is identified as head 0, track 1; track 1 on the bottom of the disk is identified as head 1, track 1.

Tracks can be either spirals, as on a phonograph record, or concentric rings. Computer media usually use one of two types of concentric rings. The first type keeps the same number of sectors on each track (*see* Sectors below) and is rotated at a constant angular velocity (CAV). The second type maintains the same recording density across the entire surface of the disk, so a track near the center of a disk contains fewer sectors than a track near the perimeter. This latter type of disk is rotated at different speeds to keep the medium under the magnetic head moving at a constant linear velocity (CLV).
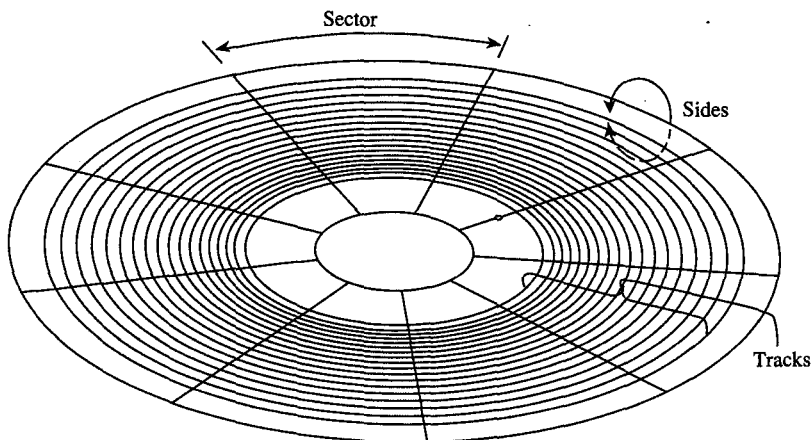
Figure 3-1. The physical layout of a CAV 9-sector, 5.25-inch floppy disk.

Most MS-DOS computers use CAV disks, although a CLV disk can store more sectors using the same type of medium. This difference in storage capacity occurs because the limiting factor is the flux density of the medium and a CAV disk must maintain the same number of magnetic flux regions per sector on the interior of the disk as at the perimeter. Thus, the sectors on or near the perimeter do not use the full capability of the medium and the heads, because the space reserved for each magnetic flux region on the perimeter is larger than that available near the center of the disk. In spite of their greater storage capacity, however, CLV disks (such as CD ROMs) usually have slower access times than CAV disks because of the constant need to fine-tune the motor speed as the head moves from track to track. Thus, CAV disks are preferred for MS-DOS systems.

Heads

Simple disk systems use a single disk, or platter, and use one or two sides of the platter; more complex systems, such as fixed disks, use multiple platters. Disk systems that use both sides of a disk have one read/write head per side; the heads are positioned over the track to be read from or written to by means of a positioning mechanism such as a solenoid or servomotor. The heads are ordinarily moved in unison, using a single head-movement mechanism; thus, heads on opposite sides of a platter in a double-sided disk system typically access the same logical track on their associated sides of the platter. (Performance can be increased by increasing the number of heads to as many as one head per track, eliminating the positioning mechanism. However, because they are quite expensive, such multiple-head systems are generally found only on high-performance minicomputers and mainframes.)

The set of like-numbered tracks on the two sides of a platter (or on all sides of all platters in a multiplatter system) is called a cylinder. Disks are usually partitioned along cylinders. Tracks and cylinders may appear to have the same meaning; however, the term track is used to define a concentric ring containing a specific number of sectors on a single side of a single platter, whereas the term cylinder refers to the number of like-numbered tracks on a device (Figure 3-2).
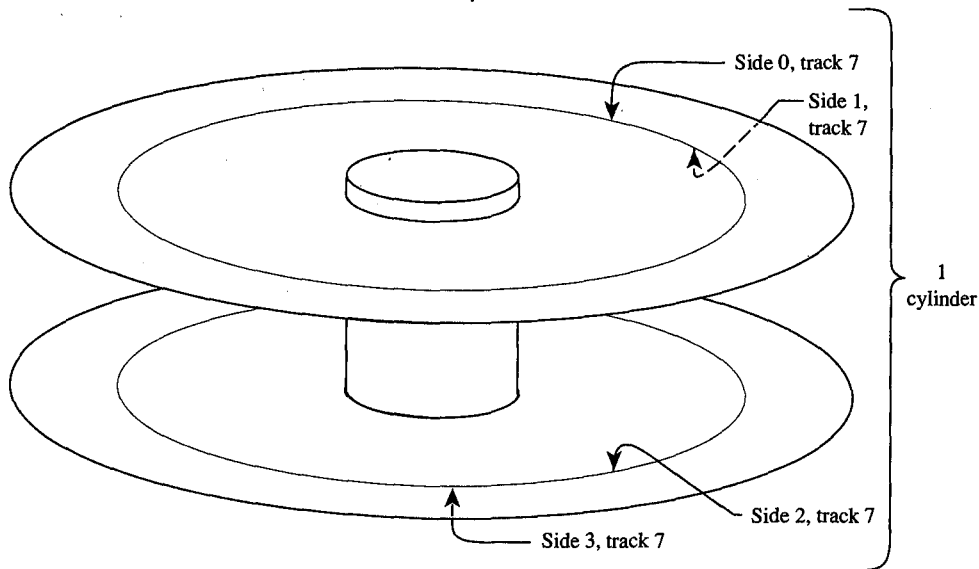
Figure 3-2. Tracks and cylinders on a fixed-disk system.

Sectors

Each track is divided into equal-size portions called sectors. The size of a sector is a power of 2 and is usually greater than 128 bytes — typically, 512 bytes.

Floppy disks are either hard-sectored or soft-sectored, depending on the disk drive and the medium. Hard-sectored disks are implemented using a series of small holes near the

ZTE (USA) 1007, Page 102

center of the disk that indicate the beginning of each sector; these holes are read by a photosensor/LED pair built into the disk drive. Soft-sectored disks are implemented by magnetically marking the beginning of each sector when the disk is formatted. A soft-sectored disk has a single hole near the center of the disk (*see* Figure 3-1) that marks the location of sector 0 for reference when the disk is formatted or when error detection is performed; this hole is also read by a photosensor/LED pair. Fixed disks use a special implementation of soft sectors (*see* below). A hard-sectored floppy disk cannot be used in a disk drive built for use with soft-sectored floppy disks (and vice versa).

In addition to a fixed number of data bytes, both sector types include a certain amount of overhead information, such as error correction and sector identification, in each sector. The structure of each sector is implemented during the formatting process.

Standard fixed disks and 5.25-inch floppy disks generally have from 8 to 17 physical sectors per track. Sectors are numbered beginning at 1. Each sector is uniquely identified by a complete specification of the read/write head, cylinder number, and sector number. To access a particular sector, the disk drive controller hardware moves all heads to the specified cylinder and then activates the appropriate head for the read or write operation.

The read/write heads are mechanically positioned using one of two hardware implementations. The first method, used with floppy disks, employs an "open-loop" servomechanism in which the software computes where the heads should be and the hardware moves them there. (A servomechanism is a device that can move a solenoid or hold it in a fixed position.) An open-loop system employs no feedback mechanism to determine whether the heads were positioned correctly — the hardware simply moves the heads to the requested position and returns an error if the information read there is not what was expected. The positioning mechanism in floppy-disk drives is made with close tolerances because if the positioning of the heads on two drives differs, disks written on one might not be usable on the other.

Most fixed disk systems use the second method — a "closed-loop" servomechanism that reserves one side of one platter for positioning information. This information, which indicates where the tracks and sectors are located, is written on the disk at the factory when the drive is assembled. Positioning the read/write heads in a closed-loop system is actually a two-step process: First, the head assembly is moved to the approximate location of the read or write operation; then the disk controller reads the closed-loop servo information, compares it to the desired location, and fine-tunes the head position accordingly. This fine-tuning approach yields faster access times and also allows for higher-capacity disks because the positioning can be more accurate and the distances between tracks can therefore be smaller. Because the "servo platter" usually has positioning information on one side and data on the other, many systems have an odd number of read/write heads for data.

Interleaving
CAV MS-DOS disks are described in terms of bytes per sector, sectors per track, number of cylinders, and number of read/write heads. Overall access time is based on how fast the disk rotates (rotational latency) and how fast the heads can move from track to track (track-to-track latency).

On most fixed disks, the sectors on the disk are logically or physically numbered so that logically sequential sectors are not physically adjacent (Figure 3-3). The underlying principle is that, because the controller cannot finish processing one sector before the next sequential sector arrives under the read/write head, the logically numbered sectors must be staggered around the track. This staggering of sectors is called skewing or, more commonly, interleaving. A 2-to-1 (2:1) interleave places sequentially accessed sectors so that there is one additional sector between them; a 3:1 interleave places two additional sectors between them. A slower disk controller needs a larger interleave factor. A 3:1 interleave means that three revolutions are required to read all sectors on a track in numeric order.
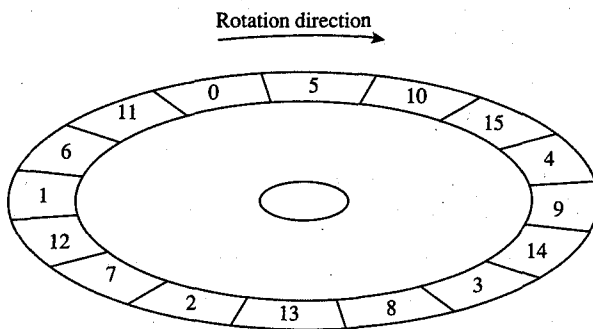
Figure 3-3. A 3:1 interleave.

One approach to improving fixed-disk performance is to decrease the interleave ratio. This generally requires a specialized utility program and also requires that the disk be reformatted to adjust to the new layout. Obviously, a 1:1 interleave is the most efficient, provided the disk controller can process at that speed. The normal interleave for an IBM PC/AT and its standard fixed disk and disk controller is 3:1, but disk controllers are available for the PC/AT that are capable of handling a 1:1 interleave. Floppy disks on MS-DOS-based computers all have a 1:1 interleave ratio.

Layout of a partition

For several reasons, large physical block devices such as fixed disks are often logically partitioned into smaller logical block devices (Figure 3-4). For instance, such partitions allow a device to be shared among different operating systems. Partitions can also be used to keep the size of each logical device within the PC-DOS 32 MB restriction (important for large fixed disks). MS-DOS permits a maximum of four partitions.

A partitioned block device has a partition table located in one sector at the beginning of the disk. This table indicates where the logical block devices are physically located. (Even a partitioned device with only one partition usually has such a table.)
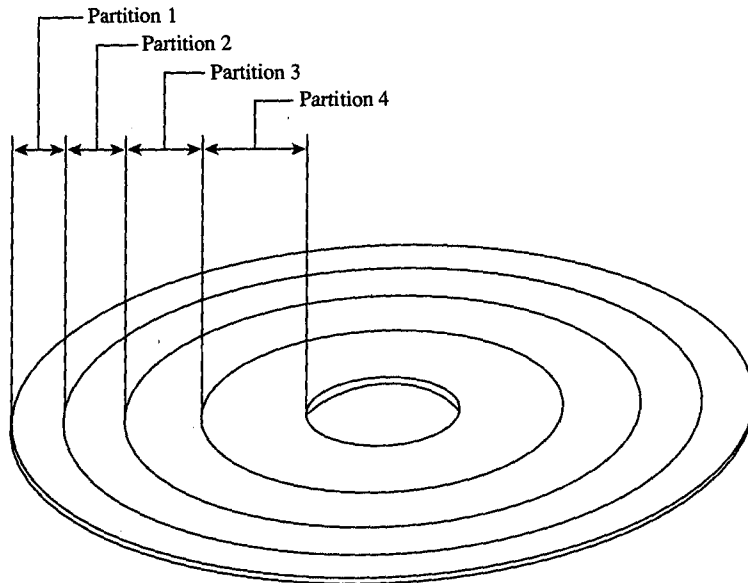
Figure 3-4. A partitioned disk.

Under the MS-DOS partitioning standard, the first physical sector on the fixed disk contains the partition table and a bootstrap program capable of checking the partition table for a bootable partition, loading the bootable partition's boot sector, and transferring control to it. The partition table, located at the end of the first physical sector of the disk, can contain a maximum of four entries:

Offset From Start of Sector	Size (bytes)	Description
01BEH	16	Partition #4
01CEH	16	Partition #3
01DEH	16	Partition #2
01EEH	16	Partition #1
01FEH	2	Signature: AA55H

The partitions are allocated in reverse order. Each 16-byte entry contains the following information:

Offset From Start of Entry	Size (bytes)	Description
00H	1	Boot indicator
01H	1	Beginning head

(more)

Offset From Start of Entry	Size (bytes)	Description
02H	1	Beginning sector
03H	1	Beginning cylinder
04H	1	System indicator
05H	1	Ending head
06H	1	Ending sector
07H	1	Ending cylinder
08H	4	Starting sector (relative to beginning of disk)
0CH	4	Number of sectors in partition

The boot indicator is zero for a nonbootable partition and 80H for a bootable (active) partition. A fixed disk can have only one bootable partition. (When setting a bootable partition, partition programs such as FDISK reset the boot indicators for all other partitions to zero.) *See* USER COMMANDS: FDISK.

The system indicators are

Code	Meaning
00H	Unknown
01H	MS-DOS, 12-bit FAT
04H	MS-DOS, 16-bit FAT

Each partition's boot sector is located at the start of the partition, which is specified in terms of beginning head, beginning sector, and beginning cylinder numbers. This information, stored in the partition table in this order, is loaded into the DX and CX registers by the PC ROM BIOS loader routine when the machine is turned on or restarted. The starting sector of the partition relative to the beginning of the disk is also indicated. The ending head, sector, and cylinder numbers, also included in the partition table, specify the last accessible sector for the partition. The total number of sectors in a partition is the difference between the starting and ending head and cylinder numbers times the number of sectors per cylinder.

MS-DOS versions 2.0 through 3.2 allow only one MS-DOS partition per partitioned device. Various device drivers have been implemented that use a different partition table that allows more than one MS-DOS partition to be installed, but the secondary MS-DOS partitions are usually accessible only by means of an installable device driver that knows about this change. (Even with additional MS-DOS partitions, a fixed disk can have only one bootable partition.)

Layout of a file system

Block devices are accessed on a sector basis. The MS-DOS kernel, through the device driver, sees a block device as a logical fixed-size array of sectors and assumes that the array contains a valid MS-DOS file system. The device driver, in turn, translates the logical sector requests from MS-DOS into physical locations on the block device.

The initial MS-DOS file system is written to the storage medium by the MS-DOS FORMAT program. *See* USER COMMANDS: FORMAT. The general layout for the file system is shown in Figure 3-5.

```
+---------------------------------------------------+
|  OEM identification, BIOS parameter block, Loader routine |
|                    Reserved area                  |
+---------------------------------------------------+
|            File allocation table (FAT) #1         |
+---------------------------------------------------+
|            Possible additional copies of FAT      |
+---------------------------------------------------+
|                                                   |
|                 Root disk directory               |
|                                                   |
+---------------------------------------------------+
|                                                   |
|                                                   |
|                   Files area                      |
+---------------------------------------------------+
```
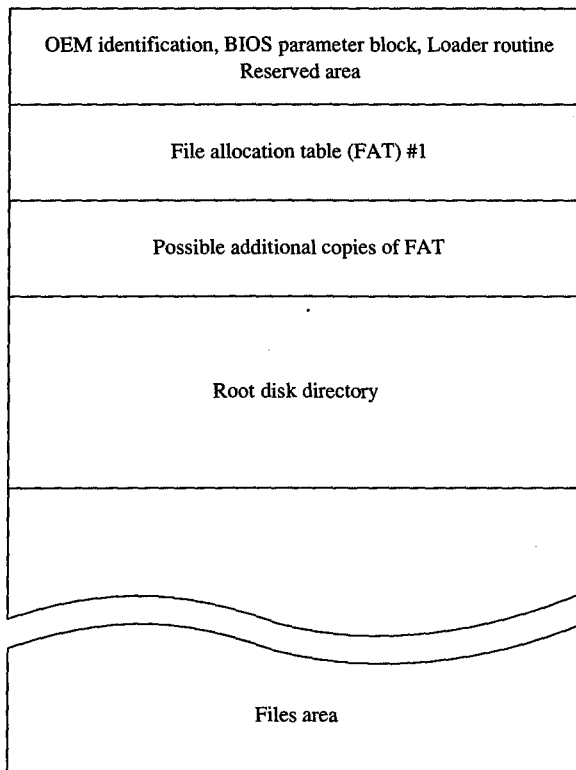
Figure 3-5. The MS-DOS file system.

The boot sector is always at the beginning of a partition. It contains the OEM identification, a loader routine, and a BIOS parameter block (BPB) with information about the device, and it is followed by an optional area of reserved sectors. *See* The Boot Sector below. The reserved area has no specific use, but an OEM might require a more complex loader routine and place it in this area. The file allocation tables (FATs) indicate how the file data area is allocated; the root directory contains a fixed number of directory entries; and the file data area contains data files, subdirectory files, and free data sectors.

All the areas just described — the boot sector, the FAT, the root directory, and the file data area — are of fixed size; that is, they do not change after FORMAT sets up the medium. The size of each of these areas depends on various factors. For instance, the size of the FAT is proportional to the file data area. The root directory size ordinarily depends on the type of device; a single-sided floppy disk can hold 64 entries, a double-sided floppy disk can hold 112, and a fixed disk can hold 256. (RAMdisk drivers such as RAMDRIVE.SYS and some implementations of FORMAT allow the number of directory entries to be specified.)

The file data area is allocated in terms of clusters. A cluster is a fixed number of contiguous sectors. Sector size and cluster size must be a power of 2. The sector size is usually 512 bytes and the cluster size is usually 1, 2, or 4 KB, but larger sector and cluster sizes are possible. Commonly used MS-DOS cluster sizes are

Disk Type	Sectors/Cluster	Bytes/Cluster*
Single-sided floppy disk	1	512
Double-sided floppy disk	2	1024
PC/AT fixed disk	4	2048
PC/XT fixed disk	8	4096
Other fixed disks	16	8192
Other fixed disks	32	16384

*Assumes 512 bytes per sector.

In general, larger cluster sizes are used to support larger fixed disks. Although smaller cluster sizes make allocation more space-efficient, larger clusters are usually more efficient for random and sequential access, especially if the clusters for a single file are not sequentially allocated.

The file allocation table contains one entry per cluster in the file data area. Doubling the sectors per cluster will also halve the number of FAT entries for a given partition. *See* The File Allocation Table below.

The boot sector

The boot sector (Figure 3-6) contains a BIOS parameter block, a loader routine, and some other fields useful to device drivers. The BPB describes a number of physical parameters of the device, as well as the location and size of the other areas on the device. The device driver returns the BPB information to MS-DOS when requested, so that MS-DOS can determine how the disk is configured.

Figure 3-7 is a hexadecimal dump of an actual boot sector. The first 3 bytes of the boot sector shown in Figure 3-7 would be E9H 2CH 00H if a long jump were used instead of a short one (as in early versions of MS-DOS). The last 2 bytes in the sector, 55H and AAH, are a fixed signature used by the loader routine to verify that the sector is a valid boot sector.

00H	E9 XX XX or EB XX 90
03H	OEM name and version (8 bytes)
0BH	Bytes per sector (2 bytes)
0DH	Sectors per allocation unit (1 byte)
0EH	Reserved sectors, starting at 0 (2 bytes)
10H	Number of FATs (1 byte)
11H	Number of root-directory entries (2 bytes)
13H	Total sectors in logical volume (2 bytes)
15H	Media descriptor byte
16H	Number of sectors per FAT (2 bytes)
18H	Sectors per track (2 bytes)
1AH	Number of heads (2 bytes)
1CH	Number of hidden sectors (2 bytes)
1EH	Loader routine

BPB (bracket spanning 0BH through 17H)

Figure 3-6. Map of the boot sector of an MS-DOS disk. Bytes 0BH through 17H are the BIOS parameter block (BPB).

The BPB information contained in bytes 0BH through 17H indicates that there are

```
 512  bytes per sector
   2  sectors per cluster
   1  reserved sector (for the boot sector)
   2  FATs
 112  root directory entries
1440  sectors on the disk
 F9H  media descriptor
   3  sectors per FAT
```

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   EB 2D 90 20 20 20 20 20-20 20 20 00 02 02 01 00    k-.         .....
0010   02 70 00 A0 05 F9 03 00-09 00 02 00 00 00 00 00    .p. .y..........
0020   00 0A 00 00 DF 02 25 02-09 2A FF 50 F6 0A 02 FA    ....—.%..*.Pv..z
0030   B8 C0 07 8E D8 BC 00 7C-33 C0 8E D0 8E C0 FB FC    8@..X<.!3@.P.@{|

       .
       .
       .
0180   0A 44 69 73 6B 20 42 6F-6F 74 20 46 61 69 6C 75    .Disk Boot Failu
0190   72 65 0D 0A 0D 0A 4E 6F-6E 2D 53 79 73 74 65 6D    re....Non-System
01A0   20 64 69 73 6B 20 6F 72-20 64 69 73 6B 20 65 72     disk or disk er
01B0   72 6F 72 0D 0A 52 65 70-6C 61 63 65 20 61 6E 64    ror..Replace and
01C0   20 70 72 65 73 73 20 61-6E 79 20 6B 65 79 20 77     press any key w
01D0   68 65 6E 20 72 65 61 64-79 0D 0A 00 00 00 00 00    hen ready.......
01E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
01F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA    ..............*
```

Figure 3-7. Hexadecimal dump of an MS-DOS boot sector. The BPB is highlighted.

Additional information immediately after the BPB indicates that there are 9 sectors per track, 2 read/write heads, and 0 hidden sectors.

The media descriptor, which appears in the BPB and in the first byte of each FAT, is used to indicate the type of medium currently in a drive. IBM-compatible media have the following descriptors:

Descriptor	Media Type	MS-DOS Versions
0F8H	Fixed disk	2, 3
0F0H	3.5-inch, 2-sided, 18 sector	3.2
0F9H	3.5-inch, 2-sided, 9 sector	3.2
0F9H	5.25-inch, 2-sided, 15 sector	3.x
0FCH	5.25-inch, 1-sided, 9 sector	2.x, 3.x
0FDH	5.25-inch, 2-sided, 9 sector	2.x, 3.x
0FEH	5.25-inch, 1-sided, 8 sector	1.x, 2.x, 3.x
0FFH	5.25-inch, 2-sided, 8 sector	1.x (except 1.0), 2, 3
0FEH	8-inch, 1-sided, single-density	
0FDH	8-inch, 2-sided, single-density	
0FEH	8-inch, 1-sided, double-density	
0FDH	8-inch, 2-sided, double-density	

The file allocation table

The file allocation table provides a map to the storage locations of files on a disk by indicating which clusters are allocated to each file and in what order. To enable MS-DOS to locate a file, the file's directory entry contains its beginning FAT entry number. This FAT entry, in turn, contains the entry number of the next cluster if the file is larger than one cluster or a last-cluster number if there is only one cluster associated with the file. A file whose size implies that it occupies 10 clusters will have 10 FAT entries and 9 FAT links. (The set of links for a particular file is called a chain.)

Additional copies of the FAT are used to provide backup in case of damage to the first, or primary, FAT; the typical floppy disk or fixed disk contains two FATs. The FATs are arranged sequentially after the boot sector, with some possible intervening reserved area. MS-DOS ordinarily uses the primary FAT but updates all FATs when a change occurs. It also compares all FATs when a disk is first accessed, to make sure they match.

MS-DOS supports two types of FAT: One uses 12-bit links; the other, introduced with version 3.0 to accommodate large fixed disks with more than 4087 clusters, uses 16-bit links.

The first two entries of a FAT are always reserved and are filled with a copy of the media descriptor byte and two (for a 12-bit FAT) or three (for a 16-bit FAT) 0FFH bytes, as shown in the following dumps of the first 16 bytes of the FAT:

12-bit FAT:

```
F9 FF FF 03 40 00 FF 6F-00 07 F0 FF 00 00 00 00
```

16-bit FAT:

```
F8 FF FF FF 03 00 04 00-FF FF 06 00 07 00 FF FF
```

The remaining FAT entries have a one-to-one relationship with the clusters in the file data area. Each cluster's use status is indicated by its corresponding FAT value. (FORMAT initially marks the FAT entry for each cluster as free.) The use status is one of the following:

12-bit	16-bit	Meaning
000H	0000H	Free cluster
001H	0001H	Unused code
FF0–FF6H	FFF0–FFF6H	Reserved
FF7H	FFF7H	Bad cluster; cannot be used
FF8–FFFH	FFF8–FFFFH	Last cluster of file
All other values	All other values	Link to next cluster in file

If a FAT entry is nonzero, the corresponding cluster has been allocated. A free cluster is found by scanning the FAT from the beginning to find the first zero value. Bad clusters are ordinarily identified during formatting. Figure 3-8 shows a typical FAT chain.
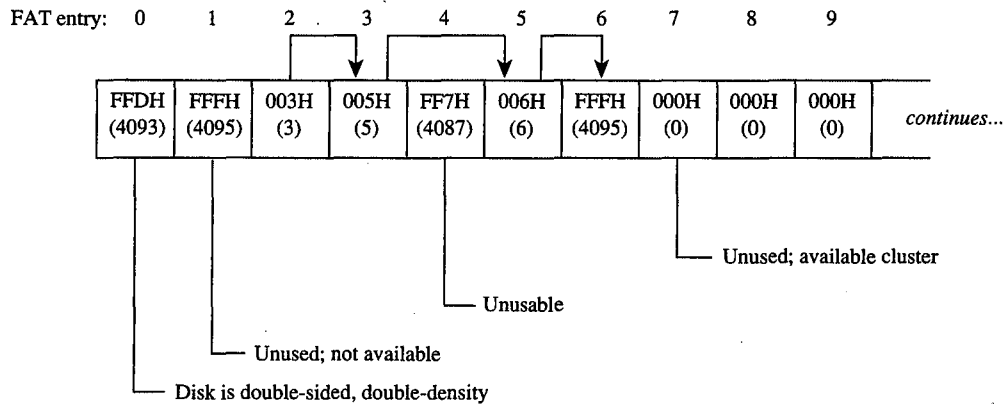
Figure 3-8. Space allocation in the FAT for a typical MS-DOS disk.

Free FAT entries contain a link value of zero; a link value of 1 is never used. Thus, the first allocatable link number, associated with the first available cluster in the file data area, is 2, which is the number assigned to the first *physical* cluster in the file data area. Figure 3-9 shows the relationship of files, FAT entries, and clusters in the file data area.

There is no *logical* difference between the operation of the 12-bit and 16-bit FAT entries; the difference is simply in the storage and access methods. Because the 8086 is specifically designed to manipulate 8- or 16-bit values efficiently, the access procedure for the 12-bit FAT is more complex than that for the 16-bit FAT (*see* Figures 3-10 and 3-11).

Special considerations
The FAT is a highly efficient bookkeeping system, but various tradeoffs and problems can occur. One tradeoff is having a partially filled cluster at the end of a file. This situation leads to an efficiency problem when a large cluster size is used, because an entire cluster is allocated, regardless of the number of bytes it contains. For example, ten 100-byte files on a disk with 16 KB clusters use 160 KB of disk space; the same files on a disk with 1 KB clusters use only 10 KB—a difference of 150 KB, or 15 times less storage used by the smaller cluster size. On the other hand, the 12-bit FAT routine in Figure 3-10 shows the difficulty (and therefore slowness) of moving through a large file that has a long linked list of many small clusters. Therefore, the nature of the data must be considered: Large database applications work best with a larger cluster size; a smaller cluster size allows many small text files to fit on a disk. (The programmer writing the device driver for a disk device ordinarily sets the cluster size.)

12-bit FAT:
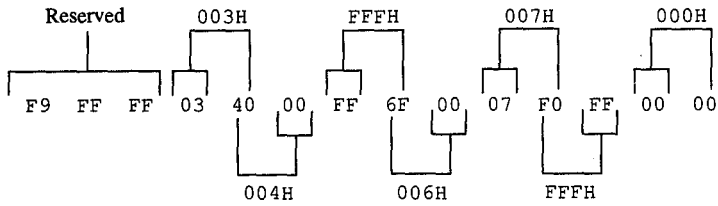
16 bit FAT:
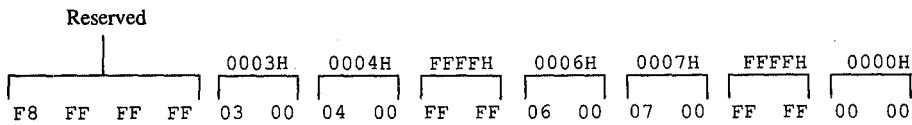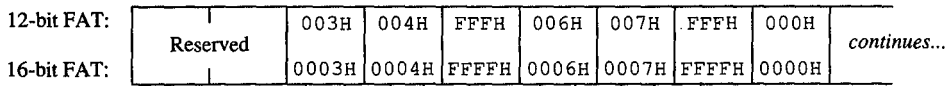
Figure 3-9. Correspondence between the FAT and the file data area.

```
; ---- Obtain the next link number from a 12-bit FAT  -----
;
; Parameters:
;       ax      = current entry number
;       ds:bx   = address of FAT (must be contiguous)
;
; Returns:
;       ax      = next link number
;
; Uses: ax, bx, cx
next12  proc    near
        add     bx,ax           ; ds:bx = partial index
        shr     ax,1            ; ax = offset/2
                                ; carry = no shift needed
        pushf                   ; save carry
        add     bx,ax           ; ds:bx = next cluster number index
        mov     ax,[bx]         ; ax = next cluster number
        popf                    ; carry = no shift needed
        jc      shift           ; skip if using top 12 bits
        and     ax,0fffh        ; ax = lower 12 bits
        ret
shift:  mov     cx,4            ; cx = shift count
        shr     ax,cl           ; ax = top 12 bits in lower 12 bits
        ret
next12  endp
```

Figure 3-10. Assembly-language routine to access a 12-bit FAT.

```
; ---- Obtain the next link number from a 16-bit FAT  -----
;
; Parameters:
;       ax      = current entry number
;       ds:bx   = address of FAT (must be contiguous)
;
; Returns:
;       ax      = next link number
;
; Uses: ax, bx, cx
next16  proc    near
        add     ax,ax           ; ax = word offset
        add     bx,ax           ; ds:bx = next link number index
        mov     ax,[bx]         ; ax = next link number
        ret
next16  endp
```

Figure 3-11. Assembly-language routine to access a 16-bit FAT.

Problems with corrupted directories or FATs, induced by such events as power failures and programs running wild, can lead to greater problems if not corrected. The MS-DOS CHKDSK program can detect and fix some of these problems. *See* USER COMMANDS: CHKDSK. For example, one common problem is dangling allocation lists caused by the absence of a directory entry pointing to the start of the list. This situation often results when the directory entry was not updated because a file was not closed before the computer was turned off or restarted. The effect is relatively benign: The data is inaccessible, but this limitation does not affect other file allocation operations. CHKDSK can fix this problem by making a new directory entry and linking it to the list.

Another difficulty occurs when the file size in a directory entry does not match the file length as computed by traversing the linked list in the FAT. This problem can result in improper operation of a program and in error responses from MS-DOS.

A more complex (and rarer) problem occurs when the directory entry is properly set up but all or some portion of the linked list is also referenced by another directory entry. The problem is grave, because writing or appending to one file changes the contents of the other file. This error usually causes severe data and/or directory corruption or causes the system to crash.

A similar difficulty occurs when a linked list terminates with a free cluster instead of a last-cluster number. If the free cluster is allocated before the error is corrected, the problem eventually reverts to the preceding problem. An associated difficulty occurs if a link value of 1 or a link value that exceeds the size of the FAT is encountered.

In addition to CHKDSK, a number of commercially available utility programs can be used to assist in FAT maintenance. For instance, disk reorganizers can be used to essentially rearrange the FAT and adjust the directory so that all files on a disk are laid out sequentially in the file data area and, of course, in the FAT.

The root directory

Directory entries, which are 32 bytes long, are found in both the root directory and the subdirectories. Each entry includes a filename and extension, the file's size, the starting FAT entry, the time and date the file was created or last revised, and the file's attributes. This structure resembles the format of the CP/M-style file control blocks (FCBs) used by the MS-DOS version 1.x file functions. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Disk Directories and Volume Labels.

The MS-DOS file-naming convention is also derived from CP/M: an eight-character filename followed by a three-character file type, each left aligned and padded with spaces if necessary. Within the limitations of the character set, the name and type are completely arbitrary. The time and date stamps are in the same format used by other MS-DOS functions and reflect the time the file was last written to.

Figure 3-12 shows a dump of a 512-byte directory sector containing 16 directory entries. (Each entry occupies two lines in this example.) The byte at offset 0ABH, containing a 10H, signifies that the entry starting at 0A0H is for a subdirectory. The byte at offset 160H, containing 0E5H, means that the file has been deleted. The byte at offset 8BH, containing

the value 08H, indicates that the directory entry beginning at offset 80H is a volume label. Finally the zero byte at offset 1E0H marks the end of the directory, indicating that the subsequent entries in the directory have never been used and therefore need not be searched (versions 2.0 and later).

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00    IO    SYS'....
0010   00 00 00 00 00 00 59 53-89 0B 02 00 D1 12 00 00    ......YS....Q...
0020   4F 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00    MSDOS   SYS'....
0030   00 00 00 00 00 00 41 49-52 0A 07 00 C9 43 00 00    ......AIR...IC..
0040   41 4E 53 49 20 20 20 20-53 59 53 20 00 00 00 00    ANSI    SYS ....
0050   00 00 00 00 00 00 41 49-52 0A 18 00 76 07 00 00    ......AIR...v...
0060   58 54 41 4C 4B 20 20 20-45 58 45 20 00 00 00 00    XTALK   EXE ....
0070   00 00 00 00 00 00 F7 7D-38 09 23 02 84 0B 01 00    ......w}8.#.....
0080   4C 41 42 45 4C 20 20 20-20 20 20 08 00 00 00 00    LABEL       ....
0090   00 00 00 00 00 00 8C 20-2A 09 00 00 00 00 00 00    ....... *.D..R..
00A0   4C 4F 54 55 53 20 20 20-20 20 20 10 00 00 00 00    LOTUS       ....
00B0   00 00 00 00 00 00 E0 0A-E1 06 A6 01 00 00 00 00    ......'.a.&.a...
00C0   4C 54 53 4C 4F 41 44 20-43 4F 4D 20 00 00 00 00    LTSLOAD COM ....
00D0   00 00 00 00 00 00 E0 0A-E1 06 A7 01 A0 27 00 00    ......'.a.'. '..
00E0   4D 43 49 2D 53 46 20 20-58 54 4B 20 00 00 00 00    MCI-SF  XTK ....
00F0   00 00 00 00 00 00 46 19-32 0D B1 01 79 04 00 00    ......F.2.1.y...
0100   58 54 41 4C 4B 20 20 20-48 4C 50 20 00 00 00 00    XTALK.  HLP ....
0110   00 00 00 00 00 00 C5 6D-73 07 A3 02 AF 88 00 00    ......Ems.#./...
0120   54 58 20 20 20 20 20 20-43 4F 4D 20 00 00 00 00    TX      COM ....
0130   00 00 00 00 00 00 05 61-65 0C 39 01 E8 20 00 00    .......ae.9.h ..
0140   43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00    COMMAND COM ....
0150   00 00 00 00 00 00 41 49-52 0A 27 00 55 3F 00 00    ......AIR.'.U?..
0160   E5 32 33 20 20 20 20 20-45 58 45 20 00 00 00 00    e23     EXE ....
0170   00 00 00 00 00 00 9C B2-85 0B 42 01 80 5F 01 00    .......2..B.._..
0180   47 44 20 20 20 20 20 20-44 52 56 20 00 00 00 00    GD      DRV ....
0190   00 00 00 00 00 00 E0 0A-E1 06 9A 01 5B 08 00 00    ......'.a...[...
01A0   4B 42 20 20 20 20 20 20-44 52 56 20 00 00 00 00    KB      DRV ....
01B0   00 00 00 00 00 00 E0 0A-E1 06 9D 01 60 01 00 00    ......'.a...'...
01C0   50 52 20 20 20 20 20 20-44 52 56 20 00 00 00 00    PR      DRV ....
01D0   00 00 00 00 00 00 E0 0A-E1 06 9E 01 49 01 00 00    ......'.a...I...
01E0   00 F6 F6 F6 F6 F6 F6 F6-F6 F6 F6 F6 F6 F6 F6 F6    ................
01F0   F6 F6 F6 F6 F6 F6 F6 F6-F6 F6 F6 F6 F6 F6 F6 F6    ................
```

Figure 3-12. Hexadecimal dump of a 512-byte directory sector.

The sector shown in Figure 3-12 is actually an example of the first directory sector in the root directory of a bootable disk. Notice that IO.SYS and MSDOS.SYS are the first two files in the directory and that the file attribute byte (offset 0BH in a directory entry) has a binary value of 00100111, indicating that both files have hidden (bit 1 = 1), system (bit 0 = 1), and read-only (bit 2 = 1) attributes. The archive bit (bit 5) is also set, marking the files for possible backup.

The root directory can optionally have a special type of entry called a volume label, identified by an attribute type of 08H, that is used to identify disks by name. A root directory can contain only one volume label. The root directory can also contain entries that point to subdirectories; such entries are identified by an attribute type of 10H and a file size of zero. Programs that manipulate subdirectories must do so by tracing through their chains of clusters in the FAT.

Two other special types of directory entries are found only within subdirectories. These entries have the filenames . and .. and correspond to the current directory and the parent directory of the current directory. These special entries, sometimes called directory aliases, can be used to move quickly through the directory structure.

The maximum pathname length supported by MS-DOS, excluding a drive specifier but including any filename and extension and subdirectory name separators, is 64 characters. The size of the directory structure itself is limited only by the number of root directory entries and the available disk space.

The file area

The file area contains subdirectories, file data, and unallocated clusters. The area is divided into fixed-size clusters and the use for a particular cluster is specified by the corresponding FAT entry.

Other MS-DOS Storage Devices

As mentioned earlier, MS-DOS supports other types of storage devices, such as magnetic-tape drives and CD ROM drives. Tape drives are most often used for archiving and for sequential transaction processing and therefore are not discussed here.

CD ROMs are compact laser discs that hold a massive amount of information — a single side of a CD ROM can hold almost 500 MB of data. However, there are some drawbacks to current CD ROM technology. For instance, data cannot be written to them — the information is placed on the compact disk at the factory when the disk is made and is available on a read-only basis. In addition, the access time for a CD ROM is much slower than for most magnetic-disk systems. Even with these limitations, however, the ability to hold so much information makes CD ROM a good method for storing large amounts of static information.

William Wong

Part B
Programming for MS-DOS

Article 4
Structure of an Application Program

Planning an MS-DOS application program requires serious analysis of the program's size. This analysis can help the programmer determine which of the two program styles supported by MS-DOS best suits the application. The .EXE program structure provides a large program with benefits resulting from the extra 512 bytes (or more) of header that preface all .EXE files. On the other hand, at the cost of losing the extra benefits, the .COM program structure does not burden a small program with the overhead of these extra header bytes.

Because .COM programs start their lives as .EXE programs (before being converted by EXE2BIN) and because several aspects of application programming under MS-DOS remain similar regardless of the program structure used, a solid understanding of .EXE structures is beneficial even to the programmer who plans on writing only .COM programs. Therefore, we'll begin our discussion with the structure and behavior of .EXE programs and then look at differences between .COM programs and .EXE programs, including restrictions on the structure and content of .COM programs.

The .EXE Program

The .EXE program has several advantages over the .COM program for application design. Considerations that could lead to the choice of the .EXE format include

- Extremely large programs
- Multiple segments
- Overlays
- Segment and far address constants
- Long calls
- Possibility of upgrading programs to MS OS/2 protected mode

The principal advantages of the .EXE format are provided by the file header. Most important, the header contains information that permits a program to make direct segment address references — a requirement if the program is to grow beyond 64 KB.

The file header also tells MS-DOS how much memory the program requires. This information keeps memory not required by the program from being allocated to the program — an important consideration if the program is to be upgraded in the future to run efficiently under MS OS/2 protected mode.

Before discussing the .EXE program structure in detail, we'll look at how .EXE programs behave.

ZTE (USA) 1007, Page 119

Giving control to the .EXE program

Figure 4-1 gives an example of how a .EXE program might appear in memory when MS-DOS first gives the program control. The diagram shows Microsoft's preferred program segment arrangement.
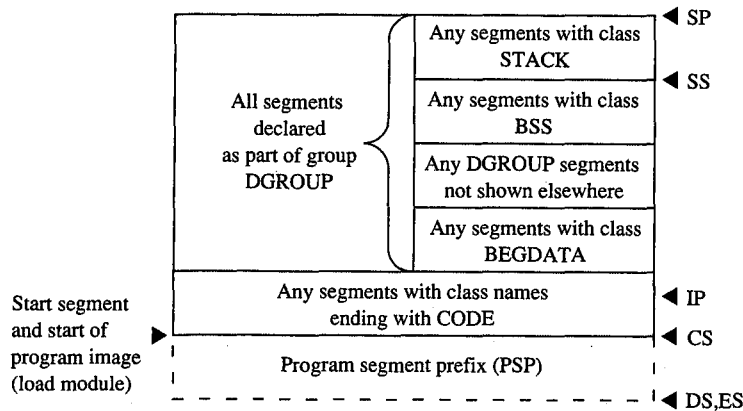
Figure 4-1. The .EXE program: memory map diagram with register pointers.

Before transferring control to the .EXE program, MS-DOS initializes various areas of memory and several of the microprocessor's registers. The following discussion explains what to expect from MS-DOS before it gives the .EXE program control.

The program segment prefix

The program segment prefix (PSP) is not a direct result of any program code. Rather, this special 256-byte (16-paragraph) page of memory is built by MS-DOS in front of all .EXE and .COM programs when they are loaded into memory. Although the PSP does contain several fields of use to newer programs, it exists primarily as a remnant of CP/M — Microsoft adopted the PSP for ease in porting the vast number of programs available under CP/M to the MS-DOS environment. Figure 4-2 shows the fields that make up the PSP.

PSP:0000H (Terminate [old Warm Boot] Vector) The PSP begins with an 8086-family INT 20H instruction, which the program can use to transfer control back to MS-DOS. The PSP includes this instruction at offset 00H because this address was the WBOOT (Warm Boot/Terminate) vector under CP/M and CP/M programs usually terminated by jumping to this vector. This method of termination should not be used in newer programs. *See* Terminating the .EXE Program below.

PSP:0002H (Address of Last Segment Allocated to Program) MS-DOS introduced the word at offset 02H into the PSP. It contains the segment address of the paragraph following the block of memory allocated to the program. This address should be used only to determine the size or the end of the memory block allocated to the program; it must not be considered a pointer to free memory that the program can appropriate. In most cases this address will *not* point to free memory, because any free memory will already have been

	x0H	x1H	x2H	x3H	x4H	x5H	x6H	x7H	x8H	x9H	xAH	xBH	xCH	xDH	xEH	xFH
0xH	INT 20H		End alloc		Resv.	Far call to MS-DOS fn handler					Prev terminate address				Prev Ctrl C...	
	0CDH	20H	seg lo	seg hi		9AH	ofs lo	ofs hi	seg lo	seg hi	ofs lo	ofs hi	seg lo	seg hi	ofs lo	ofs hi
1xH	...address		Prev critical error address				Reserved...									
	seg lo	seg hi	ofs lo	ofs hi	seg lo	seg hi										
2xH	...Reserved												Environ seg		Reserved...	
													seg lo	seg hi		
3xH	...Reserved...															
4xH	...Reserved															
5xH	INT 21H and RETF			Reserved									Primary FCB...			
	0CDH	21H	0CBH										d	F	i	l
6xH	...Primary file control block (FCB)												Secondary FCB...			
	e	n	a	m	e	E	x	t	00H	00H	00H	00H	d	F	i	l
7xH	...Secondary file control block (FCB)												Reserved			
	e	n	a	m	e	E	x	t	00H	00H	00H	00H				
8xH	Command tail and default disk transfer area (DTA) (continues through 0FFH)...															
	Len															

MS-DOS 2.0 and later only

Figure 4-2. The program segment prefix (PSP).

allocated to the program unless the program was linked using the /CPARMAXALLOC switch. Even when /CPARMAXALLOC is used, MS-DOS may fit the program into a block of memory only as big as the program requires. Well-behaved programs should acquire additional memory only through the MS-DOS function calls provided for that purpose.

PSP:0005H (MS-DOS Function Call [old BDOS] Vector) Offset 05H is also a hand-me-down from CP/M. This location contains an 8086-family far (intersegment) call instruction to MS-DOS's function request handler. (Under CP/M, this address was the Basic Disk Operating System [BDOS] vector, which served a similar purpose.) This vector should not be used to call MS-DOS in newer programs. The System Calls section of this book explains the newer, approved method for calling MS-DOS. MS-DOS provides this vector only to support CP/M-style programs and therefore honors only the CP/M-style functions (00–24H) through it.

PSP:000AH-0015H (Parent's 22H, 23H, and 24H Interrupt Vector Save) MS-DOS uses offsets 0AH through 15H to save the contents of three program-specific interrupt vectors. MS-DOS must save these vectors because it permits any program to execute another program (called a child process) through an MS-DOS function call that returns control to the original program when the called program terminates. Because the original program resumes executing when the child program terminates, MS-DOS must restore these three

interrupt vectors for the original program in case the called program changed them. The three vectors involved include the program termination handler vector (Interrupt 22H), the Control-C/Control-Break handler vector (Interrupt 23H), and the critical error handler vector (Interrupt 24H). MS-DOS saves the original preexecution contents of these vectors in the child program's PSP as doubleword fields beginning at offsets 0AH for the program termination handler vector, 0EH for the Control-C/Control-Break handler vector, and 12H for the critical error handler vector.

PSP:002CH (Segment Address of Environment) Under MS-DOS versions 2.0 and later, the word at offset 2CH contains one of the most useful pieces of information a program can find in the PSP — the segment address of the first paragraph of the MS-DOS environment. This pointer enables the program to search through the environment for any configuration or directory search path strings placed there by users with the SET command.

PSP:0050H (New MS-DOS Call Vector) Many programmers disregard the contents of offset 50H. The location consists simply of an INT 21H instruction followed by a RETF. A .EXE program can call this location using a far call as a means of accessing the MS-DOS function handler. Of course, the program can also simply do an INT 21H directly, which is smaller and faster than calling 50H. Unlike calls to offset 05H, calls to offset 50H can request the full range of MS-DOS functions.

PSP:005CH (Default File Control Block 1) and PSP:006CH (Default File Control Block 2) MS-DOS parses the first two parameters the user enters in the command line following the program's name. If the first parameter qualifies as a valid (limited) MS-DOS filename (the name can be preceded by a drive letter but not a directory path), MS-DOS initializes offsets 5CH through 6BH with the first 16 bytes of an unopened file control block (FCB) for the specified file. If the second parameter also qualifies as a valid MS-DOS filename, MS-DOS initializes offsets 6CH through 7BH with the first 16 bytes of an unopened FCB for the second specified file. If the user specifies a directory path as part of either filename, MS-DOS initializes only the drive code in the associated FCB. Many programmers no longer use this feature, because file access using FCBs does not support directory paths and other newer MS-DOS features.

Because FCBs expand to 37 bytes when the file is opened, opening the first FCB at offset 5CH causes it to grow from 16 bytes to 37 bytes and to overwrite the second FCB. Similarly, opening the second FCB at offset 6CH causes it to expand and to overwrite the first part of the command tail and default disk transfer area (DTA). (The command tail and default DTA are described below.) To use the contents of both default FCBs, the program should copy the FCBs to a pair of 37-byte fields located in the program's data area. The program can use the first FCB without moving it only after relocating the second FCB (if necessary) and only by performing sequential reads or writes when using the first FCB. To perform random reads and writes using the first FCB, the programmer must either move the first FCB or change the default DTA address. Otherwise, the first FCB's random record field will overlap the start of the default DTA. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

PSP:0080H (Command Tail and Default DTA) The default DTA resides in the entire second half (128 bytes) of the PSP. MS-DOS uses this area of memory as the default record buffer if the program uses the FCB-style file access functions. Again, MS-DOS inherited this location from CP/M. (MS-DOS provides a function the program can call to change the address MS-DOS will use as the current DTA. *See* SYSTEM CALLS: INTERRUPT 21H: Function 1AH.) Because the default DTA serves no purpose until the program performs some file activity that requires it, MS-DOS places the command tail in this area for the program to examine. The command tail consists of any text the user types following the program name when executing the program. Normally, an ASCII space (20H) is the first character in the command tail, but any character MS-DOS recognizes as a separator can occupy this position. MS-DOS stores the command-tail text starting at offset 81H and always places an ASCII carriage return (0DH) at the end of the text. As an additional aid, it places the length of the command tail at offset 80H. This length includes all characters except the final 0DH. For example, the command line

```
C>DOIT WITH CLASS   <Enter>
```

will result in the program DOIT being executed with PSP:0080H containing

```
0B   20 57 49 54 48 20 43 4C 41 53 53 0D
len  sp W  I  T  H  sp C  L  A  S  S  cr
```

The stack

Because .EXE-style programs did not exist under CP/M, MS-DOS expects .EXE programs to operate in strictly MS-DOS fashion. For example, MS-DOS expects the .EXE program to supply its own stack. (Figure 4-1 shows the program's stack as the top box in the diagram.)

Microsoft's high-level-language compilers create a stack themselves, but when writing in assembly language the programmer must specifically declare one or more segments with the STACK *combine* type. If the programmer declares multiple stack segments, possibly in different source modules, the linker combines them into one large segment. *See* Controlling the .EXE Program's Structure below.

Many programmers declare their stack segments as preinitialized with some recognizable repeating string such as *STACK.* This makes it possible to examine the program's stack in memory (using a debugger such as DEBUG) to determine how much stack space the program actually used. On the other hand, if the stack is left as uninitialized memory and linked at the end of the .EXE program, it will not require space within the .EXE file. (The reason for this will become more apparent when we examine the structure of a .EXE file.)

Note: When multiple stack segments have been declared in different .ASM files, the Microsoft Object Linker (LINK) correctly allocates the total amount of stack space specified in all the source modules, but the initialization data from all modules is overlapped module by module at the high end of the combined segment.

An important difference between .COM and .EXE programs is that MS-DOS preinitializes a .COM program's stack with a termination address before transferring control to the program. MS-DOS does not do this for .EXE programs, so a .EXE program *cannot* simply execute an 8086-family RET instruction as a means of terminating.

Note: In the assembly-language files generated for a Microsoft C program or for programs in most other high-level-languages, the compiler's placement of a RET instruction at the end of the *main* function/subroutine/procedure might seem confusing. After all, MS-DOS does not place any return address on the stack. The compiler places the RET at the end of *main* because *main* does not receive control directly from MS-DOS. A library initialization routine receives control from MS-DOS; this routine then calls *main.* When *main* performs the RET, it returns control to a library termination routine, which then terminates back to MS-DOS in an approved manner.

Preallocated memory

While loading a .EXE program, MS-DOS performs several steps to determine the initial amount of memory to be allocated to the program. First, MS-DOS reads the two values the linker places near the start of the .EXE header: The first value, MINALLOC, indicates the minimum amount of extra memory the program requires to start executing; the second value, MAXALLOC, indicates the maximum amount of extra memory the program would like allocated before it starts executing. Next, MS-DOS locates the largest free block of memory available. If the size of the program's image within the .EXE file combined with the value specified for MINALLOC exceeds the memory block it found, MS-DOS returns an error to the process trying to load the program. If that process is COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message and terminates the user's execution request. If the block exceeds the program's MINALLOC requirement, MS-DOS then compares the memory block against the program's image combined with the MAXALLOC request. If the free block exceeds the maximum memory requested by the program, MS-DOS allocates only the maximum request; otherwise, it allocates the entire block. MS-DOS then builds a PSP at the start of this block and loads the program's image from the .EXE file into memory following the PSP.

This process ensures that the extra memory allocated to the program will immediately follow the program's image. The same will not necessarily be true for any memory MS-DOS allocates to the program as a result of MS-DOS function calls the program performs during its execution. Only function calls requesting MS-DOS to increase the initial allocation can guarantee additional contiguous memory. (Of course, the granting of such increase requests depends on the availability of free memory following the initial allocation.)

Programmers writing .EXE programs sometimes find the lack of keywords or compiler/ assembler switches that deal with MINALLOC (and possibly MAXALLOC) confusing. The programmer never explicitly specifies a MINALLOC value because LINK sets MINALLOC to the total size of all uninitialized data and/or stack segments linked at the very end of the program. The MINALLOC field allows the compiler to indicate the size of the initialized data fields in the load module without actually including the fields themselves, resulting in a smaller .EXE program file. For LINK to minimize the size of the .EXE file, the program must be coded and linked in such a way as to place all uninitialized data fields at the end of the program. Microsoft high-level-language compilers handle this automatically; assembly-language programmers must give LINK a little help.

ZTE (USA) 1007, Page 124

Note: Beginning and even advanced assembly-language programmers can easily fall into an argument with the assembler over field addressing when attempting to place data fields after the code in the source file. This argument can be avoided if programmers use the SEGMENT and GROUP assembler directives. *See* Controlling the .EXE Program's Structure below.

No reliable method exists for the linker to determine the correct MAXALLOC value required by the .EXE program. Therefore, LINK uses a "safe" value of FFFFH, which causes MS-DOS to allocate all of the largest block of free memory — which is usually *all* free memory — to the program. Unless a program specifically releases the memory for which it has no use, it denies multitasking supervisor programs, such as IBM's TopView, any memory in which to execute additional programs — hence the rule that a well-behaved program releases unneeded memory during its initialization. Unfortunately, this memory conservation approach provides no help if a multitasking supervisor supports the ability to load several programs into memory without executing them. Therefore, programs that have correctly established MAXALLOC values actually are well-behaved programs.

To this end, newer versions of Microsoft LINK include the /CPARMAXALLOC switch to permit specification of the maximum amount of memory required by the program. The /CPARMAXALLOC switch can also be used to set MAXALLOC to a value that is known to be less than MINALLOC. For example, specifying a MAXALLOC value of 1 (/CP:1) forces MS-DOS to allocate only MINALLOC extra paragraphs to the program. In addition, Microsoft supplies a program called EXEMOD with most of its languages. This program permits modification of the MAXALLOC field in the headers of existing .EXE programs. *See* Modifying the .EXE File Header below.

The registers

Figure 4-1 gives a general indication of how MS-DOS sets the 8086-family registers before transferring control to a .EXE program. MS-DOS determines most of the original register values from information the linker places in the .EXE file header at the start of the .EXE file.

MS-DOS sets the SS register to the segment (paragraph) address of the start of any segments declared with the STACK *combine* type and sets the SP register to the offset from SS of the byte immediately after the combined stack segments. (If no stack segment is declared, MS-DOS sets SS:SP to CS:0000.) Because in the 8086-family architecture a stack grows from high to low memory addresses, this effectively sets SS:SP to point to the base of the stack. Therefore, if the programmer declares stack segments when writing an assembly-language program, the program will not need to initialize the SS and SP registers. Microsoft's high-level-language compilers handle the creation of stack segments automatically. In both cases, the linker determines the initial SS and SP values and places them in the header at the start of the .EXE program file.

Unlike its handling of the SS and SP registers, MS-DOS does *not* initialize the DS and ES registers to any data areas of the .EXE program. Instead, it points DS and ES to the start of

the PSP. It does this for two primary reasons: First, MS-DOS uses the DS and ES registers to tell the program the address of the PSP; second, most programs start by examining the command tail within the PSP. Because the program starts without DS pointing to the data segments, the program must initialize DS and (optionally) ES to point to the data segments before it starts trying to access any fields in those segments. Unlike .COM programs, .EXE programs can do this easily because they can make direct references to segments, as follows:

```
MOV     AX,SEG DATA_SEGMENT_OR_GROUP_NAME
MOV     DS,AX
MOV     ES,AX
```

High-level-language programs need not initialize and maintain DS and ES; the compiler and library support routines do this.

In addition to pointing DS and ES to the PSP, MS-DOS also sets AH and AL to reflect the validity of the drive identifiers it placed in the two FCBs contained in the PSP. MS-DOS sets AL to 0FFH if the first FCB at PSP:005CH was initialized with a nonexistent drive identifier; otherwise, it sets AL to zero. Similarly, MS-DOS sets AH to reflect the drive identifier placed in the second FCB at PSP:006CH.

When MS-DOS analyzes the first two command-line parameters following the program name in order to build the first and second FCBs, it treats *any* character followed by a colon as a drive prefix. If the drive prefix consists of a lowercase letter (ASCII *a* through *z*), MS-DOS starts by converting the character to uppercase (ASCII *A* through *Z*). Then it subtracts 40H from the character, regardless of its original value. This converts the drive prefix letters A through Z to the drive codes 01H through 1AH, as required by the two FCBs. Finally, MS-DOS places the drive code in the appropriate FCB.

This process does not actually preclude invalid drive specifications from being placed in the FCBs. For instance, MS-DOS will accept the drive prefix !: and place a drive code of 0E1H in the FCB (! = 21H; 21H−40H = 0E1H). However, MS-DOS will then check the drive code to see if it represents an existing drive attached to the computer and will pass a value of 0FFH to the program in the appropriate register (AL or AH) if it does not.

As a side effect of this process, MS-DOS accepts @: as a valid drive prefix because the subtraction of 40H converts the @ character (40H) to 00H. MS-DOS accepts the 00H value as valid because a 00H drive code represents the current default drive. MS-DOS will leave the FCB's drive code set to 00H rather than translating it to the code for the default drive because the MS-DOS function calls that use FCBs accept the 00H code.

Finally, MS-DOS initializes the CS and IP registers, transferring control to the program's entry point. Programs developed using high-level-language compilers usually receive control at a library initialization routine. A programmer writing an assembly-language program using the Microsoft Macro Assembler (MASM) can declare any label within the

program as the entry point by placing the label after the END statement as the last line of the program:

```
END     ENTRY_POINT_LABEL
```

With multiple source files, only one of the files should have a label following the END statement. If more than one source file has such a label, LINK uses the first one it encounters as the entry point.

The other processor registers (BX, CX, DX, BP, SI, and DI) contain unknown values when the program receives control from MS-DOS. Once again, high-level-language programmers can ignore this fact — the compiler and library support routines deal with the situation. However, assembly-language programmers should keep this fact in mind. It may give needed insight sometime in the future when a program functions at certain times and not at others.

In many cases, debuggers such as DEBUG and SYMDEB initialize uninitialized registers to some predictable but undocumented state. For instance, some debuggers may predictably set BP to zero before starting program execution. However, a program must not rely on such debugger actions, because MS-DOS makes no such promises. Situations like this could account for a program that fails when executed directly under MS-DOS but works fine when executed using a debugger.

Terminating the .EXE program

After MS-DOS has given the .EXE program control and it has completed whatever task it set out to perform, the program needs to give control back to MS-DOS. Because of MS-DOS's evolution, five methods of program termination have accumulated — not including the several ways MS-DOS allows programs to terminate but remain resident in memory.

Before using any of the termination methods supported by MS-DOS, the program should always close any files it had open, especially those to which data has been written or whose lengths were changed. Under versions 2.0 and later, MS-DOS closes any files opened using handles. However, good programming practice dictates that the program not rely on the operating system to close the program's files. In addition, programs written to use shared files under MS-DOS versions 3.0 and later should release any file locks before closing the files and terminating.

The Terminate Process with Return Code function

Of the five ways a program can terminate, only the Interrupt 21H Terminate Process with Return Code function (4CH) is recommended for programs running under MS-DOS version 2.0 or later. This method is one of the easiest approaches to terminating *any* program, regardless of its structure or segment register settings. The Terminate Process with Return Code function call simply consists of the following:

```
MOV     AH,4CH                  ;load the MS-DOS function code
MOV     AL,RETURN_CODE          ;load the termination code
INT     21H                     ;call MS-DOS to terminate program
```

The example loads the AH register with the Terminate Process with Return Code function code. Then it loads the AL register with a return code. Normally, the return code represents the reason the program terminated or the result of any operation the program performed.

A program that executes another program as a child process can recover and analyze the child program's return code if the child process used this termination method. Likewise, the child process can recover the RETURN_CODE returned by any program it executes as a child process. When a program is terminated using this method and control returns to MS-DOS, a batch (.BAT) file can be used to test the terminated program's return code using the *IF ERRORLEVEL* statement.

Only two general conventions have been adopted for the value of RETURN_CODE: First, a RETURN_CODE value of 00H indicates a normal no-error termination of the program; second, increasing RETURN_CODE values indicate increasing severity of conditions under which the program terminated. For instance, a compiler could use the RETURN_CODE 00H if it found no errors in the source file, 01H if it found only warning errors, or 02H if it found severe errors.

If a program has no need to return any special RETURN_CODE values, then the following instructions will suffice to terminate the program with a RETURN_CODE of 00H:

```
MOV    AX,4C00H
INT    21H
```

Apart from being the approved termination method, Terminate Process with Return Code is easier to use with .EXE programs than any other termination method because all other methods require that the CS register point to the start of the PSP when the program terminates. This restriction causes problems for .EXE programs because they have code segments with segment addresses different from that of the PSP.

The only problem with Terminate Process with Return Code is that it is not available under MS-DOS versions earlier than 2.0, so it cannot be used if a program must be compatible with early MS-DOS versions. However, Figure 4-3 shows how a program can use the approved termination method when available but still remain pre-2.0 compatible. *See* The Warm Boot/Terminate Vector below.

```
TEXT    SEGMENT PARA PUBLIC 'CODE'

        ASSUME  CS:TEXT,DS:NOTHING,ES:NOTHING,SS:NOTHING

TERM_VECTOR     DD      ?

ENTRY_PROC      PROC    FAR

;save pointer to termination vector in PSP

        MOV     WORD PTR CS:TERM_VECTOR+0,0000h  ;save offset of Warm Boot vector
        MOV     WORD PTR CS:TERM_VECTOR+2,DS     ;save segment address of PSP
```

Figure 4-3. Terminating properly under any MS-DOS version.   *(more)*

```
;***** Place main task here *****

;determine which MS-DOS version is active, take jump if 2.0 or later

        MOV     AH,30h          ;load Get MS-DOS Version Number function code
        INT     21h             ;call MS-DOS to get version number
        OR      AL,AL           ;see if pre-2.0 MS-DOS
        JNZ     TERM_0200       ;jump if 2.0 or later

;terminate under pre-2.0 MS-DOS

        JMP     CS:TERM_VECTOR  ;jump to Warm Boot vector in PSP

;terminate under MS-DOS 2.0 or later

TERM_0200:
        MOV     AX,4C00h        ;load MS-DOS termination function code
                                ;and return code
        INT     21h             ;call MS-DOS to terminate

ENTRY_PROC      ENDP

TEXT    ENDS

        END     ENTRY_PROC      ;define entry point
```

Figure 4-3. Continued.

The Terminate Program interrupt

Before MS-DOS version 2.0, terminating with an approved method meant executing
an INT 20H instruction, the Terminate Program interrupt. The INT 20H instruction was
replaced as the approved termination method for two primary reasons: First, it did not
provide a means whereby programs could return a termination code; second, CS had
to point to the PSP before the INT 20H instruction was executed.

The restriction placed on the value of CS at termination did not pose a problem for .COM
programs because they execute with CS pointing to the beginning of the PSP. A .EXE pro-
gram, on the other hand, executes with CS pointing to various code segments of the pro-
gram, and the value of CS cannot be changed arbitrarily when the program is ready to
terminate. Because of this, few .EXE programs attempt simply to execute a Terminate Pro-
gram interrupt from directly within their own code segments. Instead, they usually use
the termination method discussed next.

The Warm Boot/Terminate vector

The earlier discussion of the structure of the PSP briefly covered one older method a .EXE
program can use to terminate: Offset 00H within the PSP contains an INT 20H instruction
to which the program can jump in order to terminate. MS-DOS adopted this technique to
support the many CP/M programs ported to MS-DOS. Under CP/M, this PSP location was
referred to as the Warm Boot vector because the CP/M operating system was always
reloaded from disk (rebooted) whenever a program terminated.

Because offset 00H in the PSP contains an INT 20H instruction, jumping to that location terminates a program in the same manner as an INT 20H included directly within the program, but with one important difference: By jumping to PSP:0000H, the program sets the CS register to point to the beginning of the PSP, thereby satisfying the only restriction imposed on executing the Terminate Program interrupt. The discussion of MS-DOS Function 4CH gave an example of how a .EXE program can terminate via PSP:0000H. The example first asks MS-DOS for its version number and then terminates via PSP:0000H only under versions of MS-DOS earlier than 2.0. Programs can also use PSP:0000H under MS-DOS versions 2.0 and later; the example uses Function 4CH simply because it is preferred under the later MS-DOS versions.

The RET instruction

The other popular method used by CP/M programs to terminate involved simply executing a RET instruction. This worked because CP/M pushed the address of the Warm Boot vector onto the stack before giving the program control. MS-DOS provides this support only for .COM-style programs; it does *not* push a termination address onto the stack before giving .EXE programs control.

The programmer who wants to use the RET instruction to return to MS-DOS can use the variation of the Figure 4-3 listing shown in Figure 4-4.

```
TEXT     SEGMENT PARA PUBLIC 'CODE'

         ASSUME  CS:TEXT,DS:NOTHING,ES:NOTHING,SS:NOTHING

ENTRY_PROC       PROC    FAR     ;make proc FAR so RET will be FAR

;Push pointer to termination vector in PSP
         PUSH    DS              ;push PSP's segment address
         XOR     AX,AX           ;ax = 0 = offset of Warm Boot vector in PSP
         PUSH    AX              ;push Warm Boot vector offset

;***** Place main task here *****

;Determine which MS-DOS version is active, take jump if 2.0 or later

         MOV     AH,30h          ;load Get MS-DOS Version Number function code
         INT     21h             ;call MS-DOS to get version number
         OR      AL,AL           ;see if pre-2.0 MS-DOS
         JNZ     TERM_0200       ;jump if 2.0 or later

;Terminate under pre-2.0 MS-DOS (this is a FAR proc, so RET will be FAR)
         RET                     ;pop PSP:00H into CS:IP to terminate
```

Figure 4-4. Using RET to return control to MS-DOS.   (more)

```
;Terminate under MS-DOS 2.0 or later
TERM_0200:
        MOV     AX,4C00h        ;AH = MS-DOS Terminate Process with Return Code
                                ;function code, AL = return code of 00H
        INT     21h             ;call MS-DOS to terminate


ENTRY_PROC      ENDP


TEXT    ENDS


        END     ENTRY_PROC      ;declare the program's entry point
```

Figure 4-4. Continued.

The Terminate Process function

The final method for terminating a .EXE program is Interrupt 21H Function 00H (Terminate Process). This method maintains the same restriction as all other older termination
methods: CS must point to the PSP. Because of this restriction, .EXE programs typically
avoid this method in favor of terminating via PSP:0000H, as discussed above for programs
executing under versions of MS-DOS earlier than 2.0.

Terminating and staying resident

A .EXE program can use any of several additional termination methods to return control to MS-DOS but still remain resident within memory to service a special event. *See*
PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-
Stay-Resident Utilities.

Structure of the .EXE files

So far we've examined how the .EXE program looks in memory, how MS-DOS gives the
program control of the computer, and how the program should return control to MS-DOS.
Next we'll investigate what the program looks like as a disk file, before MS-DOS loads it
into memory. Figure 4-5 shows the general structure of a .EXE file.

The file header

Unlike .COM program files, .EXE program files contain information that permits the
.EXE program and MS-DOS to use the full capabilities of the 8086 family of microprocessors. The linker places all this extra information in a header at the start of the .EXE file.
Although the .EXE file structure could easily accommodate a header as small as 32 bytes,
the linker never creates a header smaller than 512 bytes. (This minimum header size corresponds to the standard record size preferred by MS-DOS.) The .EXE file header contains
the following information, which MS-DOS reads into a temporary work area in memory
for use while loading the .EXE program:

00–01H (.EXE Signature) MS-DOS does not rely on the extension (.EXE or .COM) to
determine whether a file contains a .COM or a .EXE program. Instead, MS-DOS recognizes
the file as a .EXE program if the first 2 bytes in the header contain the signature 4DH 5AH

	x0H	x1H	x2H	x3H	x4H	x5H	x6H	x7H	x8H	x9H	xAH	xBH	xCH	xDH	xEH	xFH
0xH ▶	Signature 4DH	5AH	Last Page Size lo byt	hi byt	File Pages lo byt	hi byt	Reloc Items lo byt	hi byt	Header Paras lo byt	hi byt	MINALLOC lo byt	hi byt	MAXALLOC lo byt	hi byt	PreReloc SS lo byt	hi byt
1xH ▶	Initial SP ofs lo	ofs hi	Neg Chksum lo byt	hi byt	Initial IP ofs lo	ofs hi	Pre Reloc CS seg lo	seg hi	Reloc Tbl Ofs lo byt	hi byt	Overlay Num lo byt	hi byt	Reserved			

Use Reloc Tbl Ofs at 18H (offset is from start of file) ▶

Seg Relocation Ptr #1 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #2 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #3 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #4 ofs lo|ofs hi|seg lo|seg hi

Seg Relocation Ptr #n-3 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #n-2 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #n-1 ofs lo|ofs hi|seg lo|seg hi	Seg Relocation Ptr #n ofs lo|ofs hi|seg lo|seg hi

Use Reloc Items at 06H ◀

Use Header Paras at 08H (load module always starts on paragraph boundary) ▶

Program image (load module)   Use Last Page Size at 02H   Final 512-byte page as indicated by File Pages at 04H

End of file ▶

Figure 4-5. Structure of a .EXE file.

(ASCII characters *M* and *Z*). If either or both of the signature bytes contain other values, MS-DOS assumes the file contains a .COM program, regardless of the extension. The reverse is not necessarily true — that is, MS-DOS does not accept the file as a .EXE program simply because the file begins with a .EXE signature. The file must also pass several other tests.

02–03H (Last Page Size) The word at this location indicates the actual number of bytes in the final 512-byte page of the file. This word combines with the following word to determine the actual size of the file.

04–05H (File Pages) This word contains a count of the total number of 512-byte pages required to hold the file. If the file contains 1024 bytes, this word contains the value 0002H; if the file contains 1025 bytes, this word contains the value 0003H. The previous word (Last Page Size, 02–03H) is used to determine the number of valid bytes in the final 512-byte page. Thus, if the file contains 1024 bytes, the Last Page Size word contains 0000H because no bytes overflow into a final partly used page; if the file contains 1025 bytes, the Last Page Size word contains 0001H because the final page contains only a single valid byte (the 1025th byte).

06–07H (Relocation Items) This word gives the number of entries that exist in the relocation pointer table. *See* Relocation Pointer Table below.

08–09H (Header Paragraphs) This word gives the size of the .EXE file header in 16-byte paragraphs. It indicates the offset of the program's compiled/assembled and linked image (the load module) within the .EXE file. Subtracting this word from the two file-size words starting at 02H and 04H reveals the size of the program's image. The header always spans an even multiple of 16-byte paragraphs. For example, if the file consists of a 512-byte header and a 513-byte program image, then the file's total size is 1025 bytes. As discussed before, the Last Page Size word (02–03H) will contain 0001H and the File Pages word (04–05H) will contain 0003H. Because the header is 512 bytes, the Header Paragraphs word (08–09H) will contain 32 (0020H). (That is, 32 paragraphs times 16 bytes per paragraph totals 512 bytes.) By subtracting the 512 bytes of the header from the 1025-byte total file size, the size of the program's image can be determined — in this case, 513 bytes.

0A–0BH (MINALLOC) This word indicates the minimum number of 16-byte paragraphs the program requires to begin execution *in addition to* the memory required to hold the program's image. MINALLOC normally represents the total size of any uninitialized data and/or stack segments linked at the end of the program. LINK excludes the space reserved by these fields from the end of the .EXE file to avoid wasting disk space. If not enough memory remains to satisfy MINALLOC when loading the program, MS-DOS returns an error to the process trying to load the program. If the process is COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message. The EXEMOD utility can alter this field if desired. *See* Modifying the .EXE File Header below.

0C–0DH (MAXALLOC) This word indicates the maximum number of 16-byte paragraphs the program would like allocated to it before it begins execution. MAXALLOC indicates *additional* memory desired beyond that required to hold the program's image. MS-DOS uses this value to allocate MAXALLOC extra paragraphs, if available. If MAXALLOC paragraphs are not available, the program receives the largest memory block available — at least MINALLOC additional paragraphs. The programmer could use the MAXALLOC field to request that MS-DOS allocate space for use as a print buffer or as a program-maintained heap, for example.

Unless otherwise specified with the /CPARMAXALLOC switch at link time, the linker sets MAXALLOC to FFFFH. This causes MS-DOS to allocate all of the largest block of memory it has available to the program. To make the program compatible with multitasking supervisor programs, the programmer should use /CPARMAXALLOC to set the true maximum number of extra paragraphs the program desires. The EXEMOD utility can also be used to alter this field.

Note: If both MINALLOC and MAXALLOC have been set to 0000H, MS-DOS loads the program as high in memory as possible. LINK sets these fields to 0000H if the /HIGH switch was used; the EXEMOD utility can also be used to modify these fields.

0E–0FH (Initial SS Value) This word contains the paragraph address of the stack segment relative to the start of the load module. At load time, MS-DOS relocates this value by adding the program's start segment address to it, and the resulting value is placed in the SS register before giving the program control. (The start segment corresponds to the first segment boundary in memory following the PSP.)

10–11H (Initial SP Value) This word contains the absolute value that MS-DOS loads into the SP register before giving the program control. Because MS-DOS always loads programs starting on a segment address boundary, and because the linker knows the size of the stack segment, the linker is able to determine the correct SP offset at link time; therefore, MS-DOS does not need to adjust this value at load time. The EXEMOD utility can be used to alter this field.

12–13H (Complemented Checksum) This word contains the one's complement of the summation of all words in the .EXE file. Current versions of MS-DOS basically ignore this word when they load a .EXE program; however, future versions might not. When LINK generates a .EXE file, it adds together all the contents of the .EXE file (including the .EXE header) by treating the entire file as a long sequence of 16-bit words. During this addition, LINK gives the Complemented Checksum word (12–13H) a temporary value of 0000H. If the file consists of an odd number of bytes, then the final byte is treated as a word with a high byte of 00H. Once LINK has totaled all words in the .EXE file, it performs a one's complement operation on the total and records the answer in the .EXE file header at offsets 12–13H. The validity of a .EXE file can then be checked by performing the same word-totaling process as LINK performed. The total should be FFFFH, because the total will include LINK's calculated complemented checksum, which is designed to give the file the FFFFH total.

An example 7-byte .EXE file illustrates how .EXE file checksums are calculated. (This is a totally fictitious file, because .EXE headers are never smaller than 512 bytes.) If this fictitious file contained the bytes 8CH C8H 8EH D8H BAH 10H B4H, then the file's total would be calculated using C88CH+D88EH+10BAH+00B4H=1B288H. (Overflow past 16 bits is ignored, so the value is interpreted as B288H.) If this were a valid .EXE file, then the B288H total would have been FFFFH instead.

14–15H (Initial IP Value) This word contains the absolute value that MS-DOS loads into the IP register in order to transfer control to the program. Because MS-DOS always loads programs starting on a segment address boundary, the linker can calculate the correct IP offset from the initial CS register value at link time; therefore, MS-DOS does not need to adjust this value at load time.

16–17H (Pre-Relocated Initial CS Value) This word contains the initial value, relative to the start of the load module, that MS-DOS places in the CS register to give the .EXE program control. MS-DOS adjusts this value in the same manner as the initial SS value before loading it into the CS register.

18–19H (Relocation Table Offset) This word gives the offset from the start of the file to the relocation pointer table. This word must be used to locate the relocation pointer table, because variable-length information pertaining to program overlays can occur before the table, thus causing the position of the table to vary.

1A–1BH (Overlay Number) This word is normally set to 0000H, indicating that the .EXE file consists of the resident, or primary, part of the program. This number changes only in files containing programs that use overlays, which are sections of a program that remain

on disk until the program actually requires them. These program sections are loaded into memory by special overlay managing routines included in the run-time libraries supplied with some Microsoft high-level-language compilers.

The preceding section of the header (00–1BH) is known as the formatted area. Optional information used by high-level-language overlay managers can follow this formatted area. Unless the program in the .EXE file incorporates such information, the relocation pointer table immediately follows the formatted header area.

Relocation Pointer Table The relocation pointer table consists of a list of pointers to words within the .EXE program image that MS-DOS must adjust before giving the program control. These words consist of references made by the program to the segments that make up the program. MS-DOS must adjust these segment address references when it loads the program, because it can load the program into memory starting at any segment address boundary.

Each pointer in the table consists of a doubleword. The first word contains an offset from the segment address given in the second word, which in turn indicates a segment address relative to the start of the load module. Together, these two words point to a third word within the load module that must have the start segment address added to it. (The start segment corresponds to the segment address at which MS-DOS started loading the program's
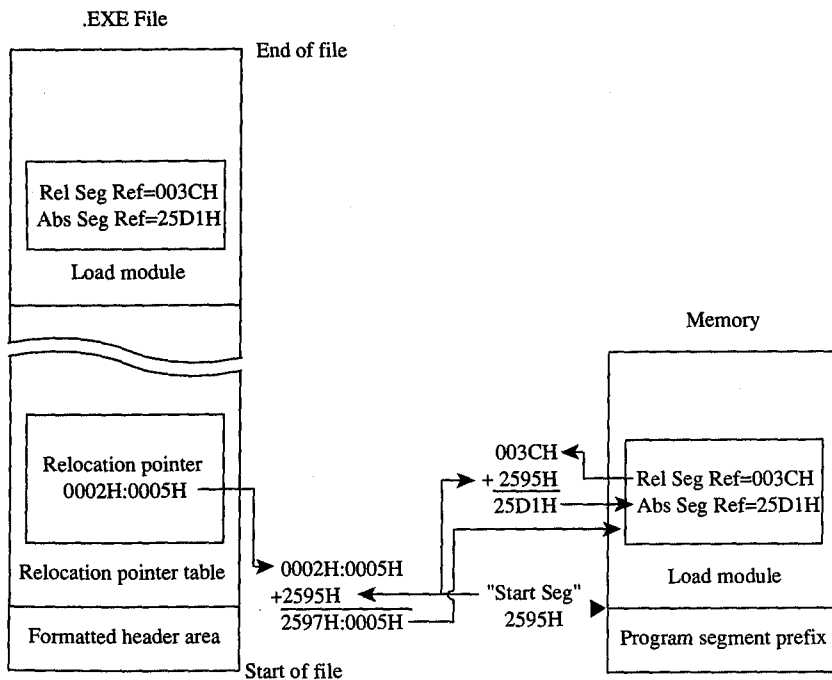
Figure 4-6. The .EXE file relocation procedure.

image, immediately following the PSP.) Figure 4-6 shows the entire procedure MS-DOS performs for each relocation table entry.

The load module

The load module starts where the .EXE header ends and consists of the fully linked image of the program. The load module appears within the .EXE file exactly as it would appear in memory if MS-DOS were to load it at segment address 0000H. The only changes MS-DOS makes to the load module involve relocating any direct segment references.

Although the .EXE file contains distinct segment images within the load module, it provides no information for separating those individual segments from one another. Existing versions of MS-DOS ignore how the program is segmented; they simply copy the load module into memory, relocate any direct segment references, and give the program control.

Loading the .EXE program

So far we've covered all the characteristics of the .EXE program as it resides in memory and on disk. We've also touched on all the steps MS-DOS performs while loading the .EXE program from disk and executing it. The following list recaps the .EXE program loading process in the order in which MS-DOS performs it:

1. MS-DOS reads the formatted area of the header (the first 1BH bytes) from the .EXE file into a work area.
2. MS-DOS determines the size of the largest available block of memory.
3. MS-DOS determines the size of the load module using the Last Page Size (offset 02H), File Pages (offset 04H), and Header Paragraphs (offset 08H) fields from the header. An example of this process is in the discussion of the Header Paragraphs field.
4. MS-DOS adds the MINALLOC field (offset 0AH) in the header to the calculated load-module size and the size of the PSP (100H bytes). If this total exceeds the size of the largest available block, MS-DOS terminates the load process and returns an error to the calling process. If the calling process was COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message.
5. MS-DOS adds the MAXALLOC field (offset 0CH) in the header to the calculated load-module size and the size of the PSP. If the memory block found earlier exceeds this calculated total, MS-DOS allocates the calculated memory size to the program from the memory block; if the calculated total exceeds the block's size, MS-DOS allocates the entire block.
6. If the MINALLOC and MAXALLOC fields both contain 0000H, MS-DOS uses the calculated load-module size to determine a start segment. MS-DOS calculates the start segment so that the load module will load into the high end of the allocated block. If either MINALLOC or MAXALLOC contains nonzero values (the normal case), MS-DOS establishes the start segment as the segment following the PSP.
7. MS-DOS loads the load module into memory starting at the start segment.

8. MS-DOS reads the relocation pointers into a work area and relocates the load module's direct segment references, as shown in Figure 4-6.
9. MS-DOS builds a PSP in the first 100H bytes of the allocated memory block. While building the two FCBs within the PSP, MS-DOS determines the initial values for the AL and AH registers.
10. MS-DOS sets the SS and SP registers to the values in the header after the start segment is added to the SS value.
11. MS-DOS sets the DS and ES registers to point to the beginning of the PSP.
12. MS-DOS transfers control to the .EXE program by setting CS and IP to the values in the header after adding the start segment to the CS value.

Controlling the .EXE program's structure

We've now covered almost every aspect of a completed .EXE program. Next, we'll discuss how to control the structure of the final .EXE program from the source level. We'll start by covering the statements provided by MASM that permit the programmer to define the structure of the program when programming in assembly language. Then we'll cover the five standard memory models provided by Microsoft's C and FORTRAN compilers (both version 4.0), which provide predefined structuring over which the programmer has limited control.

The MASM SEGMENT directive

MASM's SEGMENT directive and its associated ENDS directive mark the beginning and end of a program segment. Program segments contain collections of code or data that have offset addresses relative to the same common segment address.

In addition to the required segment name, the SEGMENT directive has three optional parameters:

segname SEGMENT [*align*] [*combine*] ['*class*']

With MASM, the contents of a segment can be defined at one point in the source file and the definition can be resumed as many times as necessary throughout the remainder of the file. When MASM encounters a SEGMENT directive with a *segname* it has previously encountered, it simply resumes the segment definition where it left off. This occurs regardless of the *combine* type specified in the SEGMENT directive — the *combine* type influences only the actions of the linker. *See* The *combine* Type Parameter below.

The *align* type parameter

The optional *align* parameter lets the programmer send the linker an instruction on how to align a segment within memory. In reality, the linker can align the segment only in relation to the start of the program's load module, but the result remains the same because MS-DOS always loads the module aligned on a paragraph (16-byte) boundary. (The PAGE *align* type creates a special exception, as discussed below.)

The following alignment types are permitted:

BYTE This *align* type instructs the linker to start the segment on the byte immediately following the previous segment. BYTE alignment prevents any wasted memory between the previous segment and the BYTE-aligned segment.

A minor disadvantage to BYTE alignment is that the 8086-family segment registers might not be able to directly address the start of the segment in all cases. Because they can address only on paragraph boundaries, the segment registers may have to point as many as 15 bytes behind the start of the segment. This means that the segment size should not be more than 15 bytes short of 64 KB. The linker adjusts offset and segment address references to compensate for differences between the physical segment start and the paragraph addressing boundary.

Another possible concern is execution speed on true 16-bit 8086-family microprocessors. When using non-8088 microprocessors, a program can actually run faster if the instructions and word data fields within segments are aligned on word boundaries. This permits the 16-bit processors to fetch full words in a single memory read, rather than having to perform two single-byte reads. The EVEN directive tells MASM to align instructions and data fields on word boundaries; however, MASM can establish this alignment only in relation to the start of the segment, so the entire segment must start aligned on a word or larger boundary to guarantee alignment of the items within the segment.

WORD This *align* type instructs the linker to start the segment on the next word boundary. Word boundaries occur every 2 bytes and consist of all even addresses (addresses in which the least significant bit contains a zero). WORD alignment permits alignment of data fields and instructions within the segment on word boundaries, as discussed for the BYTE alignment type. However, the linker may have to waste 1 byte of memory between the previous segment and the word-aligned segment in order to position the new segment on a word boundary.

Another minor disadvantage to WORD alignment is that the 8086-family segment registers might not be able to directly address the start of the segment in all cases. Because they can address only on paragraph boundaries, the segment registers may have to point as many as 14 bytes behind the start of the segment. This means that the segment size should not be more than 14 bytes short of 64 KB. The linker adjusts offset and segment address references to compensate for differences between the physical segment start and the paragraph addressing boundary.

PARA This *align* type instructs the linker to start the segment on the next paragraph boundary. The segments default to PARA if no alignment type is specified. Paragraph boundaries occur every 16 bytes and consist of all addresses with hexadecimal values ending in zero (0000H, 0010H, 0020H, and so forth). Paragraph alignment ensures that the segment begins on a segment register addressing boundary, thus making it possible to address a full 64 KB segment. Also, because paragraph addresses are even addresses, PARA alignment has the same advantages as WORD alignment. The only real disadvantage to PARA alignment is that the linker may have to waste as many as 15 bytes of memory between the previous segment and the paragraph-aligned segment.

PAGE This *align* type instructs the linker to start the segment on the next page boundary. Page boundaries occur every 256 bytes and consist of all addresses in which the low address byte equals zero (0000H, 0100H, 0200H, and so forth). PAGE alignment ensures

only that the linker positions the segment on a page boundary relative to the start of the load module. Unfortunately, this does not also ensure alignment of the segment on an absolute page within memory, because MS-DOS only guarantees alignment of the entire load module on a paragraph boundary.

When a programmer declares pieces of a segment with the same name in different source modules, the *align* type specified for each segment piece influences the alignment of that specific piece of the segment. For example, assume the following two segment declarations appear in different source modules:

```
_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '123'
_DATA   ENDS

_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '456'
_DATA   ENDS
```

The linker starts by aligning the first segment piece located in the first object module on a paragraph boundary, as requested. When the linker encounters the second segment piece in the second object module, it aligns that piece on the first paragraph boundary following the first segment piece. This results in a 13-byte gap between the first segment piece and the second. The segment pieces must exist in separate source modules for this to occur. If the segment pieces exist in the same source module, MASM assumes that the second segment declaration is simply a resumption of the first and creates an object module with segment declarations equivalent to the following:

```
_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '123'
        DB      '456'
_DATA   ENDS
```

The *combine* type parameter

The optional *combine* parameter allows the programmer to send directions to the linker on how to combine segments with the same *segname* occurring in different object modules. If no *combine* type is specified, the linker treats such segments as if each had a different *segname*. The *combine* type has no effect on the relationship of segments with different *segnames*. MASM and LINK both support the following *combine* types:

PUBLIC This *combine* type instructs the linker to concatenate multiple segments having the same *segname* into a single contiguous segment. The linker adjusts any address references to labels within the concatenated segments to reflect the new position of those labels relative to the start of the combined segment. This *combine* type is useful for accessing code or data in different source modules using a common segment register value.

STACK This *combine* type operates similarly to the PUBLIC *combine* type, except for two additional effects: The STACK type tells the linker that this segment comprises part of the program's stack and initialization data contained within STACK segments is handled differently than in PUBLIC segments. Declaring segments with the STACK *combine* type permits the linker to determine the initial SS and SP register values it places in the .EXE

file header. Normally, a programmer would declare only one STACK segment in one of the source modules. If pieces of the stack are declared in different source modules, the linker will concatenate them in the same fashion as PUBLIC segments. However, initialization data declared within any STACK segment is placed at the high end of the combined STACK segments on a module-by-module basis. Thus, each successive module's initialization data overlays the previous module's data. At least one segment must be declared with the STACK *combine* type; otherwise, the linker will issue a warning message because it cannot determine the program's initial SS and SP values. (The warning can be ignored if the program itself initializes SS and SP.)

COMMON This *combine* type instructs the linker to overlap multiple segments having the same *segname*. The length of the resulting segment reflects the length of the longest segment declared. If any code or data is declared in the overlapping segments, the data contained in the final segments linked replaces any data in previously loaded segments. This *combine* type is useful when a data area is to be shared by code in different source modules.

MEMORY Microsoft's LINK treats this *combine* type the same as it treats the PUBLIC type. MASM, however, supports the MEMORY type for compatibility with other linkers that use Intel's definition of a MEMORY *combine* type.

AT address This *combine* type instructs LINK to pretend that the segment will reside at the absolute segment *address*. LINK then adjusts all address references to the segment in accordance with the masquerade. LINK will *not* create an image of the segment in the load module, and it will ignore any data defined within the segment. This behavior is consistent with the fact that MS-DOS does not support the loading of program segments into absolute memory segments. All programs must be able to execute from any segment address at which MS-DOS can find available memory. The SEGMENT AT address *combine* type is useful for creating templates of various areas in memory outside the program. For instance, *SEGMENT AT 0000H* could be used to create a template of the 8086-family interrupt vectors. Because data contained within SEGMENT AT address segments is suppressed by LINK and not by MASM (which places the data in the object module), it is possible to use .OBJ files generated by MASM with another linker that supports ROM or other absolute code generation should the programmer require this specialized capability.

The *class* type parameter
The *class* parameter provides the means to organize different segments into classifications. For instance, here are three source modules, each with its own separate code and data segments:

```
;Module "A"
A_DATA   SEGMENT PARA PUBLIC 'DATA'
;Module "A" data fields
A_DATA   ENDS
A_CODE   SEGMENT PARA PUBLIC 'CODE'
;Module "A" code
A_CODE   ENDS
         END
```

(more)

```
;Module "B"
B_DATA   SEGMENT PARA PUBLIC 'DATA'
;Module "B" data fields
B_DATA   ENDS
B_CODE   SEGMENT PARA PUBLIC 'CODE'
;Module "B" code
B_CODE   ENDS
         END

;Module "C"
C_DATA   SEGMENT PARA PUBLIC 'DATA'
;Module "C" data fields
C_DATA   ENDS
C_CODE   SEGMENT PARA PUBLIC 'CODE'
;Module "C" code
C_CODE   ENDS
         END
```

If the 'CODE' and 'DATA' *class* types are removed from the SEGMENT directives shown above, the linker organizes the segments as it encounters them. If the programmer specifies the modules to the linker in alphabetic order, the linker produces the following segment ordering:

```
A_DATA
A_CODE
B_DATA
B_CODE
C_DATA
C_CODE
```

However, if the programmer specifies the *class* types shown in the sample source modules, the linker organizes the segments by classification as follows:

```
'DATA' class:    A_DATA
                 B_DATA
                 C_DATA

'CODE' class:    A_CODE
                 B_CODE
                 C_CODE
```

Notice that the linker still organizes the classifications in the order in which it encounters the segments belonging to the various classifications. To completely control the order in which the linker organizes the segments, the programmer must use one of three basic approaches. The preferred method involves using the /DOSSEG switch with the linker. This produces the segment ordering shown in Figure 4-1. The second method involves creating a special source module that contains empty SEGMENT–ENDS blocks for all the segments declared in the various other source modules. The programmer creates the list in the order the segments are to be arranged in memory and then specifies the .OBJ file for this module as the first file for the linker to process. This procedure establishes the order of all the segments before LINK begins processing the other program modules, so the

programmer can declare segments in these other modules in any convenient order. For instance, the following source module rearranges the result of the previous example so that the linker places the 'CODE' class before the 'DATA' class:

```
A_CODE   SEGMENT PARA PUBLIC 'CODE'
A_CODE   ENDS
B_CODE   SEGMENT PARA PUBLIC 'CODE'
B_CODE   ENDS
C_CODE   SEGMENT PARA PUBLIC 'CODE'
C_CODE   ENDS

A_DATA   SEGMENT PARA PUBLIC 'DATA'
A_DATA   ENDS
B_DATA   SEGMENT PARA PUBLIC 'DATA'
B_DATA   ENDS
C_DATA   SEGMENT PARA PUBLIC 'DATA'
C_DATA   ENDS

         END
```

Rather than creating a new module, the third method places the same segment ordering list shown above at the start of the first module containing actual code or data that the programmer will be specifying for the linker. This duplicates the approach used by Microsoft's newer compilers, such as C version 4.0.

The ordering of segments within the load module has no direct effect on the linker's adjustment of address references to locations within the various segments. Only the GROUP directive and the SEGMENT directive's *combine* parameter affect address adjustments performed by the linker. *See* The MASM GROUP Directive below.

Note: Certain older versions of the IBM Macro Assembler wrote segments to the object file in alphabetic order regardless of their order in the source file. These older versions can limit efforts to control segment ordering. Upgrading to a new version of the assembler is the best solution to this problem.

Ordering segments to shrink the .EXE file

Correct segment ordering can significantly decrease the size of a .EXE program as it resides on disk. This size-reduction ordering is achieved by placing all uninitialized data fields in their own segments and then controlling the linker's ordering of the program's segments so that the uninitialized data field segments all reside at the end of the program. When the program modules are assembled, MASM places information in the object modules to tell the linker about initialized and uninitialized areas of all segments. The linker then uses this information to prevent the writing of uninitialized data areas that occur at the end of the program image as part of the resulting .EXE file. To account for the memory space required by these fields, the linker also sets the MINALLOC field in the .EXE file header to represent the data area not written to the file. MS-DOS then uses the MINALLOC field to reallocate this missing space when loading the program.

ZTE (USA) 1007, Page 142

The MASM GROUP directive

The MASM GROUP directive can also have a strong impact on a .EXE program. However, the GROUP directive has *no* effect on the arrangement of program segments within memory. Rather, GROUP associates program segments for addressing purposes.

The GROUP directive has the following syntax:

grpname GROUP *segname,segname,segname,...*

This directive causes the linker to adjust all address references to labels within any specified *segname* to be relative to the start of the declared group. The start of the group is determined at link time. The group starts with whichever of the segments in the GROUP list the linker places lowest in memory.

That the GROUP directive neither causes nor requires contiguous arrangement of the grouped segments creates some interesting, although not necessarily desirable, possibilities. For instance, it permits the programmer to locate segments not belonging to the declared group between segments that do belong to the group. The only restriction imposed on the declared group is that the last byte of the last segment in the group must occur within 64 KB of the start of the group. Figure 4-7 illustrates this type of segment arrangement:
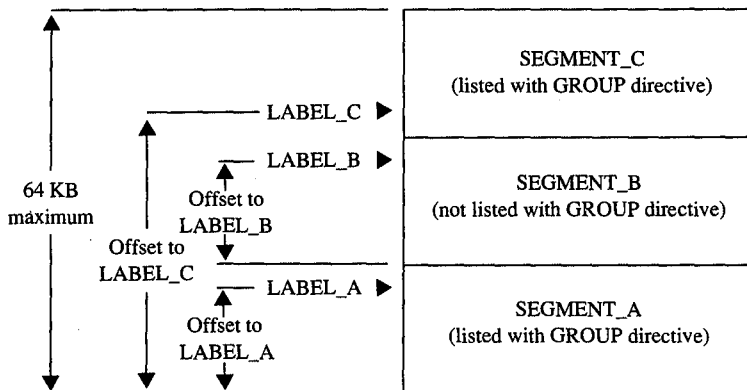
Figure 4-7. Noncontiguous segments in the same GROUP.

Warning: One of the most confusing aspects of the GROUP directive relates to MASM's OFFSET operator. The GROUP directive affects only the offset addresses generated by such direct addressing instructions as

```
MOV     AX,FIELD_LABEL
```

but it has no effect on immediate address values generated by such instructions as

```
MOV     AX,OFFSET FIELD_LABEL
```

Using the OFFSET operator on labels contained within grouped segments requires the following approach:

```
MOV     AX,OFFSET GROUP_NAME:FIELD_LABEL
```

The programmer must *explicitly* request the offset from the group base, because MASM defines the result of the OFFSET operator to be the offset of the label from the start of its segment, not its group.

Structuring a small program with SEGMENT and GROUP

Now that we have analyzed the functions performed by the SEGMENT and GROUP directives, we'll put both directives to work structuring a skeleton program. The program, shown in Figures 4-8, 4-9, and 4-10, consists of three source modules (MODULE_A, MODULE_B, and MODULE_C), each using the following four program segments:

Segment	Definition
_TEXT	The code or program text segment
_DATA	The standard data segment containing preinitialized data fields the program might change
CONST	The constant data segment containing constant data fields the program will not change
_BSS	The "block storage segment/space" segment containing uninitialized data fields*

* Programmers familiar with the IBM 1620/1630 or CDC 6000 and Cyber assemblers may recognize BSS as "block started at symbol," which reflects an equally appropriate, although somewhat more elaborate, definition of the abbreviation. Other common translations of BSS, such as "blank static storage," misrepresent the segment name, because blanking of BSS segments does not occur — the memory contains undetermined values when the program begins execution.

```
;Source Module MODULE_A

;Predeclare all segments to force the linker's segment ordering **************

_TEXT   SEGMENT BYTE PUBLIC 'CODE'
_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS

CONST   SEGMENT WORD PUBLIC 'CONST'
CONST   ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS
```

Figure 4-8. Structuring a .EXE program: MODULE_A. (more)

```
STACK    SEGMENT PARA STACK 'STACK'
STACK    ENDS

DGROUP   GROUP   _DATA,CONST,_BSS,STACK


;Constant declarations *****************************************************

CONST    SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_A    DB       'Constant A'     ;declare a MODULE_A constant

CONST    ENDS


;Preinitialized data fields ***********************************************

_DATA    SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_A     DB       'Data A'         ;declare a MODULE_A preinitialized field

_DATA    ENDS


;Uninitialized data fields ************************************************

_BSS     SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_A      DB       5 DUP(?)         ;declare a MODULE_A uninitialized field

_BSS     ENDS


;Program text *************************************************************

_TEXT    SEGMENT BYTE PUBLIC 'CODE'

         ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING

         EXTRN   PROC_B:NEAR              ;label is in _TEXT segment (NEAR)
         EXTRN   PROC_C:NEAR              ;label is in _TEXT segment (NEAR)

PROC_A   PROC    NEAR

         CALL    PROC_B                   ;call into MODULE_B
         CALL    PROC_C                   ;call into MODULE_C

         MOV     AX,4C00H                 ;terminate (MS-DOS 2.0 or later only)
         INT     21H

PROC_A   ENDP

_TEXT    ENDS
```

Figure 4-8. Continued.   *(more)*

```
;Stack *********************************************************************

STACK    SEGMENT PARA STACK 'STACK'

         DW      128 DUP(?)        .        ;declare some space to use as stack
STACK_BASE        LABEL   WORD              .

STACK    ENDS

         END     PROC_A                     ;declare PROC_A as entry point
```

Figure 4-8. Continued.

```
;Source Module MODULE_B

;Constant declarations ***************************************************

CONST    SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_B    DB       'Constant B'    ;declare a MODULE_B constant

CONST    ENDS


;Preinitialized data fields **********************************************

_DATA    SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_B     DB       'Data B'        ;declare a MODULE_B preinitialized field

_DATA    ENDS


;Uninitialized data fields ***********************************************

_BSS     SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_B      DB       5 DUP(?)        ;declare a MODULE_B uninitialized field

_BSS     ENDS


;Program text ************************************************************

DGROUP   GROUP    _DATA,CONST,_BSS

_TEXT    SEGMENT BYTE PUBLIC 'CODE'

         ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING
```

Figure 4-9. Structuring a .EXE program: MODULE_B.   *(more)*

```
        PUBLIC  PROC_B                  ;reference in MODULE_A
PROC_B  PROC    NEAR

        RET

PROC_B  ENDP

_TEXT   ENDS

        END
```

Figure 4-9. Continued.

```
;Source Module MODULE_C

;Constant declarations ****************************************************

CONST   SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_C   DB      'Constant C'    ;declare a MODULE_C constant

CONST   ENDS


;Preinitialized data fields ***********************************************

_DATA   SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_C    DB      'Data C'        ;declare a MODULE_C preinitialized field

_DATA   ENDS


;Uninitialized data fields ************************************************

_BSS    SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_C     DB      5 DUP(?)        ;declare a MODULE_C uninitialized field

_BSS    ENDS


;Program text *************************************************************

DGROUP  GROUP   _DATA,CONST,_BSS

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

        ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING
```

Figure 4-10. Structuring a .EXE program: MODULE_C.   (more)

```
        PUBLIC  PROC_C                      ;referenced in MODULE_A
PROC_C  PROC    NEAR

        RET

PROC_C  ENDP

_TEXT   ENDS

        END
```

Figure 4-10. Continued.

This example creates a small memory model program image, so the linked program can have only a single code segment and a single data segment — the simplest standard form of a .EXE program. *See* Using Microsoft's Contemporary Memory Models below.

In addition to declaring the four segments already discussed, MODULE_ A declares a STACK segment in which to define a block of memory for use as the program's stack and also defines the linking order of the five segments. Defining the linking order leaves the programmer free to declare the segments in any order when defining the segment contents — a necessity because the assembler has difficulty assembling programs that use forward references.

With Microsoft's MASM and LINK on the same disk with the .ASM files, the following commands can be made into a batch file:

```
MASM STRUCA;
MASM STRUCB;
MASM STRUCC;
LINK STRUCA+STRUCB+STRUCC/M;
```

These commands will assemble and link all the .ASM files listed, producing the memory map report file STRUCA.MAP shown in Figure 4-11.

```
Start   Stop    Length Name                Class
00000H  0000CH  0000DH _TEXT               CODE
0000EH  0001FH  00012H _DATA               DATA
00020H  0003DH  0001EH CONST               CONST
0003EH  0004EH  00011H _BSS                BSS
00050H  0014FH  00100H STACK               STACK

Origin    Group
0000:0    DGROUP


  Address           Publics by Name

0000:000B         PROC_B
0000:000C         PROC_C
```

Figure 4-11. Structuring a .EXE program: memory map report.   *(more)*

```
Address          Publics by Value

0000:000B        PROC_B
0000:000C        PROC_C
Program entry point at 0000:0000
```

Figure 4-11. Continued.

The above memory map report represents the memory diagram shown in Figure 4-12.
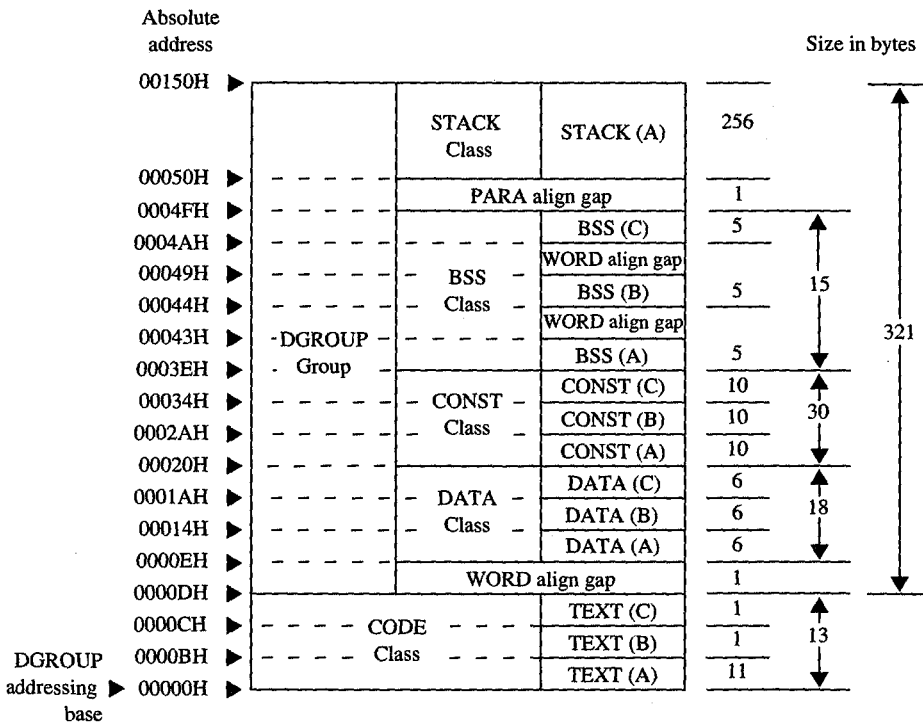
Figure 4-12. Structure of the sample .EXE program.

Using Microsoft's contemporary memory models

Now that we've analyzed the various aspects of designing assembly-language .EXE programs, we can look at how Microsoft's high-level-language compilers create .EXE programs from high-level-language source files. Even assembly-language programmers will find this discussion of interest and should seriously consider using the five standard memory models outlined here.

This discussion is based on the Microsoft C Compiler version 4.0, which, along with the Microsoft FORTRAN Compiler version 4.0, incorporates the most contemporary code generator currently available. These newer compilers generate code based on three to five

of the following standard programmer-selectable program structures, referred to as memory models. The discussion of each of these memory models will center on the model's use with the Microsoft C Compiler and will close with comments regarding any differences for the Microsoft FORTRAN Compiler.

Small (C compiler switch /AS) This model, the default, includes only a single code segment and a single data segment. All code must fit within 64 KB, and all data must fit within an additional 64 KB. Most C program designs fall into this category. Data can exceed the 64 KB limit only if the far and huge attributes are used, forcing the compiler to use far addressing, and the linker to place far and huge data items into separate segments. The data-size-threshold switch described for the compact model is ignored by the Microsoft C Compiler when used with a small model. The C compiler uses the default segment name _TEXT for all code and the default segment name _DATA for all non-far/huge data. Microsoft FORTRAN programs can generate a semblance of this model only by using the /NM (name module) and /AM (medium model) compiler switches in combination with the near attribute on all subprogram declarations.

Medium (C and FORTRAN compiler switch /AM) This model includes only a single data segment but breaks the code into multiple code segments. All data must fit within 64 KB, but the 64 KB restriction on code size applies only on a module-by-module basis. Data can exceed the 64 KB limit only if the far and huge attributes are used, forcing the compiler to use far addressing, and the linker to place far and huge data items into separate segments. The data-size-threshold switch described for the compact model is ignored by the Microsoft C Compiler when used with a medium model. The compiler uses the default segment name _DATA for all non-far/huge data and the template *module*_TEXT to create names for all code segments. The *module* element of *module*_TEXT indicates where the compiler is to substitute the name of the source module. For example, if the source module HELPFUNC.C is compiled using the medium model, the compiler creates the code segment HELPFUNC_TEXT. The Microsoft FORTRAN Compiler version 4.0 directly supports the medium model.

Compact (C compiler switch /AC) This model includes only a single code segment but breaks the data into multiple data segments. All code must fit within 64 KB, but the data is allowed to consume all the remaining available memory. The Microsoft C Compiler's optional data-size-threshold switch (/Gt) controls the placement of the larger data items into additional data segments, leaving the smaller items in the default segment for faster access. Individual data items within the program cannot exceed 64 KB under the compact model without being explicitly declared huge. The compiler uses the default segment name _TEXT for all code segments and the template *module*#_DATA to create names for all data segments. The *module* element indicates where the compiler is to substitute the source module's name; the # element represents a digit that the compiler changes for each additional data segment required to hold the module's data. The compiler starts with the digit 5 and counts up. For example, if the name of the source module is HELPFUNC.C, the compiler names the first data segment HELPFUNC5_DATA. FORTRAN programs can generate a semblance of this model only by using the /NM (name module) and /AL (large model) compiler switches in combination with the near attribute on all subprogram declarations.

Large (C and FORTRAN compiler switch /AL) This model creates multiple code and data segments. The compiler treats data in the same manner as it does for the compact model and treats code in the same manner as it does for the medium model. The Microsoft FORTRAN Compiler version 4.0 directly supports the large model.

Huge (C and FORTRAN compiler switch /AH) Allocation of segments under the huge model follows the same rules as for the large model. The difference is that individual data items can exceed 64 KB. Under the huge model, the compiler generates the necessary code to index arrays or adjust pointers across segment boundaries, effectively transforming the microprocessor's segment-addressed memory into linear-addressed memory. This makes the huge model especially useful for porting a program originally written for a processor that used linear addressing. The speed penalties the program pays in exchange for this addressing freedom require serious consideration. If the program actually contains any data structures exceeding 64 KB, it probably contains only a few. In that case, it is best to avoid using the huge model by explicitly declaring those few data items as huge using the huge keyword within the source module. This prevents penalizing all the non-huge items with extra addressing math. The Microsoft FORTRAN Compiler version 4.0 directly supports the huge model.

Figure 4-13 shows an example of the segment arrangement created by a large/huge model program. The example assumes two source modules: MSCA.C and MSCB.C. Each source module specifies enough data to cause the compiler to create two extra data segments for that module. The diagram does not show all the various segments that occur as a result of linking with the run-time library or as a result of compiling with the intention of using the CodeView debugger.

Groups	Classes	Segments	
DGROUP	STACK	STACK	◄ SMCLH: Program stack
	BSS	c_common	◄ SM: All uninitialized global items, CLH: Empty
		_BSS	◄ SMCLH: All uninitialized non-far/huge items
	CONST	CONST	◄ SMCLH: Constants (floating point constraints, segment addresses, etc.)
	DATA	_DATA	◄ SMCLH: All items that don't end up anywhere else
	FAR_BSS	FAR_BSS	◄ SM: Nonexistent, CLH: All uninitialized global items
	FAR_DATA	MSCB6_DATA	◄ From MSCB only: SM: Far/huge items, CLH: Items larger than threshold
		MSCB5_DATA	◄ From MSCB only: SM: Far/huge items, CLH: Items larger than threshold
		MSCA6_DATA	◄ From MSCA only: SM: Far/huge items, CLH: Items larger than threshold
		MSCA5_DATA	◄ From MSCA only: SM: Far/huge items, CLH: Items larger than threshold
	CODE	TEXT	◄ SC: All code, MLH: Run-time library code only
		MSCB_TEXT	◄ SC: Nonexistent, MLH: MSCB.C Code
		MSCA_TEXT	◄ SC: Nonexistent, MLH: MSCA.C Code

S = Small model   L = Large model
M = Medium model   H = Huge model
C = Compact model

Figure 4-13. General structure of a Microsoft C program.

Note that if the program declares an extremely large number of small data items, it can exceed the 64 KB size limit on the default data segment (_DATA) regardless of the memory model specified. This occurs because the data items all fall below the data-size-threshold limit (compiler /Gt switch), causing the compiler to place them in the _DATA segment. Lowering the data size threshold or explicitly using the far attribute within the source modules eliminates this problem.

Modifying the .EXE file header

With most of its language compilers, Microsoft supplies a utility program called EXEMOD. *See* PROGRAMMING UTILITIES: EXEMOD. This utility allows the programmer to display and modify certain fields contained within the .EXE file header. Following are the header fields EXEMOD can modify (based on EXEMOD version 4.0):

MAXALLOC This field can be modified by using EXEMOD's /MAX switch. Because EXEMOD operates on .EXE files that have already been linked, the /MAX switch can be used to modify the MAXALLOC field in existing .EXE programs that contain the default MAXALLOC value of FFFFH, provided the programs do not rely on MS-DOS's allocating all free memory to them. EXEMOD's /MAX switch functions in an identical manner to LINK's /CPARMAXALLOC switch.

MINALLOC This field can be modified by using EXEMOD's /MIN switch. Unlike the case with the MAXALLOC field, most programs do not have an arbitrary value for MINALLOC. MINALLOC normally represents uninitialized memory and stack space the linker has compressed out of the .EXE file, so a programmer should never *reduce* the MINALLOC value within a .EXE program written by someone else. If a program requires some minimum amount of extra dynamic memory in addition to any static fields, MINALLOC can be increased to ensure that the program will have this extra memory before receiving control. If this is done, the program will not have to verify that MS-DOS allocated enough memory to meet program needs. Of course, the same result can be achieved without EXEMOD by declaring this minimum extra memory as an uninitialized field at the end of the program.

Initial SP Value This field can be modified by using the /STACK switch to increase or decrease the size of a program's stack. However, modifying the initial SP value for programs developed using Microsoft language compiler versions earlier than the following may cause the programs to fail: C version 3.0, Pascal version 3.3, and FORTRAN version 3.3. Other language compilers may have the same restriction. The /STACK switch can also be used with programs developed using MASM, provided the stack space is linked at the end of the program, but it would probably be wise to change the size of the STACK segment declaration within the program instead. The linker also provides a /STACK switch that performs the same purpose.

Note: With the /H switch set, EXEMOD displays the current values of the fields within the .EXE header. This switch should not be used with the other switches. EXEMOD also displays field values if no switches are used.

Warning: EXEMOD also functions correctly when used with packed .EXE files created using EXEPACK or the /EXEPACK linker switch. However, it is important to use the EXEMOD version shipped with the linker or EXEPACK utility. Possible future changes in the packing method may result in incompatibilities between EXEMOD and nonassociated linker/EXEPACK versions.

Patching the .EXE program using DEBUG

Every experienced programmer knows that programs always seem to have at least one unspotted error. If a program has been distributed to other users, the programmer will probably need to provide those users with corrections when such bugs come to light. One inexpensive updating approach used by many large companies consists of mailing out single-page instructions explaining how the user can patch the program to correct the problem.

Program patching usually involves loading the program file into the DEBUG utility supplied with MS-DOS, storing new bytes into the program image, and then saving the program file back to disk. Unfortunately, DEBUG cannot load a .EXE program into memory and then save it back to disk in .EXE format. The programmer must trick DEBUG into patching .EXE program files, using the procedure outlined below. *See* PROGRAMMING UTILITIES: DEBUG.

Note: Users should be reminded to make backup copies of their program before attempting the patching procedure.

1. Rename the .EXE file using a filename extension that does not have special meaning for DEBUG. (Avoid .EXE, .COM, and .HEX.) For instance, MYPROG.BIN serves well as a temporary new name for MYPROG.EXE because DEBUG does not recognize a file with a .BIN extension as anything special. DEBUG will load the entire image of MYPROG.BIN, including the .EXE header and relocation table, into memory starting at offset 100H within a .COM-style program segment (as discussed previously).

2. Locate the area within the load module section of the .EXE file image that requires patching. The previous discussion of the .EXE file image, together with compiler/ assembler listings and linker memory map reports, provides the information necessary to locate the error within the .EXE file image. DEBUG loads the file image starting at offset 100H within a .COM-style program segment, so the programmer must compensate for this offset when calculating addresses within the file image. Also, the compiler listings and linker memory map reports provide addresses relative to the start of the program image within the .EXE file, not relative to the start of the file itself. Therefore, the programmer must first check the information contained in the .EXE file header to determine where the load module (the program's image) starts within the file.

3. Use DEBUG's E (Enter Data) or A (Assemble Machine Instructions) command to insert the corrections. (Normally, patch instructions to users would simply give an address at which the user should apply the patch. The user need not know how to determine the address.)

4. After the patch has been applied, simply issue the DEBUG W (Write File or Sectors) command to write the corrected image back to disk under the same filename, provided the patch has not increased the size of the program. If program size has

increased, first change the appropriate size fields in the .EXE header at the start of the file and use the DEBUG R (Display or Modify Registers) command to modify the BX and CX registers so that they contain the file image's new size. Then use the W command to write the image back to disk under the same name.

5. Use the DEBUG Q (Quit) command to return to MS-DOS command level, and then rename the file to the original .EXE filename extension.

.EXE summary

To summarize, the .EXE program and file structures provide considerable flexibility in the design of programs, providing the programmer with the necessary freedom to produce large-scale applications. Programs written using Microsoft's high-level-language compilers have access to five standardized program structure models (small, medium, compact, large, and huge). These standardized models are excellent examples of ways to structure assembly-language programs.

The .COM Program

The majority of differences between .COM and .EXE programs exist because .COM program files are not prefaced by header information. Therefore, .COM programs do not benefit from the features the .EXE header provides.

The absence of a header leaves MS-DOS with no way of knowing how much memory the .COM program requires in addition to the size of the program's image. Therefore, MS-DOS must always allocate the largest free block of memory to the .COM program, regardless of the program's true memory requirements. As was discussed for .EXE programs, this allocation of the largest block of free memory usually results in MS-DOS's allocating all remaining free memory — an action that can cause problems for multitasking supervisor programs.

The .EXE program header also includes the direct segment address relocation pointer table. Because they lack this table, .COM programs cannot make address references to the labels specified in SEGMENT directives, with the exception of SEGMENT AT address directives. If a .COM program did make these references, MS-DOS would have no way of adjusting the addresses to correspond to the actual segment address into which MS-DOS loaded the program. *See* Creating the .COM Program below.

The .COM program structure exists primarily to support the vast number of CP/M programs ported to MS-DOS. Currently, .COM programs are most often used to avoid adding the 512 bytes or more of .EXE header information onto small, simple programs that often do not exceed 512 bytes by themselves.

The .COM program structure has another advantage: Its memory organization places the PSP within the same address segment as the rest of the program. Thus, it is easier to access fields within the PSP in .COM programs.

Giving control to the .COM program

After allocating the largest block of free memory to the .COM program, MS-DOS builds a PSP in the lowest 100H bytes of the block. No difference exists between the PSP MS-DOS builds for .COM programs and the PSP it builds for .EXE programs. Also with .EXE programs, MS-DOS determines the initial values for the AL and AH registers at this time and then loads the entire .COM-file image into memory immediately following the PSP. Because .COM files have no file-size header fields, MS-DOS relies on the size recorded in the disk directory to determine the size of the program image. It loads the program exactly as it appears in the file, without checking the file's contents.

MS-DOS then sets the DS, ES, and SS segment registers to point to the start of the PSP. If able to allocate at least 64 KB to the program, MS-DOS sets the SP register to offset FFFFH + 1 (0000H) to establish an initial stack; if less than 64 KB are available for allocation to the program, MS-DOS sets the SP to 1 byte past the highest offset owned by the program. In either case, MS-DOS then pushes a single word of 0000H onto the program's stack for use in terminating the program.

Finally, MS-DOS transfers control to the program by setting the CS register to the PSP's segment address and the IP register to 0100H. This means that the program's entry point must exist at the very start of the program's image, as shown in later examples.

Figure 4-14 shows the overall structure of a .COM program as it receives control from MS-DOS.
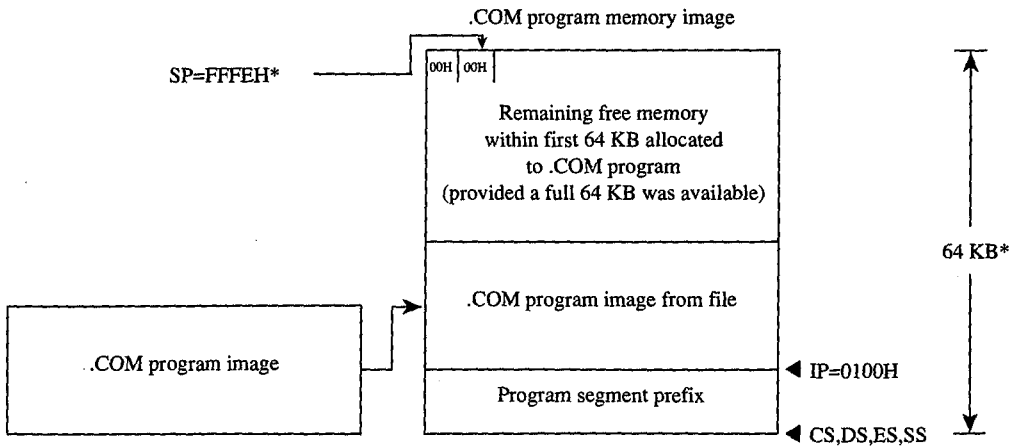
*The SP and 64 KB values are dependent upon MS-DOS having 64 KB or more of memory available to allocate to the .COM program at load time.

Figure 4-14. The .COM program: memory map diagram with register pointers.

Terminating the .COM program

A .COM program can use all the termination methods described for .EXE programs but should still use the MS-DOS Interrupt 21H Terminate Process with Return Code function (4CH) as the preferred method. If the .COM program must remain compatible with versions of MS-DOS earlier than 2.0, it can easily use any of the older termination methods, including those described as difficult to use from .EXE programs, because .COM programs execute with the CS register pointing to the PSP as required by these methods.

Creating the .COM program

A .COM program is created in the same manner as a .EXE program and then converted using the MS-DOS EXE2BIN utility. *See* PROGRAMMING UTILITIES: EXE2BIN.

Certain restrictions do apply to .COM programs, however. First, .COM programs cannot exceed 64 KB minus 100H bytes for the PSP minus 2 bytes for the zero word initially pushed on the stack.

Next, only a single segment — or at least a single addressing group — should exist within the program. The following two examples show ways to structure a .COM program to satisfy both this restriction and MASM's need to have data fields precede program code in the source file.

COMPROG1.ASM (Figure 4-15) declares only a single segment (*COMSEG*), so no special considerations apply when using the MASM OFFSET operator. *See* The MASM GROUP Directive above. COMPROG2.ASM (Figure 4-16) declares separate code (*CSEG*) and data (*DSEG*) segments, which the GROUP directive ties into a common addressing block. Thus, the programmer can declare data fields at the start of the source file and have the linker place the data fields segment (*DSEG*) after the code segment (*CSEG*) when it links the program, as discussed for the .EXE program structure. This second example simulates the program structuring provided under CP/M by Microsoft's old Macro-80 (M80) macro assembler and Link-80 (L80) linker. The design also expands easily to accommodate COMMON or other additional segments.

```
COMSEG   SEGMENT BYTE PUBLIC 'CODE'
         ASSUME  CS:COMSEG,DS:COMSEG,ES:COMSEG,SS:COMSEG
         ORG     0100H

BEGIN:
         JMP     START              ;skip over data fields
;Place your data fields here.

START:                              '
;Place your program text here.
         MOV     AX,4C00H           ;terminate (MS-DOS 2.0 or later only)
         INT     21H
COMSEG   ENDS
         END     BEGIN
```

Figure 4-15. .COM program with data at start.

```
CSEG    SEGMENT BYTE PUBLIC 'CODE'         ;establish segment order
CSEG    ENDS
DSEG    SEGMENT BYTE PUBLIC 'DATA'
DSEG    ENDS
COMGRP  GROUP   CSEG,DSEG                  ;establish joint address base
DSEG    SEGMENT
;Place your data fields here.
DSEG    ENDS
CSEG    SEGMENT

        ASSUME  CS:COMGRP,DS:COMGRP,ES:COMGRP,SS:COMGRP
        ORG     0100H

BEGIN:
;Place your program text here.  Remember to use
;OFFSET COMGRP:LABEL whenever you use OFFSET.
        MOV     AX,4C00H                   ;terminate (MS-DOS 2.0 or later only)
        INT     21H
CSEG    ENDS
        END     BEGIN
```

Figure 4-16. .COM program with data at end.

These examples demonstrate other significant requirements for producing a functioning
.COM program. For instance, the *ORG 0100H* statement in both examples tells MASM to
start assembling the code at offset 100H within the encompassing segment. This corre-
sponds to MS-DOS's transferring control to the program at IP = 0100H. In addition, the
entry-point label (BEGIN) immediately follows the ORG statement and appears again as a
parameter to the END statement. Together, these factors satisfy the requirement that .COM
programs declare their entry point at offset 100H. If any factor is missing, the MS-DOS
EXE2BIN utility will not properly convert the .EXE file produced by the linker into a .COM
file. Specifically, if a .COM program declares an entry point (as a parameter to the END
statement) that is at neither offset 0100H nor offset 0000H, EXE2BIN rejects the .EXE file
when the programmer attempts to convert it. If the program fails to declare an entry point
or declares an entry point at offset 0000H, EXE2BIN assumes that the .EXE file is to be
converted to a binary image rather than to a .COM image. When EXE2BIN converts a .EXE
file to a non-.COM binary file, it does not strip the extra 100H bytes the linker places in
front of the code as a result of the *ORG 0100H* instruction. Thus, the program actually
begins at offset 200H when MS-DOS loads it into memory, but all the program's address
references will have been assembled and linked based on the 100H offset. As a result, the
program — and probably the rest of the system as well — is likely to crash.

A .COM program also must not contain direct segment address references to any segments
that make up the program. Thus, the .COM program cannot reference any segment labels
or reference any labels as long (FAR) pointers. (This rule does not prevent the program
from referencing segment labels declared using the SEGMENT AT address directive.)
Following are various examples of direct segment address references that are *not* per-
mitted as part of .COM programs:

```
PROC_A  PROC    FAR
PROC_A  ENDP
        CALL    PROC_A          ;intersegment call
        JMP     PROC_A          ;intersegment jump
```

or

```
        EXTRN   PROC_A:FAR
        CALL    PROC_A          ;intersegment call
        JMP     PROC_A          ;intersegment jump
```

or

```
        MOV     AX,SEG SEG_A    ;segment address
        DD      LABEL_A         ;segment:offset pointer
```

Finally, .COM programs must not declare any segments with the STACK *combine* type. If a program declares a segment with the STACK *combine* type, the linker will insert initial SS and SP values into the .EXE file header, causing EXE2BIN to reject the .EXE file. A .COM program does not have explicitly declared stacks, although it can reserve space in a non-STACK *combine* type segment to which it can initialize the SP register *after* it receives control. The absence of a stack segment will cause the linker to issue a harmless warning message.

When the program is assembled and linked into a .EXE file, it must be converted into a binary file with a .COM extension by using the EXE2BIN utility as shown in the following example for the file YOURPROG.EXE:

```
C>EXE2BIN YOURPROG YOURPROG.COM  <Enter>
```

It is not necessary to delete or rename a .EXE file with the same filename as the .COM file before trying to execute the .COM file as long as both remain in the same directory, because MS-DOS's order of execution is .COM files first, then .EXE files, and finally .BAT files. However, the safest practice is to delete a .EXE file immediately after converting it to a .COM file in case the .COM file is later renamed or moved to a different directory. If a .EXE file designed for conversion to a .COM file is executed by accident, it is likely to crash the system.

Patching the .COM program using DEBUG

As discussed for .EXE files, a programmer who distributes software to users will probably want to send instructions on how to patch in error corrections. This approach to software updates lends itself even better to .COM files than it does to .EXE files.

For example, because .COM files contain only the code image, they need not be renamed in order to read and write them using DEBUG. The user need only be instructed on how to load the .COM file into DEBUG, how to patch the program, and how to write the patched image back to disk. Calculating the addresses and patch values is even easier, because no header exists in the .COM file image to cause complications. With the preceding exceptions, the details for patching .COM programs remain the same as previously outlined for .EXE programs.

.COM summary

To summarize, the .COM program and file structures are a simpler but more restricted approach to writing programs than the .EXE structure because the programmer has only a single memory model from which to choose (the .COM program segment model). Also, .COM program files do not contain the 512-byte (or more) header inherent to .EXE files, so the .COM program structure is well suited to small programs for which adding 512 bytes of header would probably at least double the file's size.

Summary of Differences

The following table summarizes the differences between .COM and .EXE programs.

	.COM program	.EXE program
Maximum size	65536 bytes minus 256 bytes for PSP and 2 bytes for stack	No limit
Entry point	PSP:0100H	Defined by END statement
CS at entry	PSP	Segment containing program's entry point
IP at entry	0100H	Offset of entry point within its segment
DS at entry	PSP	PSP
ES at entry	PSP	PSP
SS at entry	PSP	Segment with STACK attribute
SP at entry	FFFEH or top word in available memory, whichever is lower	End of segment defined with STACK attribute
Stack at entry	Zero word	Initialized or uninitialized, depending on source
Stack size	65536 bytes minus 256 bytes for PSP and size of executable code and data	Defined in segment with STACK attribute
Subroutine calls	NEAR	NEAR or FAR
Exit method	Interrupt 21H Function 4CH preferred; NEAR RET if MS-DOS versions 1.x	Interrupt 21H Function 4CH preferred; indirect jump to PSP:0000H if MS-DOS versions 1.x
Size of file	Exact size of program	Size of program plus header (at least 512 extra bytes)

Which format the programmer uses for an application usually depends on the program's intended size, but the decision can also be influenced by a program's need to address multiple memory segments. Normally, small utility programs (such as CHKDSK and FORMAT) are designed as .COM programs; large programs (such as the Microsoft C Compiler) are designed as .EXE programs. The ultimate decision is, of course, the programmer's.

Keith Burgoyne

Article 5:
Character Device Input and Output

All functional computer systems are composed of a central processing unit (CPU), some memory, and peripheral devices that the CPU can use to store data or communicate with the outside world. In MS-DOS systems, the essential peripheral devices are the keyboard (for input), the display (for output), and one or more disk drives (for nonvolatile storage). Additional devices such as printers, modems, and pointing devices extend the functionality of the computer or offer alternative methods of using the system.

MS-DOS recognizes two types of devices: block devices, which are usually floppy-disk or fixed-disk drives; and character devices, such as the keyboard, display, printer, and communications ports.

The distinction between block and character devices is not always readily apparent, but in general, block devices transfer information in chunks, or blocks, and character devices move data one character (usually 1 byte) at a time. MS-DOS identifies each block device by a drive letter assigned when the device's controlling software, the device driver, is loaded. A character device, on the other hand, is identified by a logical name (similar to a filename and subject to many of the same restrictions) built into its device driver. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

Background Information

Versions 1.x of MS-DOS, first released for the IBM PC in 1981, supported peripheral devices with a fixed set of device drivers loaded during system initialization from the hidden file IO.SYS (or IBMBIO.COM with PC-DOS). These versions of MS-DOS offered application programs a high degree of input/output device independence by allowing character devices to be treated like files, but they did not provide an easy way to augment the built-in set of drivers if the user wished to add a third-party peripheral device to the system.

With the release of MS-DOS version 2.0, the hardware flexibility of the system was tremendously enhanced. Versions 2.0 and later support installable device drivers that can reside in separate files on the disk and can be linked into the operating system simply by adding a DEVICE directive to the CONFIG.SYS file on the startup disk. *See* USER COMMANDS: CONFIG.SYS: DEVICE. A well-defined interface between installable drivers and the MS-DOS kernel allows such drivers to be written for most types of peripheral devices without the need for modification to the operating system itself.

The CONFIG.SYS file can contain a number of different DEVICE commands to load separate drivers for pointing devices, magnetic-tape drives, network interfaces, and so on. Each driver, in turn, is specialized for the hardware characteristics of the device it supports.

When the system is turned on or restarted, the installable device drivers are added to the chain, or linked list, of default device drivers loaded from IO.SYS during MS-DOS initialization. Thus, the need for the system's default set of device drivers to support a wide range of optional device types and features at an excessive cost of system memory is avoided.

One important distinction between block and character devices is that MS-DOS always adds new block-device drivers to the tail of the driver chain but adds new character-device drivers to the head of the chain. Thus, because MS-DOS searches the chain sequentially and uses the first driver it finds that satisfies its search conditions, any existing character-device driver can be superseded by simply installing another driver with an identical logical device name.

This article covers some of the details of working with MS-DOS character devices: displaying text, keyboard input, and other basic character I/O functions; the definition and use of standard input and output; redirection of the default character devices; and the use of the IOCTL function (Interrupt 21H Function 44H) to communicate directly with a character-device driver. Much of the information presented in this article is applicable only to MS-DOS versions 2.0 and later.

Accessing Character Devices

Application programs can use either of two basic techniques to access character devices in a portable manner under MS-DOS. First, a program can use the handle-type function calls that were added to MS-DOS in version 2.0. Alternatively, a program can use the so-called "traditional" character-device functions that were present in versions 1.x and have been retained in the operating system for compatibility. Because the handle functions are more powerful and flexible, they are discussed first.

A handle is a 16-bit number returned by the operating system whenever a file or device is opened or created by passing a name to MS-DOS Interrupt 21H Function 3CH (Create File with Handle), 3DH (Open File with Handle), 5AH (Create Temporary File), or 5BH (Create New File). After a handle is obtained, it can be used with Interrupt 21H Function 3FH (Read File or Device) or Function 40H (Write File or Device) to transfer data between the computer's memory and the file or device.

During an open or create function call, MS-DOS searches the device-driver chain sequentially for a character device with the specified name (the extension is ignored) before searching the disk directory. Thus, a file with the same name as any character device in the driver chain — for example, the file NUL.TXT — cannot be created, nor can an existing file be accessed if a device in the chain has the same name.

The second method for accessing character devices is through the traditional MS-DOS character input and output functions, Interrupt 21H Functions 01H through 0CH. These functions are designed to communicate directly with the keyboard, display, printer, and serial port. Each of these devices has its own function or group of functions, so neither

names nor handles need be used. However, in MS-DOS versions 2.0 and later, these function calls are translated within MS-DOS to make use of the same routines that are used by the handle functions, so the traditional keyboard and display functions are affected by I/O redirection and piping.

Use of either the traditional or the handle-based method for character device I/O results in highly portable programs that can be used on any computer that runs MS-DOS. A third, less portable access method is to use the hardware-specific routines resident in the read-only memory (ROM) of a specific computer (such as the IBM PC ROM BIOS driver functions), and a fourth, definitely nonportable approach is to manipulate the peripheral device's adapter directly, bypassing the system software altogether. Although these latter hardware-dependent methods cannot be recommended, they are admittedly sometimes necessary for performance reasons.

The Basic MS-DOS Character Devices

Every MS-DOS system supports at least the following set of logical character devices without the need for any additional installable drivers:

Device	Meaning
CON	Keyboard and display
PRN	System list device, usually a parallel port
AUX	Auxiliary device, usually a serial port
CLOCK$	System real-time clock
NUL	"Bit-bucket" device

These devices can be opened by name or they can be addressed through the "traditional" function calls; strings can be read from or written to the devices according to their capabilities on any MS-DOS system. Data written to the NUL device is discarded; reads from the NUL device always return an end-of-file condition.

PC-DOS and compatible implementations of MS-DOS typically also support the following logical character-device names:

Device	Meaning
COM1	First serial communications port
COM2	Second serial communications port
LPT1	First parallel printer port
LPT2	Second parallel printer port
LPT3	Third parallel printer port

In such systems, PRN is an alias for LPT1 and AUX is an alias for COM1. The MODE command can be used to redirect an LPT device to another device. *See* USER COMMANDS: MODE.

As previously mentioned, any of these default character-device drivers can be superseded by a user-installed device driver — for example, one that offers enhanced functionality or changes the device's apparent characteristics. One frequently used alternative character-device driver is ANSI.SYS, which replaces the standard MS-DOS CON device driver and allows ANSI escape sequences to be used to perform tasks such as clearing the screen, controlling the cursor position, and selecting character attributes. *See* USER COMMANDS: ANSI.SYS.

The standard devices

Under MS-DOS versions 2.0 and later, each program owns five previously opened handles for character devices (referred to as the standard devices) when it begins executing. These handles can be used for input and output operations without further preliminaries. The five standard devices and their associated handles are

Standard Device Name	Handle	Default Assignment
Standard input (*stdin*)	0	CON
Standard output (*stdout*)	1	CON
Standard error (*stderr*)	2	CON
Standard auxiliary (*stdaux*)	3	AUX
Standard printer (*stdprn*)	4	PRN

The standard input and standard output handles are especially important because they are subject to I/O redirection. Although these handles are associated by default with the CON device so that read and write operations are implemented using the keyboard and video display, the user can associate the handles with other character devices or with files by using redirection parameters in a program's command line:

Redirection	Result
< *file*	Causes read operations from standard input to obtain data from *file*.
> *file*	Causes data written to standard output to be placed in *file*.
>> *file*	Causes data written to standard output to be appended to *file*.
p1 | *p2*	Causes data written to standard output by program *p1* to appear as the standard input of program *p2*.

This ability to redirect I/O adds great flexibility and power to the system. For example, programs ordinarily controlled by keyboard entries can be run with "scripts" from files, the output of a program can be captured in a file or on a printer for later inspection, and general-purpose programs (filters) can be written that process text streams without regard to the text's origin or destination. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Writing MS-DOS Filters.

Ordinarily, an application program is not aware that its input or output has been redirected, although a write operation to standard output will fail unexpectedly if standard output was redirected to a disk file and the disk is full. An application can check for the existence of I/O redirection with an IOCTL (Interrupt 21H Function 44H) call, but it cannot obtain any information about the destination of the redirected handle except whether it is associated with a character device or with a file.

Raw versus cooked mode

MS-DOS associates each handle for a character device with a mode that determines how I/O requests directed to that handle are treated. When a handle is in raw mode, characters are passed between the application program and the device driver without any filtering or buffering by MS-DOS. When a handle is in cooked mode, MS-DOS buffers any data that is read from or written to the device and takes special actions when certain characters are detected.

During cooked mode input, MS-DOS obtains characters from the device driver one at a time, checking each character for a Control-C. The characters are assembled into a string within an internal MS-DOS buffer. The input operation is terminated when a carriage return (0DH) or an end-of-file mark (1AH) is received or when the number of characters requested by the application have been accumulated. If the source is standard input, lone linefeed characters are translated to carriage-return/linefeed pairs. The string is then copied from the internal MS-DOS buffer to the application program's buffer, and control returns to the application program.

During cooked mode output, MS-DOS transfers the characters in the application program's output buffer to the device driver one at a time, checking after each character for a Control-C pending at the keyboard. If the destination is standard output and standard output has not been redirected, tabs are expanded to spaces using eight-column tab stops. Output is terminated when the requested number of characters have been written or when an end-of-file mark (1AH) is encountered in the output string.

In contrast, during raw mode input or output, data is transferred directly between the application program's buffer and the device driver. Special characters such as carriage return and the end-of-file mark are ignored, and the exact number of characters in the application program's request are always read or written. MS-DOS does not break the strings into single-character calls to the device driver and does not check the keyboard buffer for Control-C entries during the I/O operation. Finally, characters read from standard input in raw mode are not echoed to standard output.

As might be expected from the preceding description, raw mode input or output is usually much faster than cooked mode input or output, because each character is not being individually processed by the MS-DOS kernel. Raw mode also allows programs to read characters from the keyboard buffer that would otherwise be trapped by MS-DOS (for example, Control-C, Control-P, and Control-S). (If BREAK is on, MS-DOS will still check for Control-C entries during other function calls, such as disk operations, and transfer control

to the Control-C exception handler if a Control-C is detected.) A program can use the MS-DOS IOCTL Get and Set Device Data services (Interrupt 21H Function 44H Subfunctions 00H and 01H) to set the mode for a character-device handle. *See* IOCTL below.

Ordinarily, raw or cooked mode is strictly an attribute of a specific handle that was obtained from a previous open operation and affects only the I/O operations requested by the program that owns the handle. However, when a program uses IOCTL to select raw or cooked mode for one of the standard device handles, the selection has a global effect on the behavior of the system because those handles are never closed. Thus, some of the "traditional" keyboard input functions might behave in unexpected ways. Consequently, programs that change the mode on a standard device handle should save the handle's mode at entry and restore it before performing a final exit to MS-DOS, so that the operation of COMMAND.COM and other applications will not be disturbed. Such programs should also incorporate custom critical error and Control-C exception handlers so that the programs cannot be terminated unexpectedly. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

The keyboard

Among the MS-DOS Interrupt 21H functions are two methods of checking for and receiving input from the keyboard: the traditional method, which uses MS-DOS character input Functions 01H, 06H, 07H, 08H, 0AH, 0BH, and 0CH (Table 5-1); and the handle method, which uses Function 3FH. Each of these methods has its own advantages and disadvantages. *See* SYSTEM CALLS.

Table 5-1. Traditional MS-DOS Character Input Functions.

Function	Name	Read Multiple Characters	Echo	Ctrl-C Check
01H	Character Input with Echo	No	Yes	Yes
06H	Direct Console I/O	No	No	No
07H	Unfiltered Character Input Without Echo	No	No	No
08H	Character Input Without Echo	No	No	Yes
0AH	Buffered Keyboard Input	Yes	Yes	Yes
0BH	Check Keyboard Status	No	No	Yes
0CH	Flush Buffer, Read Keyboard	*	*	*

* Varies depending on function (from above) called in the AL register.

The first four traditional keyboard input calls are really very similar. They all return a character in the AL register; they differ mainly in whether they echo that character to the display and whether they are sensitive to interruption by the user's entry of a Control-C. Both Functions 06H and 0BH can be used to test keyboard status (that is, whether a key has been pressed and is waiting to be read by the program); Function 0BH is simpler to use, but Function 06H is immune to Control-C entries.

Function 0AH is used to read a "buffered line" from the user, meaning that an entire line is accepted by MS-DOS before control returns to the program. The line is terminated when the user presses the Enter key or when the maximum number of characters (to 255) specified by the program have been received. While entry of the line is in progress, the usual editing keys (such as the left and right arrow keys and the function keys on IBM PCs and compatibles) are active; only the final, edited line is delivered to the requesting program.

Function 0CH allows a program to flush the type-ahead buffer before accepting input. This capability is important for occasions when a prompt must be displayed unexpectedly (such as when a critical error occurs) and the user could not have typed ahead a valid response. This function should also be used when the user is being prompted for a critical decision (such as whether to erase a file), to prevent a character that was previously pressed by accident from triggering an irrecoverable operation. Function 0CH is unusual in that it is called with the number of one of the other keyboard input functions in register AL. After any pending input has been discarded, Function 0CH simply transfers to the other specified input function; thus, its other parameters (if any) depend on the function that ultimately will be executed.

The primary disadvantage of the traditional function calls is that they handle redirected input poorly. If standard input has been redirected to a file, no way exists for a program calling the traditional input functions to detect that the end of the file has been reached — the input function will simply wait forever, and the system will appear to hang.

A program that wishes to use handle-based I/O to get input from the keyboard must use the MS-DOS Read File or Device service, Interrupt 21H Function 3FH. Ordinarily, the program can employ the predefined handle for standard input (0), which does not need to be opened and which allows the program's input to be redirected by the user to another file or device. If the program needs to circumvent redirection and ensure that its input is from the keyboard, it can open the CON device with Interrupt 21H Function 3DH and use the handle obtained from that open operation instead of the standard input handle.

A program using the handle functions to read the keyboard can control the echoing of characters and sensitivity to Control-C entries by selecting raw or cooked mode with the IOCTL Get and Set Device Data services (default = cooked mode). To test the keyboard status, the program can either issue an IOCTL Check Input Status call (Interrupt 21H Function 44H Subfunction 06H) or use the traditional Check Keyboard Status call (Interrupt 21H Function 0BH).

The primary advantages of the handle method for keyboard input are its symmetry with file operations and its graceful handling of redirected input. The handle function also allows strings as long as 65535 bytes to be requested; the traditional Buffered Keyboard Input function allows a maximum of 255 characters to be read at a time. This consideration is important for programs that are frequently used with redirected input and output (such as filters), because reading and writing larger blocks of data from files results in more efficient operation. The only real disadvantage to the handle method is that it is limited to MS-DOS versions 2.0 and later (although this is no longer a significant restriction).

Role of the ROM BIOS

When a key is pressed on the keyboard of an IBM PC or compatible, it generates a hardware interrupt (09H) that is serviced by a routine in the ROM BIOS. The ROM BIOS interrupt handler reads I/O ports assigned to the keyboard controller and translates the key's scan code into an ASCII character code. The result of this translation depends on the current state of the NumLock and CapsLock toggles, as well as on whether the Shift, Control, or Alt key is being held down. (The ROM BIOS maintains a keyboard flags byte at address 0000:0417H that gives the current status of each of these modifier keys.)

After translation, both the scan code and the ASCII code are placed in the ROM BIOS's 32-byte (16-character) keyboard input buffer. In the case of "extended" keys such as the function keys or arrow keys, the ASCII code is a zero byte and the scan code carries all the information. The keyboard buffer is arranged as a circular, or ring, buffer and is managed as a first-in/first-out queue. Because of the method used to determine when the buffer is empty, one position in the buffer is always wasted; the maximum number of characters that can be held in the buffer is therefore 15. Keys pressed when the buffer is full are discarded and a warning beep is sounded.

The ROM BIOS provides an additional module, invoked by software Interrupt 16H, that allows programs to test keyboard status, determine whether characters are waiting in the type-ahead buffer, and remove characters from the buffer. *See* Appendix O: IBM PC BIOS Calls. Its use by application programs should ordinarily be avoided, however, to prevent introducing unnecessary hardware dependence.

On IBM PCs and compatibles, the keyboard input portion of the CON driver in the BIOS is a simple sequence of code that calls ROM BIOS Interrupt 16H to do the hardware-dependent work. Thus, calls to MS-DOS for keyboard input by an application program are subject to two layers of translation: The Interrupt 21H function call is converted by the MS-DOS kernel to calls to the CON driver, which in turn remaps the request onto a ROM BIOS call that obtains the character.

Keyboard programming examples

Example: Use the ROM BIOS keyboard driver to read a character from the keyboard. The character is not echoed to the display.

```
        mov     ah,00h          ; subfunction 00H = read character
        int     16h             ; transfer to ROM BIOS
                                ; now AH = scan code, AL = character
```

Example: Use the MS-DOS traditional keyboard input function to read a character from the keyboard. The character is not echoed to the display. The input can be interrupted with a Ctrl-C keystroke.

```
        mov     ah,08h          ; function 08H = character input
                                ;    without echo
        int     21h             ; transfer to MS-DOS
                                ; now AL = character
```

Example: Use the MS-DOS traditional Buffered Keyboard Input function to read an entire line from the keyboard, specifying a maximum line length of 80 characters. All editing keys are active during entry, and the input is echoed to the display.

```
kbuf    db      80              ; maximum length of read
        db      0               ; actual length of read
        db      80 dup (0)      ; keyboard input goes here
        .
        .
        .
        mov     dx,seg kbuf     ; set DS:DX = address of
        mov     ds,dx           ; keyboard input buffer
        mov     dx,offset kbuf
        mov     ah,0ah          ; function 0AH = read buffered line
        int     21h             ; transfer to MS-DOS
                                ; terminated by a carriage return,
                                ; and kbuf+1 = length of input,
                                ; not including the carriage return
```

Example: Use the MS-DOS handle-based Read File or Device function and the standard input handle to read an entire line from the keyboard, specifying a maximum line length of 80 characters. All editing keys are active during entry, and the input is echoed to the display. (The input will not terminate on a carriage return as expected if standard input is in raw mode.)

```
kbuf    db      80 dup (0)      ; buffer for keyboard input
        .
        .
        .
        mov     dx,seg kbuf     ; set DS:DX = address of
        mov     ds,dx           ; keyboard input buffer
        mov     dx,offset kbuf
        mov     cx,80           ; CX = maximum length of input
        mov     bx,0            ; standard input handle = 0
        mov     ah,3fh          ; function 3FH = read file/device
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if function failed
                                ; otherwise AX = actual
                                ; length of keyboard input,
                                ; including carriage-return and
                                ; linefeed, and the data is
                                ; in the buffer 'kbuf'
```

The display

The output half of the MS-DOS logical character device CON is the video display. On IBM PCs and compatibles, the video display is an "option" of sorts that comes in several forms. IBM has introduced five video subsystems that support different types of displays: the Monochrome Display Adapter (MDA), the Color/Graphics Adapter (CGA), the Enhanced Graphics Adapter (EGA), the Video Graphics Array (VGA), and the Multi-Color Graphics Array (MCGA). Other, non-IBM-compatible video subsystems in common use include the Hercules Graphics Card and its variants that support downloadable fonts.

Two portable techniques exist for writing text to the video display with MS-DOS function calls. The traditional method is supported by Interrupt 21H Functions 02H (Character Output), 06H (Direct Console I/O), and 09H (Display String). The handle method is supported by Function 40H (Write File or Device) and is available only in MS-DOS versions 2.0 and later. *See* SYSTEM CALLS: INTERRUPT 21H: Functions 02H, 06H, 09H, 40H. All these calls treat the display essentially as a "glass teletype" and do not support bit-mapped graphics.

Traditional Functions 02H and 06H are similar. Both are called with the character to be displayed in the DL register; they differ in that Function 02H is sensitive to interruption by the user's entry of a Control-C, whereas Function 06H is immune to Control-C but cannot be used to output the character 0FFH (ASCII rubout). Both calls check specifically for carriage return (0DH), linefeed (0AH), and backspace (08H) characters and take the appropriate action if these characters are detected.

Because making individual calls to MS-DOS for each character to be displayed is inefficient and slow, the traditional Display String function (09H) is generally used in preference to Functions 02H and 06H. Function 09H is called with the address of a string that is terminated with a dollar-sign character ($); it displays the entire string in one operation, regardless of its length. The string can contain embedded control characters such as carriage return and linefeed.

To use the handle method for screen display, programs must call the MS-DOS Write File or Device service, Interrupt 21H Function 40H. Ordinarily, a program should use the predefined handle for standard output (1) to send text to the screen, so that any redirection requested by the user on the program's command line will be honored. If the program needs to circumvent redirection and ensure that its output goes to the screen, it can either use the predefined handle for standard error (2) or explicitly open the CON device with Interrupt 21H Function 3DH and use the resulting handle for its write operations.

The handle technique for displaying text has several advantages over the traditional calls. First, the length of the string to be displayed is passed as an explicit parameter, so the string need not contain a special terminating character and the $ character can be displayed as part of the string. Second, the traditional calls are translated to handle calls inside MS-DOS, so the handle calls have less internal overhead and are generally faster. Finally, use of the handle Write File or Device function to display text is symmetric with the methods the program must use to access its files. In short, the traditional functions should be avoided unless the program must be capable of running under MS-DOS versions 1.x.

Controlling the screen

One of the deficiencies of the standard MS-DOS CON device driver is the lack of screen-control capabilities. The default CON driver has no built-in routines to support cursor placement, screen clearing, display mode selection, and so on.

In MS-DOS versions 2.0 and later, an optional replacement CON driver is supplied in the file ANSI.SYS. This driver contains most of the screen-control capabilities needed by text-oriented application programs. The driver is installed by adding a DEVICE directive to the

CONFIG.SYS file and restarting the system. When ANSI.SYS is active, a program can position the cursor, inquire about the current cursor position, select foreground and background colors, and clear the current line or the entire screen by sending an escape sequence consisting of the ASCII Esc character (1BH) followed by various function-specific parameters to the standard output device. *See* USER COMMANDS: ANSI.SYS.

Programs that use the ANSI.SYS capabilities for screen control are portable to any MS-DOS implementation that contains the ANSI.SYS driver. Programs that seek improved performance by calling the ROM BIOS video driver or by assuming direct control of the hardware are necessarily less portable and usually require modification when new PC models or video subsystems are released.

Role of the ROM BIOS

The video subsystems in IBM PCs and compatibles use a hybrid of memory-mapped and port-addressed I/O. A range of the machine's memory addresses is typically reserved for a video refresh buffer that holds the character codes and attributes to be displayed on the screen; the cursor position, display mode, palettes, and similar global display characteristics are governed by writing control values to specific I/O ports.

The ROM BIOS of IBM PCs and compatibles contains a primitive driver for the MDA, CGA, EGA, VGA, and MCGA video subsystems. This driver supports the following functions:

* Read or write characters with attributes at any screen position.
* Query or set the cursor position.
* Clear or scroll an arbitrary portion of the screen.
* Select palette, background, foreground, and border colors.
* Query or set the display mode (40-column text, 80-column text, all-points-addressable graphics, and so on).
* Read or write a pixel at any screen coordinate.

These functions are invoked by a program through software Interrupt 10H. *See* Appendix O: IBM PC BIOS Calls. In PC-DOS-compatible implementations of MS-DOS, the display portions of the MS-DOS CON and ANSI.SYS drivers use these ROM BIOS routines. Video subsystems that are not IBM compatible either must contain their own ROM BIOS or must be used with an installable device driver that captures Interrupt 10H and provides appropriate support functions.

Text-only application programs should avoid use of the ROM BIOS functions or direct access to the hardware whenever possible, to ensure maximum portability between MS-DOS systems. However, because the MS-DOS CON driver contains no support for bit-mapped graphics, graphically oriented applications usually must resort to direct control of the video adapter and its refresh buffer for speed and precision.

Display programming examples

Example: Use the ROM BIOS Interrupt 10H function to write an asterisk character to the display in text mode. (In graphics mode, BL must also be set to the desired foreground color.)

```
        mov     ah,0eh          ; subfunction 0EH = write character
                                ; in teletype mode
        mov     al,'*'          ; AL = character to display
        mov     bh,0            ; select display page 0
        int     10h             ; transfer to ROM BIOS video driver
```

Example: Use the MS-DOS traditional function to write an asterisk character to the display. If the user's entry of a Control-C is detected during the output and standard output is in cooked mode, MS-DOS calls the Control-C exception handler whose address is found in the vector for Interrupt 23H.

```
        mov     ah,02h          ; function 02H = display character
        mov     dl,'*'          ; DL = character to display
        int     21h             ; transfer to MS-DOS
```

Example: Use the MS-DOS traditional function to write a string to the display. The output is terminated by the $ character and can be interrupted when the user enters a Control-C if standard output is in cooked mode.

```
msg     db      'This is a test message','$'
        .
        .
        .
        mov     dx,seg msg      ; DS:DX = address of text
        mov     ds,dx           ; to display
        mov     dx,offset msg
        mov     ah,09h          ; function 09H = display string
        int     21h             ; transfer to MS-DOS
```

Example: Use the MS-DOS handle-based Write File or Device function and the predefined handle for standard output to write a string to the display. Output can be interrupted by the user's entry of a Control-C if standard output is in cooked mode.

```
msg      db      'This is a test message'
msg_len  equ     $-msg
         .
         .
         .
        mov     dx,seg msg      ; DS:DX = address of text
        mov     ds,dx           ; to display
        mov     dx,offset msg
        mov     cx,msg_len      ; CX = length of text
        mov     bx,1            ; BX = handle for standard output
        mov     ah,40h          ; function 40H = write file/device
        int     21h             ; transfer to MS-DOS
```

The serial communications ports

Through version 3.2, MS-DOS has built-in support for two serial communications ports, identified as COM1 and COM2, by means of three drivers named AUX, COM1, and COM2. (AUX is ordinarily an alias for COM1.)

The traditional MS-DOS method of reading from and writing to the serial ports is through Interrupt 21H Function 03H for AUX input and Function 04H for AUX output. In MS-DOS versions 2.0 and later, the handle-based Read File or Device and Write File or Device functions (Interrupt 21H Functions 3FH and 40H) can be used to read from or write to the auxiliary device. A program can use the predefined handle for the standard auxiliary device (3) with Functions 3FH and 40H, or it can explicitly open the COM1 or COM2 devices with Interrupt 21H Function 3DH and use the handle obtained from that open operation to perform read and write operations.

MS-DOS support for the serial communications port is inadequate in several respects for high-performance serial I/O applications. First, MS-DOS provides no portable way to test for the existence or the status of a particular serial port in a system; if a program "opens" COM2 and writes data to it and the physical COM2 adapter is not present in the system, the program may simply hang. Similarly, if the serial port exists but no character has been received and the program attempts to read a character, the program will hang until one is available; there is no traditional function call to check if a character is waiting as there is for the keyboard.

MS-DOS also provides no portable method to initialize the communications adapter to a particular baud rate, word length, and parity. An application must resort to ROM BIOS calls, manipulate the hardware directly, or rely on the user to configure the port properly with the MODE command before running the application that uses it. The default settings for the serial port on PC-DOS-compatible systems are 2400 baud, no parity, 1 stop bit, and 8 databits. *See* USER COMMANDS: MODE.

A more serious problem with the default MS-DOS auxiliary device driver in IBM PCs and compatibles, however, is that it is not interrupt driven. Accordingly, when baud rates above 1200 are selected, characters can be lost during time-consuming operations performed by the drivers for other devices, such as clearing the screen or reading or writing a floppy-disk sector. Because the MS-DOS AUX device driver typically relies on the ROM BIOS serial port driver (accessed through software Interrupt 14H) and because the ROM BIOS driver is not interrupt driven either, bypassing MS-DOS and calling the ROM BIOS functions does not usually improve matters.

Because of all the problems just described, telecommunications application programs commonly take over complete control of the serial port and supply their own interrupt handler and internal buffering for character read and write operations. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Serial port programming examples

Example: Use the ROM BIOS serial port driver to write a string to COM1.

```
msg      db       'This is a test message'
msg_len  equ      $-msg
                 .
                 .
                 .
         mov      bx,seg msg       ; DS:BX = address of message
         mov      ds,bx
         mov      bx,offset msg
         mov      cx,msg_len       ; CX = length of message
         mov      dx,0             ; DX = 0 for COM1
L1:      mov      al,[bx]          ; get next character into AL
         mov      ah,01h           ; subfunction 01H = output
         int      14h              ; transfer to ROM BIOS
         inc      bx               ; bump pointer to output string
         loop     L1               ; and loop until all chars. sent
```

Example: Use the MS-DOS traditional function for auxiliary device output to write a string to COM1.

```
msg      db       'This is a test message'
msg_len  equ      $-msg
                 .
                 .
                 .
         mov      bx,seg msg       ; set DS:BX = address of message
         mov      ds,bx
         mov      bx,offset msg
         mov      cx,msg_len       ; set CX = length of message
L1:      mov      dl,[bx]          ; get next character into DL
         mov      ah,04h           ; function 04H = auxiliary output
         int      21h              ; transfer to MS-DOS
         inc      bx               ; bump pointer to output string
         loop     L1               ; and loop until all chars. sent
```

Example: Use the MS-DOS handle-based Write File or Device function and the predefined handle for the standard auxiliary device to write a string to COM1.

```
msg      db       'This is a test message'
msg_len  equ      $-msg
                 .
                 .
                 .
         mov      dx,seg msg       ; DS:DX = address of message
         mov      ds,dx
         mov      dx,offset msg
         mov      cx,msg_len       ; CX = length of message
         mov      bx,3             ; BX = handle for standard aux.
         mov      ah,40h           ; function 40H = write file/device
         int      21h              ; transfer to MS-DOS
         jc       error            ; jump if write operation failed
```

The parallel port and printer

Most MS-DOS implementations contain device drivers for four printer devices: LPT1, LPT2, LPT3, and PRN. PRN is ordinarily an alias for LPT1 and refers to the first parallel output port in the system. To provide for list devices that do not have a parallel interface, the LPT devices can be individually redirected with the MODE command to one of the serial communications ports. *See* USER COMMANDS: MODE.

As with the keyboard, the display, and the serial port, MS-DOS allows the printer to be accessed with either traditional or handle-based function calls. The traditional function call is Interrupt 21H Function 05H, which accepts a character in DL and sends it to the physical device currently assigned to logical device name LPT1.

A program can perform handle-based output to the printer with Interrupt 21H Function 40H (Write File or Device). The predefined handle for the standard printer (4) can be used to send strings to logical device LPT1. Alternatively, the program can issue an open operation for a specific printer device with Interrupt 21H Function 3DH and use the handle obtained from that open operation with Function 40H. This latter method also allows more than one printer to be used at a time from the same program.

Because the parallel ports are assumed to be output only, no traditional call exists for input from the parallel port. In addition, no portable method exists to test printer port status under MS-DOS; programs that wish to avoid sending a character to the printer adapter when it is not ready or not physically present in the system must test the adapter's status by making a call to the ROM BIOS printer driver (by means of software Interrupt 17H; *see* Appendix O: IBM PC BIOS Calls) or by accessing the hardware directly.

Parallel port programming examples

Example: Use the ROM BIOS printer driver to send a string to the first parallel printer port.

```
msg       db       'This is a test message'
msg_len   equ      $-msg
          .
          .
          .
          mov      bx,seg msg       ; DS:BX = address of message
          mov      ds,bx
          mov      bx,offset msg
          mov      cx,msg_len       ; CX = length of message
          mov      dx,0             ; DX = 0 for LPT1
L1:       mov      al,[bx]          ; get next character into AL
          mov      ah,00h           ; subfunction 00H = output
          int      17h              ; transfer to ROM BIOS
          inc      bx               ; bump pointer to output string
          loop     L1               ; and loop until all chars. sent
```

Example: Use the traditional MS-DOS function call to send a string to the first parallel printer port.

```
msg       db      'This is a test message'
msg_len equ      $-msg
          .
          .
          .
          mov     bx,seg msg      ; DS:BX = address of message
          mov     ds,bx
          mov     bx,offset msg
          mov     cx,msg_len      ; CX = length of message
L1:       mov     dl,[bx]         ; get next character into DL
          mov     ah,05h          ; function 05H = printer output
          int     21h             ; transfer to MS-DOS
          inc     bx              ; bump pointer to output string
          loop    L1              ; and loop until all chars. sent
```

Example: Use the handle-based MS-DOS Write File or Device call and the predefined handle for the standard printer to send a string to the system list device.

```
msg       db      'This is a test message'
msg_len equ      $-msg
          .
          .
          .
          mov     dx,seg msg      ; DS:DX = address of message
          mov     ds,dx
          mov     dx,offset msg
          mov     cx,msg_len      ; CX = length of message
          mov     bx,4            ; BX = handle for standard printer
          mov     ah,40h          ; function 40H = write file/device
          int     21h             ; transfer to MS-DOS
          jc      error           ; jump if write operation failed
```

IOCTL

In versions 2.0 and later, MS-DOS has provided applications with the ability to communicate directly with device drivers through a set of subfunctions grouped under Interrupt 21H Function 44H (IOCTL). *See* SYSTEM CALLS: INTERRUPT 21H: Function 44H. The IOCTL subfunctions that are particularly applicable to the character I/O needs of application programs are

Subfunction	Name
00H	Get Device Data
01H	Set Device Data
02H	Receive Control Data from Character Device

(more)

Subfunction	Name
03H	Send Control Data to Character Device
06H	Check Input Status
07H	Check Output Status
0AH	Check if Handle is Remote (version 3.1 or later)
0CH	Generic I/O Control for Handles: Get/Set Output Iteration Count

Various bits in the device information word returned by Subfunction 00H can be tested by an application to determine whether a specific handle is associated with a character device or a file and whether the driver for the device can process control strings passed by Subfunctions 02H and 03H. The device information word also allows the program to test whether a character device is the CLOCK$, standard input, standard output, or NUL device and whether the device is in raw or cooked mode. The program can then use Subfunction 01H to select raw mode or cooked mode for subsequent I/O performed with the handle.

Subfunctions 02H and 03H allow control strings to be passed between the device driver and an application; they do not usually result in any physical I/O to the device. For example, a custom device driver might allow an application program to configure the serial port by writing a specific set of control parameters to the driver with Subfunction 03H. Similarly, the custom driver might respond to Subfunction 02H by passing the application a series of bytes that defines the current configuration and status of the serial port.

Subfunctions 06H and 07H can be used by application programs to test whether a device is ready to accept an output character or has a character ready for input. These subfunctions are particularly applicable to the serial communications ports and parallel printer ports because MS-DOS does not supply traditional function calls to test their status.

Subfunction 0AH can be used to determine whether the character device associated with a handle is local or remote — that is, attached to the computer the program is running on or attached to another computer on a local area network. A program should not ordinarily attempt to distinguish between local and remote devices during normal input and output, but the information can be useful in attempts to recover from error conditions. This subfunction is available only if Microsoft Networks is running.

Finally, Subfunction 0CH allows a program to query or set the number of times a device driver tries to send output to the printer before assuming the device is not available.

IOCTL programming examples

Example: Use IOCTL Subfunction 00H to obtain the device information word for the standard input handle and save it, and then use Subfunction 01H to place standard input into raw mode.

```
info    dw      ?               ; save device information word here
        .
        .
        .
```

(more)

```
        mov     ax,4400h        ; AH = function 44H, IOCTL
                                ; AL = subfunction 00H, get device
                                ; information word
        mov     bx,0            ; BX = handle for standard input
        int     21h             ; transfer to MS-DOS
        mov     info,dx         ; save device information word
                                ; (assumes DS = data segment)
        or      dl,20h          ; set raw mode bit
        mov     dh,0            ; and clear DH as MS-DOS requires
        mov     ax,4401h        ; AL = subfunction 01H, set device
                                ; information word
                                ; (BX still contains handle)
        int     21h             ; transfer to MS-DOS
```

Example: Use IOCTL Subfunction 06H to test whether a character is ready for input on the first serial port. The function returns AL = 0FFH if a character is ready and AL = 00H if not.

```
        mov     ax,4406H        ; AH = function 44H, IOCTL
                                ; AL = subfunction 06H, get
                                ; input status
        mov     bx,3            ; BX = handle for standard aux
        int     21h             ; transfer to MS-DOS
        or      al,al           ; test status of AUX driver
        jnz     ready           ; jump if input character ready
                                ; else no character is waiting
```

Jim Kyle
Chip Rabinowitz

Article 6
Interrupt-Driven Communications

In the earliest days of personal-computer communications, when speeds were no faster than 300 bits per second, primitive programs that moved characters to and from the remote system were adequate. The PC had time between characters to determine what it ought to do next and could spend that time keeping track of the status of the remote system.

Modern data-transfer rates, however, are four to eight times faster and leave little or no time to spare between characters. At 1200 bits per second, as many as three characters can be lost in the time required to scroll the display up one line. At such speeds, a technique to permit characters to be received and simultaneously displayed becomes necessary.

Mainframe systems have long made use of hardware interrupts to coordinate such activities. The processor goes about its normal activity; when a peripheral device needs attention, it sends an interrupt request to the processor. The processor interrupts its activity, services the request, and then goes back to what it was doing. Because the response is driven by the request, this type of processing is known as interrupt-driven. It gives the effect of doing two things at the same time without requiring two separate processors.

Successful telecommunication with PCs at modern data rates demands an interrupt-driven routine for data reception. This article discusses in detail the techniques for interrupt-driven communications and culminates in two sample program packages.

The article begins by establishing the purpose of communications programs and then discusses the capability of the simple functions provided by MS-DOS to achieve this goal. To see what must be done to supplement MS-DOS functions, the hardware (both the modem and the serial port) is examined. This leads to a discussion of the method MS-DOS has provided since version 2.0 for solving the problems of special hardware interfacing: the installable device driver.

With the background established, alternate paths to interrupt-driven communications are discussed — one following recommended MS-DOS techniques, the other following standard industry practice — and programs are developed for each.

Throughout this article, the discussion is restricted to the architecture and BIOS of the IBM PC family. MS-DOS systems not totally compatible with this architecture may require substantially different approaches at the detailed level, but the same general principles apply.

Purpose of Communications Programs

The primary purpose of any communications program is communicating — that is, transmitting information entered as keystrokes (or bytes read from a file) in a form suitable for

transmission to a remote computer via phone lines and, conversely, converting information received from the remote computer into a display on the video screen (or data in a file).

Some years ago, the most abstract form of all communications programs was dubbed a modem engine, by analogy to Babbage's analytical engine or the inference-engine model used in artificial-intelligence development. The functions of the modem engine are common to all kinds of communications programs, from the simplest to the most complex, and can be described in a type of pseudo-C as follows:

```
The Modem Engine Pseudocode

        DO {   IF (input character is available)
                   send_it_to_remote;
               IF (remote character is available)
                   use_it_locally;
        } UNTIL (told_to_stop);
```

The essence of this modem-engine code is that the absence of an input character, or of a character from the remote computer, does not hang the loop in a wait state. Rather, the engine continues to cycle: If it finds work to do, it does it; if not, the engine keeps looking.

Of course, at times it is desirable to halt the continuous action of the modem engine. For example, when receiving a long message, it is nice to be able to pause and read the message before the lines scroll into oblivion. On the other hand, taking too long to study the screen means that incoming characters are lost. The answer is a technique called flow control, in which a special control character is sent to shut down transmission and some other character is later sent to start it up again.

Several conventions for flow control exist. One of the most widespread is known as XON/XOFF, from the old Teletype-33 keycap legends for the two control codes involved. In the original use, XOFF halted the paper tape reader and XON started it going again. In mid-1967, the General Electric Company began using these signals in its time-sharing computer services to control the flow of data, and the practice rapidly spread throughout the industry.

The sample program named ENGINE, shown later in this article, is an almost literal implementation of the modem-engine approach. This sample represents one extreme of simplicity in communications programs. The other sample program, CTERM.C, is much more complex, but the modem engine is still at its heart.

Using Simple MS-DOS Functions

Because MS-DOS provides, among its standard service functions, the capability of sending output to or reading input from the device named AUX (which defaults to COM1, the first

serial port on most machines), a first attempt at implementing the modem engine using MS-DOS functions might look something like the following incomplete fragment of Microsoft Macro Assembler (MASM) code:

```
;Incomplete (and Unworkable) Implementation

LOOP:   MOV     AH,08h          ; read keyboard, no echo
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,04h          ; send to AUX device
        INT     21h
        MOV     AH,03h          ; read from AUX device
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,02h          ; send to screen
        INT     21h
        JMP     LOOP            ; keep doing it
```

The problem with this code is that it violates the keep-looking principle both at the keyboard and at the AUX port: Interrupt 21H Function 08H does not return until a keyboard character is available, so no data from the AUX port can be read until a key is pressed locally. Similarly, Function 03H waits for a character to become available from AUX, so no more keys can be recognized locally until the remote system sends a character. If nothing is received, the loop waits forever.

To overcome the problem at the keyboard end, Function 0BH can be used to determine if a key has been pressed before an attempt is made to read one, as shown in the following modification of the fragment:

```
;Improved, (but Still Unworkable) Implementation

LOOP:   MOV     AH,0Bh          ; test keyboard for char
        INT     21h
        OR      AL,AL           ; test for zero
        JZ      RMT             ; no char avail, skip
        MOV     AH,08h          ; have char, read it in
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,04h          ; send to AUX device
        INT     21h
RMT:
        MOV     AH,03h          ; read from AUX device
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,02h          ; send to screen
        INT     21h
        JMP     LOOP            ; keep doing it
```

This code permits any input from AUX to be received without waiting for a local key to be pressed, but if AUX is slow about providing input, the program waits indefinitely before checking the keyboard again. Thus, the problem is only partially solved.

MS-DOS, however, simply does not provide any direct method of making the required tests for AUX or, for that matter, any of the serial port devices. That is why communications programs must be treated differently from most other types of programs under MS-DOS and why such programs must be intimately involved with machine details despite all accepted principles of portable program design.

The Hardware Involved

Personal-computer communications require at least two distinct pieces of hardware (separate devices, even though they are often combined on a single board). These hardware items are the serial port, which converts data from the computer's internal bus into a bit stream for transmission over a single external line, and the modem, which converts the bit stream into a form suitable for telephone-line (or, sometimes, radio) transmission.

The modem

The modem (a word coined from MOdulator-DEModulator) is a device that converts a stream of bits, represented as sequential changes of voltage level, into audio frequency signals suitable for transmission over voice-grade telephone circuits (modulation) and converts these signals back into a stream of bits that duplicates the original input (demodulation).

Specific characteristics of the audio signals involved were established by AT&T when that company monopolized the modem industry, and those characteristics then evolved into de facto standards when the monopoly vanished. They take several forms, depending on the data rate in use; these forms are normally identified by the original Bell specification number, such as 103 (for 600 bps and below) or 212A (for the 1200 bps standard).

The data rate is measured in bits per second (bps), often mistermed baud or even "baud per second." A baud measures the number of signals per second; as with knot (nautical miles per hour), the time reference is built in. If one signal change marks one bit, as is true for the Bell 103 standard, then baud and bps have equal values. However, they are not equivalent for more complex signals. For example, the Bell 212A diphase standard for 1200 bps uses two tone streams, each operating at 600 baud, to transmit data at 1200 bits per second.

For accuracy, this article uses bps, rather than baud, except where widespread industry misuse of baud has become standardized (as in "baud rate generator").

Originally, the modem itself was a box connected to the computer's serial port via a cable. Characteristics of this cable, its connectors, and its signals were standardized in the 1960s by the Electronic Industries Association (EIA), in Standard RS232C. Like the Bell standards for modems, RS232C has survived almost unchanged. Its characteristics are listed in Table 6-1.

Table 6-1. RS232C Signals.

DB25 Pin	232	Name	Description
1			Safety Ground
2	BA	TXD	Transmit Data
3	BB	RXD	Receive Data
4	CA	RTS	Request To Send
5	CB	CTS	Clear To Send
6	CC	DSR	Data Set Ready
7	AB	GND	Signal Ground
8	CF	DCD	Data Carrier Detected
20	CD	DTR	Data Terminal Ready
22	CE	RI	Ring Indicator

With the increasing popularity of personal computers, internal modems that plug into the PC's motherboard and combine the modem and a serial port became available.

The first such units were manufactured by Hayes Corporation, and like Bell and the EIA, they created a standard. Functionally, the internal modem is identical to the combination of a serial port, a connecting cable, and an external modem.

The serial port

Each serial port of a standard IBM PC connects the rest of the system to a type INS8250 Universal Asynchronous Receiver Transmitter (UART) integrated circuit (IC) chip developed by National Semiconductor Corporation. This chip, along with associated circuits in the port,

1. Converts data supplied via the system data bus into a sequence of voltage levels on the single TXD output line that represent binary digits.
2. Converts data received as a sequence of binary levels on the single RXD input line into bytes for the data bus.
3. Controls the modem's actions through the DTR and RTS output lines.
4. Provides status information to the processor; this information comes from the modem, via the DSR, DCD, CTS, and RI input lines, and from within the UART itself, which signals data available, data needed, or error detected.

The word *asynchronous* in the name of the IC comes from the Bell specifications. When computer data is transmitted, each bit's relationship to its neighbors must be preserved; this can be done in either of two ways. The most obvious method is to keep the bit stream strictly synchronized with a clock signal of known frequency and count the cycles to identify the bits. Such a transmission is known as synchronous, often abbreviated to synch or sometimes bisync for binary synchronous. The second method, first used with mechanical teleprinters, marks the start of each bit group with a defined start bit and the end with one or more defined stop bits, and it defines a duration for each bit time. Detection of a start bit

marks the beginning of a received group; the signal is then sampled at each bit time until the stop bit is encountered. This method is known as asynchronous (or just asynch) and is the one used by the standard IBM PC.

The start bit is, by definition, exactly the same as that used to indicate binary zero, and the stop bit is the same as that indicating binary one. A zero signal is often called SPACE, and a one signal is called MARK, from terms used in the teleprinter industry.

During transmission, the least significant bit of the data is sent first, after the start bit. A parity bit, if used, appears as the most significant bit in the data group, before the stop bit or bits; it cannot be distinguished from a databit except by its position. Once the first stop bit is sent, the line remains in MARK (sometimes called idling) condition until a new start bit indicates the beginning of another group.

In most PC uses, the serial port transfers one 8-bit byte at a time, and the term *word* specifies a 16-bit quantity. In the UART world, however, a word is the unit of information sent by the chip in each chunk. The word length is part of the control information set into the chip during setup operations and can be 5, 6, 7, or 8 bits. This discussion follows UART conventions and refers to words, rather than to bytes.

One special type of signal, not often used in PC-to-PC communications but sometimes necessary in communicating with mainframe systems, is a BREAK. The BREAK is an all-SPACE condition that extends for more than one word time, including the stop-bit time. (Many systems require the BREAK to last at least 150 milliseconds regardless of data rate.) Because it cannot be generated by any normal data character transmission, the BREAK is used to interrupt, or break into, normal operation. The IBM PC's 8250 UART can generate the BREAK signal, but its duration must be determined by a program, rather than by the chip.

The 8250 UART architecture

The 8250 UART contains four major functional areas: receiver, transmitter, control circuits, and status circuits. Because these areas are closely related, some terms used in the following descriptions are, of necessity, forward references to subsequent paragraphs.

The major parts of the receiver are a shift register and a data register called the Received Data Register. The shift register assembles sequentially received data into word-parallel form by shifting the level of the RXD line into its front end at each bit time and, at the same time, shifting previous bits over. When the shift register is full, all bits in it are moved over to the data register, the shift register is cleared to all zeros, and the bit in the status circuits that indicates data ready is set. If an error is detected during receipt of that word, other bits in the status circuits are also set.

Similarly, the major parts of the transmitter are a holding register called the Transmit Holding Register and a shift register. Each word to be transmitted is transferred from the

data bus to the holding register. If the holding register is not empty when this is done, the previous contents are lost. The transmitter's shift register converts word-parallel data into bit-serial form for transmission by shifting the most significant bit out to the TXD line once each bit time, at the same time shifting lower bits over and shifting in an idling bit at the low end of the register. When the last databit has been shifted out, any data in the holding register is moved to the shift register, the holding register is filled with idling bits in case no more data is forthcoming, and the bit in the status circuits that indicates the Transmit Holding Register is empty is set to indicate that another word can be transferred. The parity bit, if any, and stop bits are added to the transmitted stream after the last databit of each word is shifted out.

The control circuits establish three communications features: first, line control values, such as word length, whether or not (and how) parity is checked, and the number of stop bits; second, modem control values, such as the state of the DTR and RTS output lines; and third, the rate at which data is sent and received. These control values are established by two 8-bit registers and one 16-bit register, which are addressed as four 8-bit registers. They are the Line Control Register (LCR), the Modem Control Register (MCR), and the 16-bit BRG Divisor Latch, addressed as Baud0 and Baud1.

The BRG Divisor Latch sets the data rate by defining the bit time produced by the Programmable Baud Rate Generator (PBRG), a major part of the control circuits. The PBRG can provide any data speed from a few bits per second to 38400 bps; in the BIOS of the IBM PC, PC/XT, and PC/AT, though, only the range 110 through 9600 bps is supported. How the LCR and the MCR establish their control values, how the PBRG is programmed, and how interrupts are enabled are discussed later.

The fourth major area in the 8250 UART, the status circuits, records (in a pair of status registers) the conditions in the receive and transmit circuits, any errors that are detected, and any change in state of the RS232C input lines from the modem. When any status register's content changes, an interrupt request, if enabled, is generated to notify the rest of the PC system. This approach lets the PC attend to other matters without having to continually monitor the status of the serial port, yet it assures immediate action when something does occur.

The 8250 programming interface

Not all the registers mentioned in the preceding section are accessible to programmers. The shift registers, for example, can be read from or written to only by the 8250's internal circuits. There are 10 registers available to the programmer, and they are accessed by only seven distinct addresses (shown in Table 6-2). The Received Data Register and the Transmit Holding Register share a single address (a read gets the received data; a write goes to the holding register). In addition, both this address and that of the Interrupt Enable Register (IER) are shared with the PBRG Divisor Latch. A bit in the Line Control Register called the Divisor Latch Access Bit (DLAB) determines which register is addressed at any specific time.

ZTE (USA) 1007, Page 185

In the IBM PC, the seven addresses used by the 8250 are selected by the low 3 bits of the port number (the higher bits select the specific port). Thus, each serial port occupies eight positions in the address space. However, only the lowest address used — the one in which the low 3 bits are all 0 — need be remembered in order to access all eight addresses.

Because of this, any serial port in the PC is referred to by an address that, in hexadecimal notation, ends with either 0 or 8: The COM1 port normally uses address 03F8H, and COM2 uses 02F8H. This lowest port address is usually called the base port address, and each addressable register is then referenced as an offset from this base value, as shown in Table 6-2.

Table 6-2. 8250 Port Offsets from Base Address.

Offset	Name	Description
If DLAB bit in LCR = 0:		
00H	DATA	Received Data Register if read from, Transmit Holding Register if written to
01H	IER	Interrupt Enable Register
If DLAB bit in LCR = 1:		
00H	Baud0	BRG Divisor Latch, low byte
01H	Baud1	BRG Divisor Latch, high byte
Not affected by DLAB bit:		
02H	IID	Interrupt Identifier Register
03H	LCR	Line Control Register
04H	MCR	Modem Control Register
05H	LSR	Line Status Register
06H	MSR	Modem Status Register

The control circuits

The control circuits of the 8250 include the Programmable Baud Rate Generator (PBRG), the Line Control Register (LCR), the Modem Control Register (MCR), and the Interrupt Enable Register (IER).

The PBRG establishes the bit time used for both transmitting and receiving data by dividing an external clock signal. To select a desired bit rate, the appropriate divisor is loaded into the PBRG's 16-bit Divisor Latch by setting the Divisor Latch Access Bit (DLAB) in the Line Control Register to 1 (which changes the functions of addresses 0 and 1) and then writing the divisor into Baud0 and Baud1. After the bit rate is selected, DLAB is changed back to 0, to permit normal operation of the DATA registers and the IER.

With the 1.8432 MHz external UART clock frequency used in standard IBM systems, divisor values (in decimal notation) for bit rates between 45.5 and 38400 bps are listed in Table 6-3. These speeds are established by a crystal contained in the serial port (or internal modem) and are totally unrelated to the speed of the processor's clock.

Table 6-3. Bit Rate Divisor Table for 8250/IBM.

BPS	Divisor
45.5	2532
50	2304
75	1536
110	1047
134.5	857
150	768
300	384
600	192
1200	96
1800	64
2000	58
2400	48
4800	24
9600	12
19200	6
38400	3

The remaining control circuits are the Line Control Register, the Modem Control Register, and the Interrupt Enable Register. Bits in the LCR control the assignment of offsets 0 and 1, transmission of the BREAK signal, parity generation, the number of stop bits, and the word length sent and received, as shown in Table 6-4.

Table 6-4. 8250 Line Control Register Bit Values.

Bit	Name	Binary	Meaning
Address Control:			
7	DLAB	0xxxxxxx	Offset 0 refers to DATA; offset 1 refers to IER
		1xxxxxxx	Offsets 0 and 1 refer to BRG Divisor Latch
BREAK Control:			
6	SETBRK	x0xxxxxx	Normal UART operation
		x1xxxxxx	Send BREAK signal

(more)

Table 6-4. *Continued.*

Bit	Name	Binary	Meaning
Parity Checking:			
5,4,3	GENPAR	xxxx0xxx	No parity bit
		xx001xxx	Parity bit is ODD
		xx011xxx	Parity bit is EVEN
		xx101xxx	Parity bit is 1
		xx111xxx	Parity bit is 0
Stop Bits:			
2	XSTOP	xxxxx0xx	Only 1 stop bit
		xxxxx1xx	2 stop bits (1.5 if WL = 5)
Word Length:			
1,0	WD5	xxxxxx00	Word length = 5
	WD6	xxxxxx01	Word length = 6
	WD7	xxxxxx10	Word length = 7
	WD8	xxxxxx11	Word length = 8

Two bits in the MCR (Table 6-5) control output lines DTR and RTS; two other MCR bits (OUT1 and OUT2) are left free by the UART to be assigned by the user; a fifth bit (TEST) puts the UART into a self-test mode of operation. The upper 3 bits have no effect on the UART. The MCR can be both read from and written to.

Both of the user-assignable bits are defined in the IBM PC. OUT1 is used by Hayes internal modems to cause a power-on reset of their circuits; OUT2 controls the passage of UART-generated interrupt request signals to the rest of the PC. Unless OUT2 is set to 1, interrupt signals from the UART cannot reach the rest of the PC, even though all other controls are properly set. This feature is documented, but obscurely, in the IBM *Technical Reference* manuals and the asynchronous-adapter schematic; it is easy to overlook when writing an interrupt-driven program for these machines.

Table 6-5. 8250 Modem Control Register Bit Values.

Name	Binary	Description
TEST	xxx1xxxx	Turns on UART self-test configuration.
OUT2	xxxx1xxx	Controls 8250 interrupt signals (User2 Output).
OUT1	xxxxx1xx	Resets Hayes 1200b internal modem (User1 Output).
RTS	xxxxxx1x	Sets RTS output to RS232C connector.
DTR	xxxxxxx1	Sets DTR output to RS232C connector.

The 8250 can generate any or all of four classes of interrupts, each individually enabled or disabled by setting the appropriate control bit in the Interrupt Enable Register (Table 6-6). Thus, setting the IER to 00H disables all the UART interrupts within the 8250 without regard to any other settings, such as OUT2, system interrupt masking, or the CLI/STI commands. The IER can be both read from and written to. Only the low 4 bits have any effect on the UART.

Table 6-6. 8250 Interrupt Enable Register Constants.

Binary	Action
xxxx1xxx	Enable Modem Status Interrupt.
xxxxx1xx	Enable Line Status Interrupt.
xxxxxx1x	Enable Transmit Register Interrupt.
xxxxxxx1	Enable Received Data Ready Interrupt.

The status circuits

The status circuits of the 8250 include the Line Status Register (LSR), the Modem Status Register (MSR), the Interrupt Identifier (IID) Register, and the interrupt-request generation system.

The 8250 includes circuitry that detects a received BREAK signal and also detects three classes of data-reception errors. Separate bits in the LSR (Table 6-7) are set to indicate that a BREAK has been received and to indicate any of the following: a parity error (if lateral parity is in use), a framing error (incoming bit = 0 at stop-bit time), or an overrun error (word not yet read from receive buffer by the time the next word must be moved into it).

The remaining bits of the LSR indicate the status of the Transmit Shift Register, the Transmit Holding Register, and the Received Data Register; the most significant bit of the LSR is not used and is always 0. The LSR is a read-only register; writing to it has no effect.

Table 6-7. 8250 Line Status Register Bit Values.

Bit	Binary	Meaning
7	0xxxxxxx	Always zero
6	x1xxxxxx	Transmit Shift Register empty
5	xx1xxxxx	Transmit Holding Register empty
4	xxx1xxxx	BREAK received
3	xxxx1xxx	Framing error
2	xxxxx1xx	Parity error
1	xxxxxx1x	Overrun error
0	xxxxxxx1	Received data ready

```
CLRGS:
        MOV     DX,03FDh        ; clear LSR
        IN      AL,DX
        MOV     DX,03F8h        ; clear RX reg
        IN      AL,DX
        MOV     DX,03FEh        ; clear MSR
        IN      AL,DX
        MOV     DX,03FAh        ; IID reg
        IN      AL,DX
        IN      AL,DX           ; repeat to be sure
        TEST    AL,1            ; int pending?
        JZ      CLRGS           ; yes, repeat
```

Note: This code does not completely set up the IBM serial port. Although it fully programs
the 8250 itself, additional work remains to be done. The system interrupt vectors must be
changed to provide linkage to the interrupt service routine (ISR) code, and the 8259
Priority Interrupt Controller (PIC) chip must also be programmed to respond to interrupt
requests from the UART channels. *See* PROGRAMMING IN THE MS-DOS ENVIRON-
MENT: CUSTOMIZING MS-DOS: Hardware Interrupt Handlers.

Device Drivers

All versions of MS-DOS since 2.0 have permitted the installation of user-provided device
drivers. From the standpoint of operating-system theory, using such drivers is the proper
way to handle generic communications interfacing. The following paragraphs are intended
as a refresher and to explain this article's departure from standard device-driver terminol-
ogy. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS:
Installable Device Drivers.

An installable device driver consists of (1) a driver header that links the driver to
others in the chain maintained by MS-DOS, tells the system the characteristics of this spe-
cific driver, provides pointers to the two major routines contained in the driver, and (for a
character-device driver) identifies the driver by name; (2) any data and storage space the
driver may require; and (3) the two major code routines.

The code routines are called the Strategy routine and the Interrupt routine in normal
device-driver descriptions. Neither has any connection with the hardware interrupts dealt
with by the drivers presented in this article. Because of this, the term Request routine is
used instead of Interrupt routine, so that hardware interrupt code can be called an
interrupt service routine (ISR) with minimal chances for confusion.

MS-DOS communicates with a device driver by reserving space for a command packet
of as many as 22 bytes and by passing this packet's address to the driver with a call to the
Strategy routine. All data transfer between MS-DOS and the driver, in both directions,
occurs via this command packet and the Request routine. The operating system places a
command code and, optionally, a byte count and a buffer address into the packet at the
specified locations, then calls the Request routine. The driver performs the command
and returns the status (and sometimes a byte count) in the packet.

Two Alternative Approaches

Now that the factors involved in creating interrupt-driven communications programs have been discussed, they can be put together into practical program packages. Doing so brings out not only general principles but also minor details that make the difference between success and failure of program design in this hardware-dependent and time-critical area.

The traditional way: Going it alone

Because MS-DOS provides no generic functions suitable for communications use, virtually all popular communications programs provide and install their own port driver code, and then remove it before returning to MS-DOS. This approach entails the creation of a communications handler for each program and requires the "uninstallation" of the handler on exit from the program that uses it. Despite the extra requirements, most communications programs use this method.

The alternative: Creating a communications device driver

Instead of providing temporary interface code that must be removed from the system before returning to the command level, an installable device driver can be built as a replacement for COM*x* so that every program can have all features. However, this approach is not compatible with existing terminal programs because it has never been a part of MS-DOS.

Comparison of the two methods

The traditional approach has several advantages, the most obvious being that the driver code can be fully tailored to the needs of the program. Because only one program will ever use the driver, no general cases need be considered.

However, if a user wants to keep communications capability available in a terminate-and-stay-resident (TSR) module for background use and also wants a different type of communications program running in the foreground (not, of course, while the background task is using the port), the background program and the foreground job must each have its own separate driver code. And, because such code usually includes buffer areas, the duplicated drivers represent wasted resources.

A single communications device driver that is installed when the system powers up and that remains active until shutdown avoids wasting resources by allowing both the background and foreground tasks to share the driver code. Until such drivers are common, however, it is unlikely that commercial software will be able to make use of them. In addition, such a driver must either provide totally general capabilities or it must include control interfaces so each user program can dynamically alter the driver to suit its needs.

At this time, the use of a single driver is an interesting exercise rather than a practical application, although a possible exception is a dedicated system in which all software is either custom designed or specially modified. In such a system, the generalized driver can provide significant improvement in the efficiency of resource allocation.

A Device-Driver Program Package

Despite the limitations mentioned in the preceding section, the first of the two complete packages in this article uses the concept of a separate device driver. The driver handles all hardware-dependent interfacing and thus permits extreme simplicity in all other modules of the package. This approach is presented first because it is especially well suited for introducing the concepts of communications programs. However, the package is not merely a tutorial device: It includes some features that are not available in most commercial programs.

The package itself consists of three separate programs. First is the device driver, which becomes a part of MS-DOS via the CONFIG.SYS file. Second is the modem engine, which is the actual terminal program. (A functionally similar component forms the heart of every communications program, whether it is written in assembly language or a high-level language and regardless of the machine or operating system in use.) Third is a separately executed support program that permits changing such driver characteristics as word length, parity, and baud rate.

In most programs that use the traditional approach, the driver and the support program are combined with the modem engine in a single unit and the resulting mass of detail obscures the essential simplicity of each part. Here, the parts are presented as separate modules to emphasize that simplicity.

The device driver: COMDVR.ASM

The device driver is written to augment the default COM1 and COM2 devices with other devices named ASY1 and ASY2 that use the same physical hardware but are logically separate. The driver (COMDVR.ASM) is implemented in MASM and is shown in the listing in Figure 6-1. Although the driver is written basically as a skeleton, it is designed to permit extensive expansion and can be used as a general-purpose sample of device-driver source code.

The code

```
 1 : Title   COMDVR  Driver for IBM COM Ports
 2 : ;        Jim Kyle, 1987
 3 : ;            Based on ideas from many sources......
 4 : ;                including Mike Higgins, CLM March 1985;
 5 : ;                public-domain INTBIOS program from BBS's;
 6 : ;                COMBIOS.COM from CIS Programmers' SIG; and
 7 : ;                ADVANCED MS-DOS by Ray Duncan.
 8 : Subttl  MS-DOS Driver Definitions
 9 :
10 :         Comment *     This comments out the Dbg macro.....
11 : Dbg     Macro   Ltr1,Ltr2,Ltr3  ; used only to debug driver...
12 :         Local   Xxx
13 :         Push    Es              ; save all regs used
```

Figure 6-1. COMDVR.ASM.   *(more)*

```
14 :          Push   Di
15 :          Push   Ax
16 :          Les    Di,Cs:Dbgptr     ; get pointer to CRT
17 :          Mov    Ax,Es:[di]
18 :          Mov    Al,Ltr1          ; move in letters
19 :          Stosw
20 :          Mov    Al,Ltr2
21 :          Stosw
22 :          Mov    Al,Ltr3
23 :          Stosw
24 :          Cmp    Di,1600          ; top 10 lines only
25 :          Jb     Xxx
26 :          Xor    Di,Di
27 : Xxx:     Mov    Word Ptr Cs:Dbgptr,Di
28 :          Pop    Ax
29 :          Pop    Di
30 :          Pop    Es
31 :          Endm
32 :          *                       ; asterisk ends commented-out region
33 : ;
34 : ;               Device Type Codes
35 : DevChr  Equ    8000h   ; this is a character device
36 : DevBlk  Equ    0000h   ; this is a block (disk) device
37 : DevIoc  Equ    4000h   ; this device accepts IOCTL requests
38 : DevNon  Equ    2000h   ; non-IBM disk driver (block only)
39 : DevOTB  Equ    2000h   ; MS-DOS 3.x out until busy supported (char)
40 : DevOCR  Equ    0800h   ; MS-DOS 3.x open/close/rm supported
41 : DevX32  Equ    0040h   ; MS-DOS 3.2 functions supported
42 : DevSpc  Equ    0010h   ; accepts special interrupt 29H
43 : DevClk  Equ    0008h   ; this is the CLOCK device
44 : DevNul  Equ    0004h   ; this is the NUL device
45 : DevSto  Equ    0002h   ; this is standard output
46 : DevSti  Equ    0001h   ; this is standard input
47 : ;
48 : ;               Error Status BITS
49 : StsErr  Equ    8000h   ; general error
50 : StsBsy  Equ    0200h   ; device busy
51 : StsDne  Equ    0100h   ; request completed
52 : ;
53 : ;               Error Reason values for lower-order bits
54 : ErrWp   Equ    0       ; write protect error
55 : ErrUu   Equ    1       ; unknown unit
56 : ErrDnr  Equ    2       ; drive not ready
57 : ErrUc   Equ    3       ; unknown command
58 : ErrCrc  Equ    4       ; cyclical redundancy check error
59 : ErrBsl  Equ    5       ; bad drive request structure length
60 : ErrSl   Equ    6       ; seek error
61 : ErrUm   Equ    7       ; unknown media
62 : ErrSnf  Equ    8       ; sector not found
63 : ErrPop  Equ    9       ; printer out of paper
64 : ErrWf   Equ    10      ; write fault
```

Figure 6-1. Continued.   *(more)*

```
 65 : ErrRf     Equ     11          ; read fault
 66 : ErrGf     Equ     12          ; general failure
 67 : ;
 68 : ;         Structure of an I/O request packet header.
 69 : ; .
 70 : Pack      Struc
 71 : Len       Db      ?           ; length of record
 72 : Prtno     Db      ?           ; unit code
 73 : Code      Db      ?           ; command code
 74 : Stat      Dw      ?           ; return status
 75 : Dosq      Dd      ?           ; (unused MS-DOS queue link pointer)
 76 : Devq      Dd      ?           ; (unused driver queue link pointer)
 77 : Media     Db      ?           ; media code on read/write
 78 : Xfer      Dw      ?           ; xfer address offset
 79 : Xseg      Dw      ?           ; xfer address segment
 80 : Count     Dw      ?           ; transfer byte count
 81 : Sector    Dw      ?           ; starting sector value (block only)
 82 : Pack      Ends
 83 :
 84 : Subttl    IBM-PC Hardware Driver Definitions
 85 : page
 86 : ;
 87 : ;                 8259 data
 88 : PIC_b     Equ     020h    ; port for EOI
 89 : PIC_e     Equ     021h    ; port for Int enabling
 90 : EOI       Equ     020h    ; EOI control word
 91 : ;
 92 : ;                 8250 port offsets
 93 : RxBuf     Equ     0F8h      ; base address
 94 : Baud1     Equ     RxBuf+1 ; baud divisor high byte
 95 : IntEn     Equ     RxBuf+1 ; interrupt enable register
 96 : IntId     Equ     RxBuf+2 ; interrupt identification register
 97 : Lctrl     Equ     RxBuf+3 ; line control register
 98 : Mctrl     Equ     RxBuf+4 ; modem control register
 99 : Lstat     Equ     RxBuf+5 ; line status register
100 : Mstat     Equ     RxBuf+6 ; modem status register
101 : ;
102 : ;                 8250 LCR constants
103 : Dlab      Equ     10000000b ; divisor latch access bit
104 : SetBrk    Equ     01000000b ; send break control bit
105 : StkPar    Equ     00100000b ; stick parity control bit
106 : EvnPar    Equ     00010000b ; even parity bit
107 : GenPar    Equ     00001000b ; generate parity bit
108 : Xstop     Equ     00000100b ; extra stop bit
109 : Wd8       Equ     00000011b ; word length = 8
110 : Wd7·      Equ     00000010b ; word length = 7
111 : Wd6       Equ     00000001b ; word length = 6
112 : ;
113 : ;                 8250 LSR constants
114 : xsre      Equ     01000000b ; xmt SR empty
115 : xhre      Equ     00100000b ; xmt HR empty
```

Figure 6-1. Continued.   *(more)*

```
116 : BrkRcv    Equ     00010000b ; break received
117 : FrmErr    Equ     00001000b ; framing error
118 : ParErr    Equ     00000100b ; parity error
119 : OveRun    Equ     00000010b ; overrun error
120 : rdta      Equ     00000001b ; received data ready
121 : AnyErr    Equ     BrkRcv+FrmErr+ParErr+OveRun
122 : ;
123 : ;                 8250 MCR constants
124 : LpBk      Equ     00010000b ; UART out loops to in (test)
125 : Usr2      Equ     00001000b ; Gates 8250 interrupts
126 : Usr1      Equ     00000100b ; aux user1 output
127 : SetRTS    Equ     00000010b ; sets RTS output
128 : SetDTR    Equ     00000001b ; sets DTR output
129 : ;
130 : ;                 8250 MSR constants
131 : CDlvl     Equ     10000000b ; carrier detect level
132 : RIlvl     Equ     01000000b ; ring indicator level
133 : DSRlvl    Equ     00100000b ; DSR level
134 : CTSlvl    Equ     00010000b ; CTS level
135 : CDchg     Equ     00001000b ; Carrier Detect change
136 : RIchg     Equ     00000100b ; Ring Indicator change
137 : DSRchg    Equ     00000010b ; DSR change
138 : CTSchg    Equ     00000001b ; CTS change
139 : ;
140 : ;                 8250 IER constants
141 : S_Int     Equ     00001000b ; enable status interrupt
142 : E_Int     Equ     00000100b ; enable error interrupt
143 : X_Int     Equ     00000010b ; enable transmit interrupt
144 : R_Int     Equ     00000001b ; enable receive interrupt
145 : AllInt    Equ     00001111b ; enable all interrupts
146 :
147 : Subttl  Definitions for THIS Driver
148 : page
149 : ;
150 : ;                 Bit definitions for the output status byte
151 : ;                         ( this driver only )
152 : LinIdl    Equ     0ffh    ; if all bits off, xmitter is idle
153 : LinXof    Equ     1       ; output is suspended by XOFF
154 : LinDSR    Equ     2       ; output is suspended until DSR comes on again
155 : LinCTS    Equ     4       ; output is suspended until CTS comes on again
156 : ;
157 : ;                 Bit definitions for the input status byte
158 : ;                         ( this driver only )
159 : BadInp    Equ     1       ; input line errors have been detected
160 : LostDt    Equ     2       ; receiver buffer overflowed, data lost
161 : OffLin    Equ     4       ; device is off line now
162 : ;
163 : ;                 Bit definitions for the special characteristics words
164 : ;                         ( this driver only )
165 : ;                 InSpec controls how input from the UART is treated
166 : ;
```

Figure 6-1. Continued.   *(more)*

```
167 : InEpc     Equ      0001h   ; errors translate to codes with parity bit on
168 : ;
169 : ;                  OutSpec controls how output to the UART is treated
170 : ;
171 : OutDSR    Equ      0001h    ; DSR is used to throttle output data
172 : OutCTS    Equ      0002h    ; CTS is used to throttle output data
173 : OutXon    Equ      0004h    ; XON/XOFF is used to throttle output data
174 : OutCdf    Equ      0010h    ; carrier detect is off-line signal
175 : OutDrf    Equ      0020h    ; DSR is off-line signal
176 : ;
177 : Unit      Struc             ; each unit has a structure defining its state:
178 : Port      Dw       ?        ; I/O port address
179 : Vect      Dw       ?        ; interrupt vector offset (NOT interrupt number!)
180 : Isradr    Dw       ?        ; offset to interrupt service routine
181 : OtStat    Db       Wd8      ; default LCR bit settings during INIT,
182 :                             ; output status bits after
183 : InStat    Db       Usr2+SetRTS+SetDTR   ; MCR bit settings during INIT,
184 :                             ; input status bits after
185 : InSpec    Dw       InEpc    ; special mode bits for INPUT
186 : OutSpec   Dw       OutXon   ; special mode bits for OUTPUT
187 : Baud      Dw       96       ; current baud rate divisor value (1200 b)
188 : Ifirst    Dw       0        ; offset of first character in input buffer
189 : Iavail    Dw       0        ; offset of next available byte
190 : Ibuf      Dw       ?        ; pointer to input buffer
191 : Ofirst    Dw       0        ; offset of first character in output buffer
192 : Oavail    Dw       0        ; offset of next avail byte in output buffer
193 : Obuf      Dw       ?        ; pointer to output buffer
194 : Unit      Ends
195 :
196 : ;
197 : ;         Beginning of driver code and data
198 : ;
199 : Driver    Segment
200 :          Assume  Cs:driver, ds:driver, es:driver
201 :          Org     0                     ; drivers start at 0
202 :
203 :          Dw      Async2,-1      ; pointer to next device
204 :          Dw      DevChr + DevIoc ; character device with IOCTL
205 :          Dw      Strtegy         ; offset of Strategy routine
206 :          Dw      Request1        ; offset of interrupt entry point 1
207 :          Db      'ASY1    '      ; device 1 name
208 : Async2:
209 :          Dw      -1,-1           ; pointer to next device: MS-DOS fills in
210 :          Dw      DevChr + DevIoc ; character device with IOCTL
211 :          Dw      Strtegy         ; offset of Strategy routine
212 :          Dw      Request2        ; offset of interrupt entry point 2
213 :          Db      'ASY2    '      ; device 2 name
214 :
215 : ;dbgptr Dd       0b0000000h
216 : ;
217 : ;         Following is the storage area for the request packet pointer
```

Figure 6-1. Continued.   *(more)*

```
218 : ;
219 : PackHd   Dd        0
220 : ;
221 : ;             baud rate conversion table
222 : Asy_baudt Dw            50,2304        ; first value is desired baud rate
223 :           Dw            75,1536        ; second is divisor register value
224 :           Dw           110,1047
225 :           Dw           134, 857
226 :           Dw           150, 786
227 :           Dw           300, 384
228 :           Dw           600, 192
229 :           Dw          1200,  96
230 :           Dw          1800,  64
231 :           Dw          2000,  58
232 :           Dw          2400,  48
233 :           Dw          3600,  32
234 :           Dw          4800,  24
235 :           Dw          7200,  16
236 :           Dw          9600,  12
237 :
238 : ; table of structures
239 : ;        ASY1 defaults to the COM1 port, INT 0CH vector, XON,
240 : ;        no parity, 8 databits, 1 stop bit, and 1200 baud
241 : Asy_tab1:
242 :           Unit    <3f8h,30h,asy1isr,,,,,,,,in1buf,,,out1buf>
243 :
244 : ;        ASY2 defaults to the COM2 port, INT 0BH vector, XON,
245 : ;        no parity, 8 databits, 1 stop bit, and 1200 baud
246 : Asy_tab2:
247 :           Unit    <2f8h,2ch,asy2isr,,,,,,,,in2buf,,,out2buf>
248 :
249 : Bufsiz   Equ     256          ; input buffer size
250 : Bufmsk   =       Bufsiz-1     ; mask for calculating offsets modulo bufsiz
251 : In1buf   Db      Bufsiz DUP (?)
252 : Out1buf  Db      Bufsiz DUP (?)
253 : In2buf   Db      Bufsiz DUP (?)
254 : Out2buf  Db      Bufsiz DUP (?)
255 : ;
256 : ;              Following is a table of offsets to all the driver functions
257 :
258 : Asy_funcs:
259 :           Dw      Init       ;  0 initialize driver
260 :           Dw      Mchek      ;  1 media check (block only)
261 :           Dw      BldBPB     ;  2 build BPB (block only)
262 :           Dw      Ioctlin    ;  3 IOCTL read
263 :           Dw      Read       ;  4 read
264 :           Dw      Ndread     ;  5 nondestructive read
265 :           Dw      Rxstat     ;  6 input status
266 :           Dw      Inflush    ;  7 flush input buffer
267 :           Dw      Write      ;  8 write
268 :           Dw      Write      ;  9 write with verify
```

(more)

Figure 6-1. Continued.

```
269 :           Dw      Txstat          ; 10 output status
270 :           Dw      Txflush         ; 11 flush output buffer
271 :           Dw      Ioctlout        ; 12 IOCTL write
272 : ; Following are not used in this driver.....
273 :           Dw      Zexit           ; 13 open (3.x only, not used)
274 :           Dw      Zexit           ; 14 close (3.x only, not used)
275 :           Dw      Zexit           ; 15 rem med (3.x only, not used)
276 :           Dw      Zexit           ; 16 out until bsy (3.x only, not used)
277 :           Dw      Zexit           ; 17
278 :           Dw      Zexit           ; 18
279 :           Dw      Zexit           ; 19 generic IOCTL request (3.2 only)
280 :           Dw      Zexit           ; 20
281 :           Dw      Zexit           ; 21
282 :           Dw      Zexit           ; 22
283 :           Dw      Zexit           ; 23 get logical drive map (3.2 only)
284 :           Dw      Zexit           ; 24 set logical drive map (3.2 only)
285 :
286 : Subttl   Driver Code
287 : Page
288 : ;
289 : ;         The Strategy routine itself:
290 : ;
291 : Strtegy Proc    Far
292 : ;         dbg     'S','R',' '
293 :           Mov     Word Ptr CS:PackHd,BX    ; store the offset
294 :           Mov     Word Ptr CS:PackHd+2,ES ; store the segment
295 :           Ret
296 : Strtegy Endp
297 : ;
298 : Request1:                         ; async1 has been requested
299 :           Push    Si              ; save SI
300 :           Lea     Si,Asy_tab1     ; get the device unit table address
301 :           Jmp     Short   Gen_request
302 :
303 : Request2:                         ; async2 has been requested
304 :           Push    Si              ; save SI
305 :           Lea     Si,Asy_tab2     ; get unit table two's address
306 :
307 : Gen_request:
308 : ;         dbg     'R','R',' '
309 :           Pushf                   ; save all regs
310 :           Cld
311 :           Push    Ax
312 :           Push    Bx
313 :           Push    Cx
314 :           Push    Dx
315 :           Push    Di
316 :           Push    Bp
317 :           Push    Ds
318 :           Push    Es
319 :           Push    Cs              ; set DS = CS
```

Figure 6-1. Continued.   *(more)*

```
320 :          Pop     Ds
321 :          Les     Bx,PackHd       ; get packet pointer
322 :          Lea     Di,Asy_funcs    ; point DI to jump table
323 :          Mov     Al,es:code[bx]  ; command code
324 :          Cbw
325 :          Add     Ax,Ax           ; double to word
326 :          Add     Di,ax
327 :          Jmp     [di]            ; go do it
328 : ;
329 : ;       Exit from driver request
330 : ;
331 : ExitP   Proc    Far
332 : Bsyexit:
333 :          Mov     Ax,StsBsy
334 :          Jmp     Short   Exit
335 :
336 : Mchek:
337 : BldBPB:
338 : Zexit:  Xor     Ax,Ax
339 : Exit:   Les     Bx,PackHd       ; get packet pointer
340 :          Or      Ax,StsDne
341 :          Mov     Es:Stat[Bx],Ax  ; set return status
342 :          Pop     Es              ; restore registers
343 :          Pop     Ds
344 :          Pop     Bp
345 :          Pop     Di
346 :          Pop     Dx
347 :          Pop     Cx
348 :          Pop     Bx
349 :          Pop     Ax
350 :          Popf
351 :          Pop     Si
352 :          Ret
353 : ExitP   Endp
354 :
355 : Subttl  Driver Service Routines
356 : Page
357 :
358 : ;       Read data from device
359 :
360 : Read:
361 : ;        dbg     'R','d',' '
362 :          Mov     Cx,Es:Count[bx] ; get requested nbr
363 :          Mov     Di,Es:Xfer[bx]  ; get target pointer
364 :          Mov     Dx,Es:Xseg[bx]
365 :          Push    Bx              ; save for count fixup
366 :          Push    Es
367 :          Mov     Es,Dx
368 :          Test    InStat[si],BadInp Or LostDt
369 :          Je      No_lerr         ; no error so far...
370 :          Add     Sp,4            ; error, flush SP
```

Figure 6-1. Continued.   *(more)*

```
371 :           And     InStat[si],Not ( BadInp Or LostDt )
372 :           Mov     Ax,ErrRf        ; error, report it
373 :           Jmp     Exit
374 : No_lerr:
375 :           Call    Get_in          ; go for one
376 :           Or      Ah,Ah
377 :           Jnz     Got_all         ; none to get now
378 :           Stosb                   ; store it
379 :           Loop    No_lerr         ; go for more
380 : Got_all:
381 :           Pop     Es
382 :           Pop     Bx
383 :           Sub     Di,Es:Xfer[bx]  ; calc number stored
384 :           Mov     Es:Count[bx],Di ; return as count
385 :           Jmp     Zexit
386 :
387 : ;         Nondestructive read from device
388 :
389 : Ndread:
390 :           Mov     Di,ifirst[si]
391 :           Cmp     Di,iavail[si]
392 :           Jne     Ndget
393 :           Jmp     Bsyexit         ; buffer empty
394 : Ndget:
395 :           Push    Bx
396 :           Mov     Bx,ibuf[si]
397 :           Mov     Al,[bx+di]
398 :           Pop     Bx
399 :           Mov     Es:media[bx],al ; return char
400 :           Jmp     Zexit
401 :
402 : ;         Input status request
403 :
404 : Rxstat:
405 :           Mov     Di,ifirst[si]
406 :           Cmp     Di,iavail[si]
407 :           Jne     Rxful
408 :           Jmp     Bsyexit         ; buffer empty
409 : Rxful:
410 :           Jmp     Zexit           ; have data
411 :
412 : ;         Input flush request
413 :
414 : Inflush:
415 :           Mov     Ax,iavail[si]
416 :           Mov     Ifirst[si],ax
417 :           Jmp     Zexit
418 :
419 : ;         Output data to device
420 :
```

Figure 6-1. Continued.   *(more)*

```
421 : Write:
422 : ;          dbg     'W','r',' '
423 :            Mov     Cx,es:count[bx]
424 :            Mov     Di,es:xfer[bx]
425 :            Mov     Ax,es:xseg[bx]
426 :            Mov     Es,ax
427 : Wlup:
428 :            Mov     Al,es:[di]        ; get the byte
429 :            Inc     Di
430 : Wwait:
431 :            Call    Put_out           ; put away
432 :            Cmp     Ah,0
433 :            Jne     Wwait             ; wait for room!
434 :            Call    Start_output      ; get it going
435 :            Loop    Wlup
436 :
437 :            Jmp     Zexit
438 :
439 : ;          Output status request
440 :
441 : Txstat:
442 :            Mov     Ax,ofirst[si]
443 :            Dec     Ax
444 :            And     Ax,bufmsk
445 :            Cmp     Ax,oavail[si]
446 :            Jne     Txroom
447 :            Jmp     Bsyexit           ; buffer full
448 : Txroom:
449 :            Jmp     Zexit             ; room exists
450 :
451 : ;          IOCTL read request, return line parameters
452 :
453 : Ioctlin:
454 :            Mov     Cx,es:count[bx]
455 :            Mov     Di,es:xfer[bx]
456 :            Mov     Dx,es:xseg[bx]
457 :            Mov     Es,dx
458 :            Cmp     Cx,10
459 :            Je      Doiocin
460 :            Mov     Ax,errbsl
461 :            Jmp     Exit
462 : Doiocin:
463 :            Mov     Dx,port[si]       ; base port
464 :            Mov     Dl,Lctrl          ; line status
465 :            Mov     Cx,4              ; LCR, MCR, LSR, MSR
466 : Getport:
467 :            In      Al,dx
468 :            Stos    Byte Ptr [DI]
469 :            Inc     Dx
470 :            Loop    Getport
471 :
```

(more)

Figure 6-1. Continued.

```
472 :            Mov     Ax,InSpec[si]    ; spec in flags
473 :            Stos    Word Ptr [DI]
474 :            Mov     Ax,OutSpec[si]   ; out flags
475 :            Stos    Word Ptr [DI]
476 :            Mov     Ax,baud[si]      ; baud rate
477 :            Mov     Bx,di
478 :            Mov     Di,offset Asy_baudt+2
479 :            Mov     Cx,15
480 : Baudcin:
481 :            Cmp     [di],ax
482 :            Je      Yesinb
483 :            Add     Di,4
484 :            Loop    Baudcin
485 : Yesinb:
486 :            Mov     Ax,-2[di]
487 :            Mov     Di,bx
488 :            Stos    Word Ptr [DI]
489 :            Jmp     Zexit
490 :
491 : ;          Flush output buffer request
492 :
493 : Txflush:
494 :            Mov     Ax,oavail[si]
495 :            Mov     Ofirst[si],ax
496 :            Jmp     Zexit
497 :
498 : ;          IOCTL request: change line parameters for this driver
499 :
500 : Ioctlout:
501 :            Mov     Cx,es:count[bx]
502 :            Mov     Di,es:xfer[bx]
503 :            Mov     Dx,es:xseg[bx]
504 :            Mov     Es,dx
505 :            Cmp     Cx,10
506 :            Je      Doiocout
507 :            Mov     Ax,errbsl
508 :            Jmp     Exit
509 :
510 : Doiocout:
511 :            Mov     Dx,port[si]      ; base port
512 :            Mov     Dl,Lctrl         ; line ctrl
513 :            Mov     Al,es:[di]
514 :            Inc     Di
515 :            Or      Al,Dlab          ; set baud
516 :            Out     Dx,al
517 :            Clc
518 :            Jnc     $+2
519 :            Inc     Dx               ; mdm ctrl
520 :            Mov     Al,es:[di]
521 :            Or      Al,Usr2          ; Int Gate
522 :            Out     Dx,al
```

(mⁱ

Figure 6-1. Continued.

```
523 :           Add    Di,3            ; skip LSR,MSR
524 :           Mov    Ax,es:[di]
525 :           Add    Di,2
526 :           Mov    InSpec[si],ax
527 :           Mov    Ax,es:[di]
528 :           Add    Di,2
529 :           Mov    OutSpec[si],ax
530 :           Mov    Ax,es:[di]      ; set baud
531 :           Mov    Bx,di
532 :           Mov    Di,offset Asy_baudt
533 :           Mov    Cx,15
534 : Baudcout:
535 :           Cmp    [di],ax
536 :           Je     Yesoutb
537 :           Add    Di,4
538 :           Loop   Baudcout
539 :
540 :           Mov    Dl,Lctrl        ; line ctrl
541 :           In     Al,dx           ; get LCR data
542 :           And    Al,not Dlab     ; strip
543 :           Clc
544 :           Jnc    $+2
545 :           Out    Dx,al           ; put back
546 :           Mov    Ax,ErrUm        ; "unknown media"
547 :           Jmp    Exit
548 :
549 : Yesoutb:
550 :           Mov    Ax,2[di]        ; get divisor
551 :           Mov    Baud[si],ax     ; save to report later
552 :           Mov    Dx,port[si]     ; set divisor
553 :           Out    Dx,al
554 :           Clc
555 :           Jnc    $+2
556 :           Inc    Dx
557 :           Mov    Al,ah
558 :           Out    Dx,al
559 :           Clc
560 :           Jnc    $+2
561 :           Mov    Dl,Lctrl        ; line ctrl
562 :           In     Al,dx           ; get LCR data
563 :           And    Al,not Dlab     ; strip
564 :           Clc
565 :           Jnc    $+2
566 :           Out    Dx,al           ; put back
567 :           Jmp    Zexit
568 :
569 : Subttl  Ring Buffer Routines
570 : Page
571 :
572 : Put_out Proc   Near     ; puts AL into output ring buffer
573 :           Push   Cx
```

Figure 6-1. Continued.   *(more)*

```
574 :           Push    Di
575 :           Pushf
576 :           Cli
577 :           Mov     Cx,oavail[si]    ; put ptr
578 :           Mov     Di,cx
579 :           Inc     Cx               ; bump
580 :           And     Cx,bufmsk
581 :           Cmp     Cx,ofirst[si]    ; overflow?
582 :           Je      Poerr            ; yes, don't
583 :           Add     Di,obuf[si]      ; no
584 :           Mov     [di],al          ; put in buffer
585 :           Mov     Oavail[si],cx
586 : ;         dbg     'p','o',' '
587 :           Mov     Ah,0
588 :           Jmp     Short   Poret
589 : Poerr:
590 :       ,   Mov     Ah,-1
591 : Poret:
592 :           Popf
593 :           Pop     Di
594 :           Pop     Cx
595 :           Ret
596 : Put_out Endp
597 :
598 : Get_out Proc    Near     ; gets next character from output ring buffer
599 :           Push    Cx
600 :           Push    Di
601 :           Pushf
602 :           Cli
603 :           Mov     Di,ofirst[si]    ; get ptr
604 :           Cmp     Di,oavail[si]    ; put ptr
605 :           Jne     Ngoerr
606 :           Mov     Ah,-1            ; empty
607 :           Jmp     Short   Goret
608 : Ngoerr:
609 : ;         dbg     'g','o',' '
610 :           Mov     Cx,di
611 :           Add     Di,obuf[si]
612 :           Mov     Al,[di]          ; get char
613 :           Mov     Ah,0
614 :           Inc     Cx               ; bump ptr
615 :           And     Cx,bufmsk        ; wrap
616 :           Mov     Ofirst[si],cx
617 : Goret:
618 :           Popf
619 :           Pop     Di
620 :           Pop     Cx
621 :           Ret
622 : Get_out Endp
623 :
624 : Put_in  Proc    Near     ; puts the char from AL into input ring buffer
```

Figure 6-1. Continued.   *(more)*

```
625 :          Push    Cx
626 :          Push    Di
627 :          Pushf
628 :          Cli
629 :          Mov     Di,iavail[si]
630 :          Mov     Cx,di
631 :          Inc     Cx
632 :          And     Cx,bufmsk
633 :          Cmp     Cx,ifirst[si]
634 :          Jne     Npierr
635 :          Mov     Ah,-1
636 :          Jmp     Short   Piret
637 : Npierr:
638 :          Add     Di,ibuf[si]
639 :          Mov     [di],al
640 :          Mov     Iavail[si],cx
641 : ;        dbg     'p','i',' '
642 :          Mov     Ah,0
643 : Piret:
644 :          Popf
645 :          Pop     Di
646 :          Pop     Cx
647 :          Ret
648 : Put_in   Endp
649 :
650 : Get_in   Proc    Near    ; gets one from input ring buffer into AL
651 :          Push    Cx
652 :          Push    Di
653 :          Pushf
654 :          Cli
655 :          Mov     Di,ifirst[si]
656 :          Cmp     Di,iavail[si]
657 :          Je      Gierr
658 :          Mov     Cx,di
659 :          Add     Di,ibuf[si]
660 :          Mov     Al,[di]
661 :          Mov     Ah,0
662 : ;        dbg     'g','i',' '
663 :          Inc     Cx
664 :          And     Cx,bufmsk
665 :          Mov     Ifirst[si],cx
666 :          Jmp     Short   Giret
667 : Gierr:
668 :          Mov     Ah,-1
669 : Giret:
670 :          Popf
671 :          Pop     Di
672 :          Pop     Cx
673 :          Ret
674 : Get_in   Endp
675 :
```

Figure 6-1. Continued.   *(more)*

```
676 : Subttl  Interrupt Dispatcher Routine
677 : Page
678 :
679 : Asy1isr:
680 :        Sti
681 :        Push   Si
682 :        Lea    Si,asy_tab1
683 :        Jmp    Short   Int_serve
684 :
685 : Asy2isr:
686 :        Sti
687 :        Push   Si
688 :        Lea    Si,asy_tab2
689 :
690 : Int_serve:
691 :        Push   Ax              ; save all regs
692 :        Push   Bx
693 :        Push   Cx
694 :        Push   Dx
695 :        Push   Di
696 :        Push   Ds
697 :        Push   Cs              ; set DS = CS
698 :        Pop    Ds
699 : Int_exit:
700 : ;      dbg    'I','x',' '
701 :        Mov    Dx,Port[si]    ; base address
702 :        Mov    Dl,IntId       ; check Int ID
703 :        In     Al,Dx
704 :        Cmp    Al,00h         ; dispatch filter
705 :        Je     Int_modem
706 :        Jmp    Int_mo_no
707 : Int_modem:
708 : ;      dbg    'M','S',' '
709 :        Mov    Dl,Mstat
710 :        In     Al,dx          ; read MSR content
711 :        Test   Al,CDlvl       .; carrier present?
712 :        Jnz    Msdsr          ; yes, test for DSR
713 :        Test   OutSpec[si],OutCdf    ; no, is CD off line?
714 :        Jz     Msdsr
715 :        Or     InStat[si],OffLin
716 : Msdsr:
717 :        Test   Al,DSRlvl      ; DSR present?
718 :        Jnz    Dsron          ; yes, handle it
719 :        Test   OutSpec[si],OutDSR    ; no, is DSR throttle?
720 :        Jz     Dsroff
721 :        Or     OtStat[si],LinDSR     ; yes, throttle down
722 : Dsroff:
723 :        Test   OutSpec[si],OutDrf    ; is DSR off line?
724 :        Jz     Mscts
725 :        Or     InStat[si],OffLin     ; yes, set flag
726 :        Jmp    Short   Mscts
```

Figure 6-1. Continued.   (*more*)

```
727 :  Dsron:
728 :           Test    OtStat[si],LinDSR       ; throttled for DSR?
729 :           Jz      Mscts
730 :           Xor     OtStat[si],LinDSR       ; yes, clear it out
731 :           Call    Start_output
732 :  Mscts:
733 :           Test    Al,CTSlvl       ; CTS present?
734 :           Jnz     Ctson           ; yes, handle it
735 :           Test    OutSpec[si],OutCTS ·    ; no, is CTS throttle?
736 :           Jz      Int_exit2
737 :           Or      OtStat[si],LinCTS       ; yes, shut it down
738 :           Jmp     Short   Int_exit2
739 :  Ctson:
740 :           Test    OtStat[si],LinCTS       ; throttled for CTS?
741 :           Jz      Int_exit2
742 :           Xor     OtStat[si],LinCTS       ; yes, clear it out
743 :           Jmp     Short   Int_exit1
744 :  Int_mo_no:
745 :           Cmp     Al,02h
746 :           Jne     Int_tx_no
747 :  Int_txmit:
748 : ;         dbg     'T','x',' '
749 :  Int_exit1:
750 :           Call    Start_output    ; try to send another
751 :  Int_exit2:
752 :           Jmp     Int_exit
753 :  Int_tx_no:
754 :           Cmp     Al,04h
755 :           Jne     Int_rec_no
756 :  Int_receive:
757 : ;         dbg     'R','x',' '
758 :           Mov     Dx,port[si]
759 :           In      Al,dx           ; take char from 8250
760 :           Test    OutSpec[si],OutXon  ; is XON/XOFF enabled?
761 :           Jz      Stuff_in        ; no
762 :           Cmp     Al,'S' And 01FH ; yes, is this XOFF?
763 :           Jne     Isq             ; no, check for XON
764 :           Or      OtStat[si],LinXof ; yes, disable output
765 :           Jmp     Int_exit2       ; don't store this one
766 :  Isq:
767 :           Cmp     Al,'Q' And 01FH ; is this XON?
768 :           Jne     Stuff_in        ; no, save it
769 :           Test    OtStat[si],LinXof ; yes, waiting?
770 :           Jz      Int_exit2       ; no, ignore it
771 :           Xor     OtStat[si],LinXof ; yes, clear the XOFF bit
772 :           Jmp     Int_exit1       ; and try to resume xmit
773 :  Int_rec_no:
774 :           Cmp     Al,06h
775 :           Jne     Int_done
776 :  Int_rxstat:
777 : ;         dbg     'E','R',' '
```

Figure 6-1. Continued.   *(more)*

```
778 :          Mov     Dl,Lstat
779 :          In      Al,dx
780 :          Test    InSpec[si],InEpc ; return them as codes?
781 :          Jz      Nocode           ; no, just set error alarm
782 :          And     Al,AnyErr        ; yes, mask off all but error bits
783 :          Or      Al,080h
784 : Stuff_in:
785 :          Call    Put_in           ; put input char in buffer
786 :          Cmp     Ah,0        •    ; did it fit?
787 :          Je      Int_exit3        ; yes, all OK
788 :          Or      InStat[si],LostDt  ; no, set DataLost bit
789 : Int_exit3:
790 :          Jmp     Int_exit
791 : Nocode:
792 :          Or      InStat[si],BadInp
793 :          Jmp     Int_exit3
794 : Int_done:
795 :          Clc
796 :          Jnc     $+2
797 :          Mov     Al,EOI           ; all done now
798 :          Out     PIC_b,Al
799 :          Pop     Ds               ; restore regs
800 :          Pop     Di
801 :          Pop     Dx
802 :          Pop     Cx
803 :          Pop     Bx
804 :          Pop     Ax
805 :          Pop     Si
806 :          Iret
807 :
808 : Start_output    Proc    Near
809 :          Test    OtStat[si],LinIdl ; Blocked?
810 :          Jnz     Dont_start       ; yes, no output
811 :          Mov     Dx,port[si]      ; no, check UART
812 :          Mov     Dl,Lstat
813 :          In      Al,Dx
814 :          Test    Al,xhre          ; empty?
815 :          Jz      Dont_start       ; no
816 :          Call    Get_out          ; yes, anything waiting?
817 :          Or      Ah,Ah
818 :          Jnz     Dont_start       ; no
819 :          Mov     Dl,RxBuf         ; yes, send it out
820 :          Out     Dx,al
821 : ;        dbg     's','o',' '
822 : Dont_start:
823 :          ret
824 : Start_output    Endp
825 :
826 : Subttl  Initialization Request Routine
827 : Page
828 :
```

Figure 6-1. Continued.   *(more)*

```
829 : Init:   Lea    Di,$              ; release rest...
830 :         Mov    Es:Xfer[bx],Di
831 :         Mov    Es:Xseg[bx],Cs
832 :
833 :         Mov    Dx,Port[si]       ; base port
834 :         Mov    Dl,Lctrl
835 :         Mov    Al,Dlab           ; enable divisor
836 :         Out    Dx,Al
837 :         Clc
838 :         Jnc    $+2
839 :         Mov    Dl,RxBuf
840 :         Mov    Ax,Baud[si]       ; set baud
841 :         Out    Dx,Al
842 :         Clc
843 :         Jnc    $+2
844 :         Inc    Dx
845 :         Mov    Al,Ah
846 :         Out    Dx,Al
847 :         Clc
848 :         Jnc    $+2
849 :
850 :         Mov    Dl,Lctrl          ; set LCR
851 :         Mov    Al,OtStat[si]     ; from table
852 :         Out    Dx,Al
853 :         Mov    OtStat[si],0      ; clear status
854 :         Clc
855 :         Jnc    $+2
856 :         Mov    Dl,IntEn          ; IER
857 :         Mov    Al,AllInt         ; enable ints in 8250
858 :         Out    Dx,Al
859 :         Clc
860 :         Jnc    $+2
861 :         Mov    Dl,Mctrl          ; set MCR
862 :         Mov    Al,InStat[si]     ; from table
863 :         Out    Dx,Al
864 :         Mov    InStat[si],0      ; clear status
865 :
866 : ClRgs:  Mov    Dl,Lstat          ; clear LSR
867 :         In     Al,Dx
868 :         Mov    Dl,RxBuf          ; clear RX reg
869 :         In     Al,Dx
870 :         Mov    Dl,Mstat          ; clear MSR
871 :         In     Al,Dx
872 :         Mov    Dl,IntId          ; IID reg
873 :         In     Al,Dx
874 :         In     Al,Dx
875 :         Test   Al,1              ; int pending?
876 :         Jz     ClRgs             ; yes, repeat
877 :
878 :         Cli
879 :         Xor    Ax,Ax             ; set int vec
```

Figure 6-1. Continued.

(more)

```
880 :          Mov     Es,Ax
881 :          Mov     Di,Vect[si]
882 :          Mov     Ax,IsrAdr[si]    ; from table
883 :          Stosw
884 :          Mov     Es:[di],cs
885 :
886 :          In      Al,PIC_e         ; get 8259
887 :          And     Al,0E7h          ; com1/2 mask
888 :          Clc
889 :          Jnb     $+2
890 :          Out     PIC_e,Al
891 :          Sti
892 :
893 :          Mov     Al,EOI           ; now send EOI just in case
894 :          Out     PIC_b,Al
895 :
896 :  ;       dbg     'D','I',' '      ; driver installed
897 :          Jmp     Zexit
898 :
899 : Driver   Ends
900 :          End
```

Figure 6-1. Continued.

The first part of the driver source code (after the necessary MASM housekeeping details
in lines 1 through 8) is a commented-out macro definition (lines 10 through 32). This
macro is used only during debugging and is part of a debugging technique that requires
no sophisticated hardware and no more complex debugging program than the venerable
DEBUG.COM. (Debugging techniques are discussed after the presentation of the driver
program itself.)

Definitions

The actual driver source program consists of three sets of EQU definitions (lines 34
through 194), followed by the modular code and data areas (lines 197 through 900). The
first set of definitions (lines 34 through 82) gives symbolic names to the permissible values
for MS-DOS device-driver control bits and the device-driver structures.

The second set of definitions (lines 84 through 145) assigns names to the ports and bit
values that are associated with the IBM hardware — both the 8259 PIC and the 8250 UART.
The third set of definitions (lines 147 through 194) assigns names to the control values and
structures associated with this driver.

The definition method used here is recommended for all drivers. To move this driver from
the IBM architecture to some other hardware, the major change required to the program
would be reassignment of the port addresses and bit values in lines 84 through 145.

The control values and structures for this specific driver (defined in the third EQU set)
provide the means by which the separate support program can modify the actions of each
of the two logical drivers. They also permit the driver to return status information to both

the support program and the using program as necessary. Only a few features are implemented, but adequate space for expansion is provided. The addition of a few more definitions in this area and one or two extra procedures in the code section would do all that is necessary to extend the driver's capabilities to such features as automatic expansion of tab characters, case conversion, and so forth, should they be desired.

Headers and structure tables

The driver code itself starts with a linked pair of device-driver header blocks, one for *ASY1* (lines 201 through 207) and the other for *ASY2* (lines 208 through 213). Following the headers, in lines 215 through 236, are a commented-out space reservation used by the debugging procedure (line 215), the pointer to the command packet (line 219), and the baud-rate conversion table (lines 221 through 236).

The conversion table is followed by structure tables containing all data unique to *ASY1* (lines 239 through 242) and *ASY2* (lines 244 through 247). After the structure tables, buffer areas are reserved in lines 249 through 254. One input buffer and one output buffer are reserved for each port. All buffers are the same size; for simplicity, buffer size is given a name (at line 249) so that it can be changed by editing a single line of the program.

The size is arbitrary in this case, but if file transfers are anticipated, the buffer should be able to hold at least 2 seconds' worth of data (240 bytes at 1200 bps) to avoid data loss during writes to disk. Whatever size is chosen should be a power of 2 for simple pointer arithmetic and, if video display is intended, should not be less than 8 bytes, to prevent losing characters when the screen scrolls.

If additional ports are desired, more headers can be added after line 213; corresponding structure tables for each driver, plus matching pairs of buffers, would also be necessary. The final part of this area is the dispatch table (lines 256 through 284), which lists offsets of all request routines in the driver; its use is discussed below.

Strategy and Request routines

With all data taken care of, the program code begins at the Strategy routine (lines 289 through 296), which is used by both ports. This code saves the command packet address passed to it by MS-DOS for use by the Request routine and returns to MS-DOS.

The Request routines (lines 298 through 567) are also shared by both ports, but the two drivers are distinguished by the address placed into the SI register. This address points to the structure table that is unique to each port and contains such data as the port's base address, the associated hardware interrupt vector, the interrupt service routine offset within the driver's segment, the base offsets of the input and output buffers for that port, two pointers for each of the buffers, and the input and output status conditions (including baud rate) for the port. The only difference between one port's driver and the other's is the data pointed to by SI; all Request routine code is shared by both ports.

Each driver's Request routine has a unique entry point (at line 298 for *ASY1* and at line 303 for *ASY2*) that saves the original content of the SI register and then loads it with the address of the structure table for that driver. The routines then join as a common stream at line 307 (*Gen_request*).

This common code preserves all other registers used (lines 309 through 318), sets DS equal to CS (lines 319 and 320), retrieves the command-packet pointer saved by the Strategy routine (line 321), uses the pointer to get the command code (line 323), uses the code to calculate an offset into a table of addresses (lines 324 through 326), and performs an indexed jump (lines 322 and 327) by way of the dispatch table (lines 256 through 284) to the routine that executes the requested command (at line 336, 360, 389, 404, 414, 421, 441, 453, 500, or 829).

Although the device-driver specifications for MS-DOS version 3.2 list command request codes ranging from 0 to 24, not all are used. Earlier versions of MS-DOS permitted only 0 to 12 (versions 2.x) or 0 to 16 (versions 3.0 and 3.1) codes. In this driver, all 24 codes are accounted for; those not implemented in this driver return a DONE and NO ERROR status to the caller. Because the Request routine is called only by MS-DOS itself, there is no check for invalid codes. Actually, because the header attribute bits are *not* set to specify that codes 13 through 24 are valid, the 24 bytes occupied by their table entries (lines 273 through 284) could be saved by omitting the entries. They are included only to show how nonexistent commands can be accommodated.

Immediately following the dispatch indexed jump, at lines 329 through 353 within the same PROC declaration, is the common code used by all Request routines to store status information in the command packet, restore the registers, and return to the caller. The alternative entry points for BUSY status (line 332), NO ERROR status (line 338), or an error code (in the AX register at entry to *Exit*, line 339) not only save several bytes of redundant code but also improve readability of the code by providing unique single labels for BUSY, NO ERROR, and ERROR return conditions.

All of the Request routines, except for the *Init* code at line 829, immediately follow the dispatching shell in lines 358 through 568. Each is simplified to perform just one task, such as read data in or write data out. The *Read* routine (lines 360 through 385) is typical: First, the requested byte count and user's buffer address are obtained from the command packet. Next, the pointer to the command packet is saved with a PUSH instruction, so that the ES and BX registers can be used for a pointer to the port's input buffer.

Before the *Get_in* routine that actually accesses the input buffer is called, the input status byte is checked (line 368). If an error condition is flagged, lines 370 through 373 clear the status flag, flush the saved pointers from the stack, and jump to the error-return exit from the driver. If no error exists, line 375 calls *Get_in* to access the input buffer and lines 376 and 377 determine whether a byte was obtained. If a byte is found, it is stored in the user's buffer by line 378, and line 379 loops back to get another byte until the requested count has been obtained or until no more bytes are available. In practice, the count is an upper limit and the loop is normally broken when data runs out.

No matter how it happens, control eventually reaches the *Got_all* routine and lines 381 and 382, where the saved pointers to the command packet are restored from the stack. Lines 383 and 384 adjust the count value in the packet to reflect the actual number of bytes obtained. Finally, line 385 jumps to the normal, no-error exit from the driver.

Buffering

Both buffers for each driver are of the type known as circular, or ring, buffers. Effectively, such a buffer is endless; it is accessed via pointers, and when a pointer increments past the end of the buffer, the pointer returns to the buffer's beginning. Two pointers are used here for each buffer, one to put data into it and one to get data out. The *get* pointer always points to the next byte to be read; the *put* pointer points to where the next byte will be written, just past the last byte written to the buffer.

If both pointers point to the same byte, the buffer is empty; the next byte to be read has not yet been written. The full-buffer condition is more difficult to test for: The *put* pointer is incremented and compared with the *get* pointer; if they are equal, doing a write would force a false buffer-empty condition, so the buffer must be full.

All buffer manipulation is done via four procedures (lines 569 through 674). *Put_out* (lines 572 through 596) writes a byte to the driver's output buffer or returns a buffer-full indication by setting AH to 0FFH. *Get_out* (lines 598 through 622) gets a byte from the output buffer or returns 0FFH in AH to indicate that no byte is available. *Put_in* (lines 624 through 648) and *Get_in* (lines 650 through 674) do exactly the same as *Put_out* and *Get_out*, but for the input buffer. These procedures are used both by the Request routines and by the hardware interrupt service routine (ISR).

Interrupt service routines

The most complex part of this driver is the ISR (lines 676 through 806), which decides which of the four possible services for a port is to be performed and where. Like the Request routines, the ISR provides unique entry points for each port (line 679 for *ASY1* and line 685 for *ASY2*); these entry points first preserve the SI register and then load it with the address of the port's structure table. With SI indicating where the actions are to be performed, the two entries then merge at line 690 into common code that first preserves all registers to be used by the ISR (lines 690 through 698) and then tests for each of the four possible types of service and performs each requested action.

Much of the complexity of the ISR is in the decoding of modem-status conditions. Because the resulting information is not used by this driver (although it could be used to prevent attempts to transmit while off line), these ISR options can be removed so that only the Transmit and Receive interrupts are serviced. To do this, *AllInt* (at line 145) should be changed from the OR of all four bits to include only the transmit and receive bits (03H, or 00000011B).

The transmit and receive portions of the ISR incorporate XON/XOFF flow control (for transmitted data only) by default. This control is done at the ISR level, rather than in the using program, to minimize the time required to respond to an incoming XOFF signal. Presence of the flow-control decisions adds complexity to what would otherwise be extremely simple actions.

Flow control is enabled or disabled by setting the *OutSpec* word in the structure table with the Driver Status utility (presented later) via the IOCTL function (Interrupt 21H Function 44H). When flow control is enabled, any XOFF character (11H) that is received halts all outgoing data until XON (13H) is received. No XOFF or XON is retained in the input

buffer to be sent on to any program, although all patterns other than XOFF and XON *are* passed through by the driver. When flow control is disabled, the driver passes all patterns in both directions. For binary file transfer, flow control must be disabled.

The transmit action is simple: The code merely calls the *Start_output* procedure at line 750. *Start_output* is described in detail below.

The receive action is almost as simple as transmit, except for the flow-control testing. First, the ISR takes the received byte from the UART (lines 758 and 759) to avoid any chance of an overrun error. The ISR then tests the input specifier (at line 760) to determine whether flow control is in effect. If it is not, processing jumps directly to line 784 to store the received byte in the input buffer with *Put_in* (line 785).

If flow control is active, however, the received byte is compared with the XOFF character (lines 762 through 765). If the byte matches, output is disabled and the byte is ignored. If the byte is not XOFF, it is compared with XON (lines 766 through 768). If it is not XON either, control jumps to line 784. If the byte is XON, output is re-enabled if it was disabled. Regardless, the XON byte itself is ignored.

When control reaches *Stuff_in* at line 784, *Put_in* is called to store the received byte in the input buffer. If there is no room for it, a lost-databit is set in the input status flags (line 788); otherwise, the receive routine is finished.

If the interrupt was a line-status action, the LSR is read (lines 776 through 779). If the input specifier so directs, the content is converted to an IBM PC extended graphics character by setting bit 7 to 1 and the character is stored in the input buffer as if it were a received byte. Otherwise, the Line Status interrupt merely sets the generic *BadInp* error bit in the input status flags, which can be read with the IOCTL Read function of the driver.

When all ISR action is complete, lines 794 through 806 restore machine conditions to those existing at the time of the interrupt and return to the interrupted procedure.

The *Start_output* routine

Start_output (lines 808 through 824) is a routine that, like the four buffer procedures, is used by both the Request routines and the ISR. Its purpose is to initiate transmission of a byte, provided that output is not blocked by flow control, the UART Transmit Holding Register is empty, and a byte to be transmitted exists in the output ring buffer. This routine uses the *Get_out* buffer routine to access the buffer and determine whether a byte is available. If all conditions are met, the byte is sent to the UART holding register by lines 819 and 820.

The Initialization Request routine

The Initialization Request routine (lines 829 through 897) is critical to successful operation of the driver. This routine is placed last in the package so that it can be discarded as soon as it has served its purpose by installing the driver. It is essential to clear each register of the 8250 by reading its contents before enabling the interrupts and to loop through this

action until the 8250 finally shows no requests pending. The strange *Clc jnc $+2* sequence that appears repeatedly in this routine is a time delay required by high-speed machines (6 MHz and up) so that the 8250 has time to settle before another access is attempted; the delay does no harm on slower machines.

Using COMDVR

The first step in using this device driver is assembling it with the Microsoft Macro Assembler (MASM). Next, use the Microsoft Object Linker (LINK) to create a .EXE file. Convert the .EXE file into a binary image file with the EXE2BIN utility. Finally, include the line *DEVICE = COMDVR.SYS* in the CONFIG.SYS file so that COMDVR will be installed when the system is restarted.

Note: The number and colon at the beginning of each line in the program listings in this article are for reference only and should not be included in the source file.

Figure 6-2 shows the sequence of actions required, assuming that EDLIN is used for modifying (or creating) the CONFIG.SYS file and that all commands are issued from the root directory of the boot drive.

Creating the driver:

```
C>MASM COMDVR;   <Enter>
C>LINK COMDVR;   <Enter>
C>EXE2BIN COMDVR.EXE COMDVR.SYS   <Enter>
```

Modifying CONFIG.SYS (^Z = press Ctrl-Z):

```
C>EDLIN CONFIG.SYS   <Enter>
*#I   <Enter>
*DEVICE=COMDVR.SYS   <Enter>
*^Z   <Enter>
*E   <Enter>
```

Figure 6-2. Assembling, linking, and installing COMDVR.

Because the devices installed by COMDVR do not use the standard MS-DOS device names, no conflict occurs with any program that uses conventional port references. Such a program will not use the driver, and no problems should result if the program is well behaved and restores all interrupt vectors before returning to MS-DOS.

Device-driver debugging techniques

The debugging of device drivers, like debugging for any part of MS-DOS itself, is more difficult than normal program checking because the debugging program, DEBUG.COM or DEBUG.EXE, itself uses MS-DOS functions to display output. When these functions are being checked, their use by DEBUG destroys the data being examined. And because MS-DOS always saves its return address in the same location, any call to a function from inside the operating system usually causes a system lockup that can be cured only by shutting the system down and powering up again.

One way to overcome this difficulty is to purchase costly debugging tools. An easier way is to bypass the problem: Instead of using MS-DOS functions to track program operation, write data directly to video RAM, as in the macro *DBG* (lines 10 through 32 of COMDVR.ASM).

This macro is invoked with a three-character parameter string at each point in the program a progress report is desired. Each invocation has its own unique three-character string so that the sequence of actions can be read from the screen. When invoked, *DBG* expands into code that saves all registers and then writes the three-character string to video RAM. Only the top 10 lines of the screen (800 characters, or 1600 bytes) are used: The macro uses a single far pointer to the area and treats the video RAM like a ring buffer.

The pointer, *Dbgptr* (line 215), is set up for use with the monochrome adapter and points to location B000:0000H; to use a CGA or EGA (in CGA mode), the location should be changed to B800:0000H.

Most of the frequently used Request routines, such as *Read* and *Write*, have calls to *DBG* as their first lines (for example, lines 361 and 422). As shown, these calls are commented out, but for debugging, the source file should be edited so that all the calls and the macro itself are enabled.

With *DBG* active, the top 10 lines of the display are overwritten with a continual sequence of reports, such as *RR Tx*, put directly into video RAM. Because MS-DOS functions are not used, no interference with the driver itself can occur.

Although this technique prevents normal use of the system during debugging, it greatly simplifies the problem of knowing what is happening in time-critical areas, such as hardware interrupt service. In addition, all invocations of *DBG* in the critical areas are in conditional code that is executed only when the driver is working as it should.

Failure to display the *pi* message, for instance, indicates that the received-data hardware interrupt is not being serviced, and absence of *go* after an *Ix* report shows that data is not being sent out as it should.

Of course, once debugging is complete, the calls to *DBG* should be deleted or commented out. Such calls are usually edited out of the source code before release. In this case, they remain to demonstrate the technique and, most particularly, to show placement of the calls to provide maximum information with minimal clutter on the screen.

A simple modem engine

The second part of this package is the modem engine itself (ENGINE.ASM), shown in the listing in Figure 6-3. The main loop of this program consists of only a dozen lines of code (lines 9 through 20). Of these, five (lines 9 through 13) are devoted to establishing initial contact between the program and the serial-port driver and two (lines 19 and 20) are for returning to command level at the program's end.

Thus, only five lines of code (lines 14 through 18) actually carry out the bulk of the program as far as the main loop is concerned. Four of these lines are calls to subroutines that

get and put data from and to the console and the serial port; the fifth is the JMP that closes the loop. This structure underscores the fact that a basic modem engine is simply a data-transfer loop.

```
 1 :             TITLE    engine
 2 :
 3 : CODE        SEGMENT PUBLIC 'CODE'
 4 :
 5 :             ASSUME   CS:CODE,DS:CODE,ES:CODE,SS:CODE
 6 :
 7 :             ORG      0100h
 8 :
 9 : START:  mov      dx,offset devnm ; open named device (ASY1)
10 :         mov      ax,3d02h
11 :         int      21h
12 :         mov      handle,ax       ; save the handle
13 :         jc       quit
14 : alltim: call     getmdm          ; main engine loop
15 :         call     putcrt
16 :         call     getkbd
17 :         call     putmdm
18 :         jmp      alltim
19 : quit:   mov      ah,4ch          ; come here to quit
20 :         int      21h
21 :
22 : getmdm  proc                     ; get input from modem
23 :         mov      cx,256
24 :         mov      bx,handle
25 :         mov      dx,offset mbufr
26 :         mov      ax,3F00h
27 :         int      21h
28 :         jc       quit
29 :         mov      mdlen,ax
30 :         ret
31 : getmdm  endp
32 :
33 : getkbd  proc                     ; get input from keyboard
34 :         mov      kblen,0         ; first zero the count
35 :         mov      ah,11           ; key pressed?
36 :         int      21h
37 :         inc      al
38 :         jnz      nogk            ; no
39 :         mov      ah,7            ; yes, get it
40 :         int      21h
41 :         cmp      al,3            ; was it Ctrl-C?
42 :         je       quit            ; yes, get out
43 :         mov      kbufr,al        ; no, save it
44 :         inc      kblen
45 :         cmp      al,13           ; was it Enter?
46 :         jne      nogk            ; no
```

Figure 6-3. ENGINE.ASM.   *(more)*

```
47 :            mov     byte ptr kbufr+1,10 ; yes, add LF
48 :            inc     kblen
49 : nogk:      ret
50 : getkbd     endp
51 :
52 : putmdm     proc                    ; put output to modem
53 :            mov     cx,kblen
54 :            jcxz    nopm
55 :            mov     bx,handle
56 :            mov     dx,offset kbufr
57 :            mov     ax,4000h
58 :            int     21h
59 :            jc      quit
60 : nopm:      ret
61 : putmdm     endp
62 :
63 : putcrt     proc                    ; put output to CRT
64 :            mov     cx,mdlen
65 :            jcxz    nopc
66 :            mov     bx,1
67 :            mov     dx,offset mbufr
68 :            mov     ah,40h
69 :            int     21h
70 :            jc      quit
71 : nopc:      ret
72 : putcrt     endp
73 :
74 : devnm      db      'ASY1',0        ; miscellaneous data and buffers
75 : handle     dw      0
76 : kblen      dw      0
77 : mdlen      dw      0
78 : mbufr      db      256 dup (0)
79 : kbufr      db      80 dup (0)
80 :
81 : CODE       ENDS
82 :            END     START
```

Figure 6-3. Continued.

Because the details of timing and data conversion are handled by the driver code, each of the four subroutines is — to show just how simple the whole process is — essentially a buffered interface to the MS-DOS Read File or Device or Write File or Device routine.

For example, the *getmdm* procedure (lines 22 through 31) asks MS-DOS to read a maximum of 256 bytes from the serial device and then stores the number actually read in a word named *mdlen*. The driver returns immediately, without waiting for data, so the normal number of bytes returned is either 0 or 1. If screen scrolling causes the loop to be delayed, the count might be higher, but it should never exceed about a dozen characters.

When called, the *putcrt* procedure (lines 63 through 72) checks the value in *mdlen*. If the value is zero, *putcrt* does nothing; otherwise, it asks MS-DOS to write that number of bytes from *mbufr* (where *getmdm* put them) to the display, and then it returns.

Similarly, *getkbd* gets keystrokes from the keyboard, stores them in *kbufr*, and posts a count in *kblen*; *putmdm* checks *kblen* and, if the count is not zero, sends the required number of bytes from *kbufr* to the serial device.

Note that *getkbd* does not use the Read File or Device function, because that would wait for a keystroke and the loop must never wait for reception. Instead, it uses the MS-DOS functions that test keyboard status (0BH) and read a key without echo (07H). In addition, special treatment is given to the Enter key (lines 45 through 48): A linefeed is inserted in *kbufr* immediately behind Enter and *kblen* is set to 2.

A Ctrl-C keystroke ends program operation; it is detected in *getkbd* (line 41) and causes immediate transfer to the *quit* label (line 19) at the end of the main loop. Because ENGINE uses only permanently resident routines, there is no need for any uninstallation before returning to the MS-DOS command prompt.

ENGINE.ASM is written to be used as a .COM file. Assemble and link it the same as COMDVR.SYS (Figure 6-2) but use the extension COM instead of SYS; no change to CONFIG.SYS is needed.

The driver-status utility: CDVUTL.C

The driver-status utility program CDVUTL.C, presented in Figure 6-4, permits either of the two drivers (*ASY1 and ASY2*) to be reconfigured after being installed, to suit different needs. After one of the drivers has been specified (port 1 or port 2), the baud rate, word length, parity, and number of stop bits can be changed; each change is made independently, with no effect on any of the other characteristics. Additionally, flow control can be switched between two types of hardware handshaking—the software XON/XOFF control or disabled—and error reporting can be switched between character-oriented and message-oriented operation.

```
 1 : /* cdvutl.c - COMDVR Utility
 2 : *      Jim Kyle - 1987
 3 : *      for use with COMDVR.SYS Device Driver.
 4 : */
 5 :
 6 : #include <stdio.h>              /* i/o definitions     */
 7 : #include <conio.h>              /* special console i/o */
 8 : #include <stdlib.h>             /* misc definitions    */
 9 : #include <dos.h>                /* defines intdos()    */
10 :
11 : /*      the following define the driver status bits    */
12 :
13 : #define HWINT 0x0800            /* MCR, first word, HW Ints gated */
14 : #define o_DTR 0x0200            /* MCR, first word, output DTR    */
15 : #define o_RTS 0x0100            /* MCR, first word, output RTS    */
16 :
17 : #define m_PG  0x0010            /* LCR, first word, parity ON     */
18 : #define m_PE  0x0008            /* LCR, first word, parity EVEN   */
```

Figure 6-4. CDVUTL.C   (more)

ZTE (USA) 1007, Page 219

```
19 : #define m_XS   0x0004         /* LCR, first word, 2 stop bits   */
20 : #define m_WL   0x0003         /* LCR, first word, wordlen mask  */
21 :
22 : #define i_CD   0x8000         /* MSR, 2nd word, Carrier Detect  */
23 : #define i_RI   0x4000         /* MSR, 2nd word, Ring Indicator  */
24 : #define i_DSR  0x2000         /* MSR, 2nd word, Data Set Ready  */
25 : #define i_CTS  0x1000         /* MSR, 2nd word, Clear to Send   */
26 :
27 : #define l_SRE  0x0040         /* LSR, 2nd word, Xmtr SR Empty   */
28 : #define l_HRE  0x0020         /* LSR, 2nd word, Xmtr HR Empty   */
29 : #define l_BRK  0x0010         /* LSR, 2nd word, Break Received  */
30 : #define l_ER1  0x0008         /* LSR, 2nd word, FrmErr          */
31 : #define l_ER2  0x0004         /* LSR, 2nd word, ParErr          */
32 : #define l_ER3  0x0002         /* LSR, 2nd word, OveRun          */
33 : #define l_RRF  0x0001         /* LSR, 2nd word, Rcvr DR Full    */
34 :
35 : /*           now define CLS string for ANSI.SYS          */
36 : #define CLS     "\033[2J"
37 :
38 : FILE * dvp;
39 : union REGS rvs;
40 : int iobf [ 5 ];
41 :
42 : main ()
43 : { cputs ( "\nCDVUTL - COMDVR Utility Version 1.0 - 1987\n" );
44 :   disp ();                          /* do dispatch loop          */
45 : }
46 :
47 : disp ()                             /* dispatcher; infinite loop */
48 : { int c,
49 :     u;
50 :   u = 1;
51 :   while ( 1 )
52 :     { cputs ( "\r\n\tCommand (? for help): " );
53 :       switch ( tolower ( c = getche ()))   /* dispatch           */
54 :         {
55 :         case '1' :                   /* select port 1            */
56 :           fclose ( dvp );
57 :           dvp = fopen ( "ASY1", "rb+" );
58 :             u = 1;
59 :           break;
60 :
61 :         case '2' :                   /* select port 2            */
62 :           fclose ( dvp );
63 :           dvp = fopen ( "ASY2", "rb+" );
64 :           u = 2;
65 :           break;
66 :
67 :         case 'b' :                   /* set baud rate            */
68 :           if ( iobf [ 4 ] == 300 )
69 :             iobf [ 4 ] = 1200;
```

Figure 6-4. Continued.   *(more)*

```
 70 :           else
 71 :             if ( iobf [ 4 ] == 1200 )
 72 :               iobf [ 4 ] = 2400;
 73 :           else
 74 :             if ( iobf [ 4 ] == 2400 )
 75 :               iobf [ 4 ] = 9600;
 76 :           else
 77 :             iobf [ 4 ] = 300;
 78 :           iocwr ();
 79 :           break;
 80 :
 81 :         case 'e' :                    /* set parity even          */
 82 :           iobf [ 0 ] |= ( m_PG + m_PE );
 83 :           iocwr ();
 84 :           break;
 85 :
 86 :         case 'f' :                    /* toggle flow control      */
 87 :           if ( iobf [ 3 ] == 1 )
 88 :             iobf [ 3 ] = 2;
 89 :           else
 90 :             if ( iobf [ 3 ] == 2 )
 91 :               iobf [ 3 ] = 4;
 92 :           else
 93 :             if ( iobf [ 3 ] == 4 )
 94 :               iobf [ 3 ] = 0;
 95 :           else
 96 :             iobf [ 3 ] = 1;
 97 :           iocwr ();
 98 :           break;
 99 :
100 :         case 'i' :                    /* initialize MCR/LCR to 8N1 : */
101 :           iobf [ 0 ] = ( HWINT + o_DTR + o_RTS + m_WL );
102 :           iocwr ();
103 :           break;
104 :
105 :         case '?' :              /* this help list            */
106 :           cputs ( CLS );        /* clear the display         */
107 :           center (."COMMAND LIST \n" );
108 : center ( "1 = select port 1         L = toggle word LENGTH  " );
109 : center ( "2 = select port 2         N = set parity to NONE  " );
110 : center ( "B = set BAUD rate         O = set parity to ODD   " );
111 : center ( "E = set parity to EVEN    R = toggle error REPORTS" );
112 : center ( "F = toggle FLOW control   S = toggle STOP bits    " );
113 : center ( "I = INITIALIZE ints, etc. Q = QUIT                " );
114 :           continue;
115 :
116 :         case 'l' :                    /* toggle word length       */
117 :           iobf [ 0 ] ^= 1;
118 :           iocwr ();
119 :           break;
120 :
```

Figure 6-4. Continued.   *(more)*

```
121 :          case 'n' :                    /* set parity off        */
122 :            iobf [ 0 ] &=~ ( m_PG + m_PE );
123 :            iocwr ();
124 :            break;
125 :
126 :          case 'o' :                    /* set parity odd        */
127 :            iobf [ 0 ] |= m_PG;
128 :            iobf [ 0 ] &=~ m_PE;
129 :            iocwr ();
130 :            break;
131 :
132 :          case 'r' :                    /* toggle error reports  */
133 :            iobf [ 2 ] ^= 1;
134 :            iocwr ();
135 :            break;
136 :
137 :          case 's' :                    /* toggle stop bits      */
138 :            iobf [ 0 ] ^= m_XS;
139 :            iocwr ();
140 :            break;
141 :
142 :          case 'q' :
143 :            fclose ( dvp );
144 :            exit ( 0 );                 /* break the loop, get out */
145 :          }
146 :        cputs ( CLS );                  /* clear the display     */
147 :        center ( "CURRENT COMDVR STATUS" );
148 :        report ( u, dvp );              /* report current status */
149 :      }
150 : }
151 :
152 : center ( s ) char * s;                 /* centers a string on CRT */
153 : { int i ;
154 :   for ( i = 80 - strlen ( s ); i > 0; i -= 2 )
155 :     putch ( ' ' );
156 :   cputs ( s );
157 :   cputs ( "\r\n" );
158 : }
159 :
160 : iocwr ()                               /* IOCTL Write to COMDVR */
161 : { rvs . x . ax = 0x4403;
162 :   rvs . x . bx = fileno ( dvp );
163 :   rvs . x . cx = 10;
164 :   rvs . x . dx = ( int ) iobf;
165 :   intdos ( & rvs, & rvs );
166 : }
167 :
168 : char * onoff ( x ) int x ;
169 : { return ( x ? " ON" : " OFF" );
170 : }
171 :
```

Figure 6-4. Continued.   *(more)*

```
172 : report ( unit ) int unit ;
173 : { char temp [ 80 ];
174 :    rvs . x . ax = 0x4402;
175 :    rvs . x . bx = fileno ( dvp );
176 :    rvs . x . cx = 10;
177 :    rvs . x . dx = ( int ) iobf;
178 :    intdos ( & rvs, & rvs );          /* use IOCTL Read to get data */
179 :    sprintf ( temp, "\nDevice ASY%d\t%d BPS, %d-c-%c\r\n\n",
180 :             unit, iobf [ 4 ],         /* baud rate            */
181 :             5 + ( iobf [ 0 ] & m_WL ),  /* word length        */
182 :             ( iobf [ 0 ] & m_PG ?
183 :                ( iobf [ 0 ] & m_PE ? 'E' : 'O' ) : 'N' ),
184 :             ( iobf [ 0 ] & m_XS ? '2' : '1' )); /* stop bits      */
185 :    cputs ( temp );
186 :
187 :    cputs ( "Hardware Interrupts are" );
188 :    cputs ( onoff ( iobf [ 0 ] & HWINT ));
189 :    cputs ( ", Data Terminal Rdy" );
190 :    cputs ( onoff ( iobf [ 0 ] & o_DTR ));
191 :    cputs ( ", Rqst To Send" );
192 :    cputs ( onoff ( iobf [ 0 ] & o_RTS ));
193 :    cputs ( ".\r\n" );
194 :
195 :    cputs ( "Carrier Detect" );
196 :    cputs ( onoff ( iobf [ 1 ] & i_CD ));
197 :    cputs ( ", Data Set Rdy" );
198 :    cputs ( onoff ( iobf [ 1 ] & i_DSR ));
199 :    cputs ( ", Clear to Send" );
200 :    cputs ( onoff ( iobf [ 1 ] & i_CTS ));
201 :    cputs ( ", Ring Indicator" );
202 :    cputs ( onoff ( iobf [ 1 ] & i_RI ));
203 :    cputs ( ".\r\n" );
204 :
205 :    cputs ( l_SRE & iobf [ 1 ] ? "Xmtr SR Empty, " : "" );
206 :    cputs ( l_HRE & iobf [ 1 ] ? "Xmtr HR Empty, " : "" );
207 :    cputs ( l_BRK & iobf [ 1 ] ? "Break Received, " : "" );
208 :    cputs ( l_ER1 & iobf [ 1 ] ? "Framing Error, " : "" );
209 :    cputs ( l_ER2 & iobf [ 1 ] ? "Parity Error, " : "" );
210 :    cputs ( l_ER3 & iobf [ 1 ] ? "Overrun Error, " : "" );
211 :    cputs ( l_RRF & iobf [ 1 ] ? "Rcvr DR Full, " : "" );
212 :    cputs ( "\b\b.\r\n" );
213 :
214 :    cputs ( "Reception errors " );
215 :    if ( iobf [ 2 ] == 1 )
216 :       cputs ( "are encoded as graphics in buffer" );
217 :    else
218 :       cputs ( "set failure flag" );
219 :    cputs ( ".\r\n" );
220 :
221 :    cputs ( "Outgoing Flow Control " );
222 :    if ( iobf [ 3 ] & 4 )
```

Figure 6-4. Continued.   *(more)*

```
223 :     cputs ( "by XON and XOFF" );
224 :   else
225 :     if ( iobf [ 3 ] & 2 )
226 :       cputs ( "by RTS and CTS" );
227 :   else
228 :     if ( iobf [ 3 ] & 1 )
229 :       cputs ( "by DTR and DSR" );
230 :   else
231 :     cputs ( "disabled" );
232 :   cputs ( ".\r\n" );
233 : }
234 :
235 : /*end of cdvutl.c */
```

Figure 6-4. Continued.

Although CDVUTL appears complicated, most of the complexity is concentrated in the routines that map driver bit settings into on-screen display text. Each such mapping requires several lines of source code to generate only a few words of the display report. Table 6-10 summarizes the functions found in this program.

Table 6-10. CDVUTL Program Functions.

Lines	Name	Description
42–45	*main()*	Conventional entry point.
47–150	*disp()*	Main dispatching loop.
152–158	*center()*	Centers text on CRT.
160–166	*iocwr()*	Writes control string to driver with IOCTL Write.
168–170	*onoff()*	Returns pointer to ON or OFF.
172–233	*report()*	Reads driver status and reports it on display.

The long list of *#define* operations at the start of the listing (lines 11 through 33) helps make the bitmapping comprehensible by assigning a symbolic name to each significant bit in the four UART registers.

The *main()* procedure of CDVUTL displays a banner line and then calls the dispatcher routine, *disp()*, to start operation. CDVUTL makes no use of either command-line parameters or the environment, so the usual argument declarations are omitted.

Upon entry to *disp()*, the first action is to establish the default driver as *ASY1* by setting $u = 1$ and opening *ASY1* (line 50); the program then enters an apparent infinite loop (lines 51 through 149).

With each repetition, the loop first prompts for a command (line 52) and then gets the next keystroke and uses it to control a huge *switch()* statement (lines 53 through 145). If no case matches the key pressed, the *switch()* statement does nothing; the program simply displays a report of all current conditions at the selected driver (lines 146 through 148) and then closes the loop back to issue a new prompt and get another keystroke.

However, if the key pressed matches one of the cases in the *switch()* statement, the corresponding command is executed. The digits *1* (line 55) and *2* (line 61) select the driver to be affected. The *?* key (line 105) causes the list of valid command keys to be displayed. The *q* key (line 142) causes the program to terminate by calling *exit(0)* and is the only exit from the infinite loop. The other valid keys all change one or more bits in the IOCTL control string to modify corresponding attributes of the driver and then send the string to the driver by using the MS-DOS IOCTL Write function (Interrupt 21H Function 44H Subfunction 03H) via function *iocwr()* (lines 160 through 166).

After the command is executed (except for the *q* command, which terminates operation of CDVUTL and returns to MS-DOS command level, and the *?* command, which displays the command list), the *report()* function (lines 172 through 233) is called (at line 148) to display all of the driver's attributes, including those just changed. This function issues an IOCTL Read command (Interrupt 21H Function 44H Subfunction 02H, in lines 174 through 178) to get new status information into the control string and then uses a sequence of bit filtering (lines 179 through 232) to translate the obtained status information into words for display.

The special console I/O routines provided in Microsoft C libraries have been used extensively in this routine. Other compilers may require changes in the names of such library routines as *getch* or *dosint* as well as in the names of *#include* files (lines 6 through 9).

Each of the actual command sequences changes only a few bits in one of the 10 bytes of the command string and then writes the string to the driver. A full-featured communications program might make several changes at one time — for example, switching from 7-bit, even parity, XON/XOFF flow control to 8-bit, no parity, without flow control to prevent losing any bytes with values of 11H or 13H while performing a binary file transfer with error-correcting protocol. In such a case, the program could make all required changes to the control string before issuing a single IOCTL Write to put them into effect.

The Traditional Approach

Because the necessary device driver has never been a part of MS-DOS, most communications programs are written to provide and install their own port driver code and remove it before returning to MS-DOS. The second sample program package in this article illustrates this approach. Although the major part of the package is written in Microsoft C, three assembly-language modules are required to provide the hardware interrupt service routines, the exception handler, and faster video display. They are discussed first.

The hardware ISR module

The first module is a handler to service UART interrupts. Code for this handler, including routines to install it at entry and remove it on exit, appears in CH1.ASM, shown in Figure 6-5.

```
 1 :            TITLE   CH1.ASM
 2 :
 3 : ; CH1.ASM -- support file for CTERM.C terminal emulator
 4 : ;          set up to work with COM2
 5 : ;          for use with Microsoft C and SMALL model only...
 6 :
 7 : _TEXT    segment byte public 'CODE'
 8 : _TEXT    ends
 9 : _DATA    segment byte public 'DATA'
10 : _DATA    ends
11 : CONST    segment byte public 'CONST'
12 : CONST    ends
13 : _BSS     segment byte public 'BSS'
14 : _BSS     ends
15 :
16 : DGROUP   GROUP   CONST, _BSS, _DATA
17 :          assume  cs:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP
18 :
19 : _TEXT    segment
20 :
21 :          public  _i_m,_rdmdm,_Send_Byte,_wrtmdm,_set_mdm,_u_m
22 :
23 : bport    EQU     02F8h           ; COM2 base address, use 03F8H for COM1
24 : getiv    EQU     350Bh           ; COM2 vectors, use 0CH for COM1
25 : putiv    EQU     250Bh
26 : imrmsk   EQU     00001000b       ; COM2 mask, use 00000100b for COM1
27 : oiv_o    DW      0               ; old int vector save space
28 : oiv_s    DW      0
29 :
30 : bf_pp    DW      in_bf           ; put pointer (last used)
31 : bf_gp    DW      in_bf           ; get pointer (next to use)
32 : bf_bg    DW      in_bf           ; start of buffer
33 : bf_fi    DW      b_last          ; end of buffer
34 :
35 : in_bf    DB      512 DUP (?)      ; input buffer
36 :
37 : b_last   EQU     $               ; address just past buffer end
38 :
39 : bd_dv    DW      0417h           ; baud rate divisors (0=110 bps)
40 :          DW      0300h           ; code 1 =  150 bps
41 :          DW      0180h           ; code 2 =  300 bps
42 :          DW      00C0h           ; code 3 =  600 bps
43 :          DW      0060h           ; code 4 = 1200 bps
44 :          DW      0030h           ; code 5 = 2400 bps
45 :          DW      0018h           ; code 6 = 4800 bps
46 :          DW      000Ch           ; code 7 = 9600 bps
47 :
48 : _set_mdm proc    near            ; replaces BIOS 'init' function
49 :          PUSH    BP
50 :          MOV     BP,SP           ; establish stackframe pointer
51 :          PUSH    ES              ; save registers
```

Figure 6-5. CH1.ASM   *(more)*

```
 52 :          PUSH     DS
 53 :          MOV      AX,CS           ; point them to CODE segment
 54 :          MOV      DS,AX
 55 :          MOV      ES,AX
 56 :          MOV      AH,[BP+4]       ; get parameter passed by C
 57 :          MOV      DX,BPORT+3      ; point to Line Control Reg
 58 :          MOV      AL,80h          ; set DLAB bit (see text)
 59 :          OUT      DX,AL
 60 :          MOV      DL,AH           ; shift param to BAUD field
 61 :          MOV      CL,4
 62 :          ROL      DL,CL
 63 :          AND      DX,00001110b    ; mask out all other bits
 64 :          MOV      DI,OFFSET bd_dv
 65 :          ADD      DI,DX           ; make pointer to true divisor
 66 :          MOV      DX,BPORT+1      ; set to high byte first
 67 :          MOV      AL,[DI+1]
 68 :          OUT      DX,AL           ; put high byte into UART
 69 :          MOV      DX,BPORT        ; then to low byte
 70 :          MOV      AL,[DI]
 71 :          OUT      DX,AL
 72 :          MOV      AL,AH           ; now use rest of parameter
 73 :          AND      AL,00011111b    ; to set Line Control Reg
 74 :          MOV      DX,BPORT+3
 75 :          OUT      DX,AL
 76 :          MOV      DX,BPORT+2      ; Interrupt Enable Register
 77 :          MOV      AL,1            ; Receive type only
 78 :          OUT      DX,AL
 79 :          POP      DS              ; restore saved registers
 80 :          POP      ES
 81 :          MOV      SP,BP
 82 :          POP      BP
 83 :          RET
 84 : _set_mdm endp
 85 :
 86 : _wrtmdm proc     near            ; write char to modem
 87 : _Send_Byte:                      ; name used by main program
 88 :          PUSH     BP
 89 :          MOV      BP,SP           ; set up pointer and save regs
 90 :          PUSH     ES
 91 :          PUSH     DS
 92 :          MOV      AX,CS
 93 :          MOV      DS,AX
 94 :          MOV      ES,AX
 95 :          MOV      DX,BPORT+4      ; establish DTR, RTS, and OUT2
 96 :          MOV      AL,0Bh
 97 :          OUT      DX,AL
 98 :          MOV      DX,BPORT+6      ; check for on line, CTS
 99 :          MOV      BH,30h
100 :          CALL     w_tmr
101 :          JNZ      w_out           ; timed out
102 :          MOV      DX,BPORT+5      ; check for UART ready
```

(more)

Figure 6-5. Continued.

```
103 :           MOV     BH,20h
104 :           CALL    w_tmr
105 :           JNZ     w_out         ; timed out
106 :           MOV     DX,BPORT      ; send out to UART port
107 :           MOV     AL,[BP+4]     ; get char passed from C
108 :           OUT     DX,AL
109 : w_out:    POP     DS            ; restore saved regs
110 :           POP     ES
111 :           MOV     SP,BP
112 :           POP     BP
113 :           RET
114 : _wrtmdm   endp
115 :
116 : _rdmdm    proc    near          ; reads byte from buffer
117 :           PUSH    BP
118 :           MOV     BP,SP         ; set up ptr, save regs
119 :           PUSH    ES
120 :           PUSH    DS
121 :           MOV     AX,CS
122 :           MOV     DS,AX
123 :           MOV     ES,AX
124 :           MOV     AX,0FFFFh     ; set for EOF flag
125 :           MOV     BX,bf_gp      ; use "get" ptr
126 :           CMP     BX,bf_pp      ; compare to "put"
127 :           JZ      nochr         ; same, empty
128 :           INC     BX            ; else char available
129 :           CMP     BX,bf_fi      ; at end of bfr?
130 :           JNZ     noend         ; no
131 :           MOV     BX,bf_bg      ; yes, set to beg
132 : noend:    MOV     AL,[BX]       ; get the char
133 :           MOV     bf_gp,BX      ; update "get" ptr
134 :           INC     AH            ; zero AH as flag
135 : nochr:    POP     DS            ; restore regs
136 :           POP     ES
137 :           MOV     SP,BP
138 :           POP     BP
139 :           RET
140 : _rdmdm    endp
141 :
142 : w_tmr     proc    near
143 :           MOV     BL,1          ; wait timer, double loop
144 : w_tm1:    SUB     CX,CX         ; set up inner loop
145 : w_tm2:    IN      AL,DX         ; check for requested response
146 :           MOV     AH,AL         ; save what came in
147 :           AND     AL,BH         ; mask with desired bits
148 :           CMP     AL,BH         ; then compare
149 :           JZ      w_tm3         ; got it, return with ZF set
150 :           LOOP    w_tm2         ; else keep trying
151 :           DEC     BL            ; until double loop expires
152 :           JNZ     w_tm1
153 :           OR      BH,BH         ; timed out, return NZ
```

Figure 6-5. Continued.   *(more)*

```
154 : w_tm3:   RET
155 : w_tmr    endp
156 :
157 : ; hardware interrupt service routine
158 : rts_m:   CLI
159 :          PUSH   DS              ; save all regs
160 :          PUSH   AX
161 :          PUSH   BX
162 :          PUSH   CX
163 :          PUSH   DX
164 :          PUSH   CS              ; set DS same as CS
165 :          POP    DS
166 :          MOV    DX,BPORT        ; grab the char from UART
167 :          IN     AL,DX
168 :          MOV    BX,bf_pp        ; use "put" ptr
169 :          INC    BX              ; step to next slot
170 :          CMP    BX,bf_fi        ; past end yet?
171 :          JNZ    nofix           ; no
172 :          MOV    BX,bf_bg        ; yes, set to begin
173 : nofix:   MOV    [BX],AL         ; put char in buffer
174 :          MOV    bf_pp,BX        ; update "put" ptr
175 :          MOV    AL,20h          ; send EOI to 8259 chip
176 :          OUT    20h,AL
177 :          POP    DX              ; restore regs
178 :          POP    CX
179 :          POP    BX
180 :          POP    AX
181 :          POP    DS
182 :          IRET
183 :
184 : _i_m     proc   near            ; install modem service
185 :          PUSH   BP
186 :          MOV    BP,SP           ; save all regs used
187 :          PUSH   ES
188 :          PUSH   DS
189 :          MOV    AX,CS           ; set DS,ES=CS
190 :          MOV    DS,AX
191 :          MOV    ES,AX
192 :          MOV    DX,BPORT+1      ; Interrupt Enable Reg
193 :          MOV    AL,0Fh          ; enable all ints now
194 :          OUT    DX,AL
195 :
196 : im1:     MOV    DX,BPORT+2      ; clear junk from UART
197 :          IN     AL,DX           ; read IID reg of UART
198 :          MOV    AH,AL           ; save what came in
199 :          TEST   AL,1            ; anything pending?
200 :          JNZ    im5             ; no, all clear now
201 :          CMP    AH,0            ; yes, Modem Status?
202 :          JNZ    im2             ; no
203 :          MOV    DX,BPORT+6      ; yes, read MSR to clear
204 :          IN     AL,DX
```

Figure 6-5. Continued.

(more)

```
205 : im2:     CMP     AH,2              ; Transmit HR empty?
206 :          JNZ     im3               ; no (no action needed)
207 : im3:     CMP     AH,4              ; Received Data Ready?
208 :          JNZ     im4               ; no
209 :          MOV     DX,BPORT          ; yes, read it to clear
210 :          IN      AL,DX
211 : im4:     CMP     AH,6              ; Line Status?
212 :          JNZ     im1               ; no, check for more
213 :          MOV     DX,BPORT+5        ; yes, read LSR to clear
214 :          IN      AL,DX
215 :          JMP     im1               ; then check for more
216 :
217 : im5:     MOV     DX,BPORT+4        ; set up working conditions
218 :          MOV     AL,0Bh            ; DTR, RTS, OUT2 bits
219 :          OUT     DX,AL
220 :          MOV     AL,1              ; enable RCV interrupt only
221 :          MOV     DX,BPORT+1
222 :          OUT     DX,AL
223 :          MOV     AX,GETIV          ; get old int vector
224 :          INT     21h
225 :          MOV     oiv_o,BX          ; save for restoring later
226 :          MOV     oiv_s,ES
227 :          MOV     DX,OFFSET rts_m   ; set in new one
228 :          MOV     AX,PUTIV
229 :          INT     21h
230 :          IN      AL,21h            ; now enable 8259 PIC
231 :          AND     AL,NOT IMRMSK
232 :          OUT     21h,AL
233 :          MOV     AL,20h            ; then send out an EOI
234 :          OUT     20h,AL
235 :          POP     DS                ; restore regs
236 :          POP     ES
237 :          MOV     SP,BP
238 :          POP     BP
239 :          RET
240 : _i_m     endp
241 :
242 : _u_m     proc    near              ; uninstall modem service
243 :          PUSH    BP
244 :          MOV     BP,SP             ; save registers
245 :          IN      AL,21h            ; disable COM int in 8259
246 :          OR      AL,IMRMSK
247 :          OUT     21h,AL
248 :          PUSH    ES
249 :          PUSH    DS
250 :          MOV     AX,CS             ; set same as CS
251 :          MOV     DS,AX
252 :          MOV     ES,AX
253 :          MOV     AL,0              ; disable UART ints
254 :          MOV     DX,BPORT+1
255 :          OUT     DX,AL
```

Figure 6-5. Continued.   *(more)*

```
256 :             MOV     DX,oiv_o        ;'restore original vector
257 :             MOV     DS,oiv_s
258 :             MOV     AX,PUTIV
259 :             INT     21h
260 :             POP     DS              ; restore registers
261 :             POP     ES
262 :             MOV     SP,BP
263 :             POP     BP
264 :             RET
265 : _u_m        endp
266 :
267 : _TEXT       ends
268 :
269 :             END
```

Figure 6-5. Continued.

The routines in CH1 are set up to work only with port COM2; to use them with COM1, the three symbolic constants BPORT (base address), GETIV, and PUTIV must be changed to match the COM1 values. Also, as presented, this code is for use with the Microsoft C small memory model only; for use with other memory models, the C compiler manuals should be consulted for making the necessary changes. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

The parts of CH1 are listed in Table 6-11, as they occur in the listing. The leading underscore that is part of the name for each of the six functions is supplied by the C compiler; within the C program that calls the function, the underscore is omitted.

Table 6-11. CH1 Module Functions.

Lines	Name	Description
1–26		Administrative details.
27–46		Data areas.
48–84	_set_mdm	Initializes UART as specified by parameter passed from C.
86–114	_wrtmdm	Outputs character to UART.
87	_Send_Byte	Entry point for use if flow control is added to system.
116–140	_rdmdm	Gets character from buffer where ISR put it, or signals that no character available.
142–155	w_tmr	Wait timer; internal routine used to prevent infinite wait in case of problems.
157–182	rts_m	Hardware ISR; installed by _i_m and removed by _u_m.
184–240	_i_m	Installs ISR, saving old interrupt vector.
242–265	_u_m	Uninstalls ISR, restoring saved interrupt vector.

For simplest operation, the ISR used in this example (unlike the device driver) services *only* the received-data interrupt; the other three types of IRQ are disabled at the UART. Each time a byte is received by the UART, the ISR puts it into the buffer. The *_rdmdm* code, when called by the C program, gets a byte from the buffer if one is available. If not, *_rdmdm* returns the C EOF code (−1) to indicate that no byte can be obtained.

To send a byte, the C program can call either *_Send_Byte* or *_wrtmdm*; in the package as shown, these are alternative names for the same routine. In the more complex program from which this package was adapted, *_Send_Byte* is called when flow control is desired and the flow-control routine calls *_wrtmdm*. To implement flow control, line 87 should be deleted from CH1.ASM and a control function named *Send_Byte()* should be added to the main C program. Flow-control tests must occur in *Send_Byte()*; *_wrtmdm* performs the actual port interfacing.

To set the modem baud rate, word length, and parity, *_set_mdm* is called from the C program, with a setup parameter passed as an argument. The format of this parameter is shown in Table 6-12 and is identical to the IBM BIOS Interrupt 14H Function 00H (Initialization).

Table 6-12. *set_mdm()* **Parameter Coding.**

Binary	Meaning
000xxxxx	Set to 110 bps
001xxxxx	Set to 150 bps
010xxxxx	Set to 300 bps
011xxxxx	Set to 600 bps
100xxxxx	Set to 1200 bps
101xxxxx	Set to 2400 bps
110xxxxx	Set to 4800 bps
111xxxxx	Set to 9600 bps
xxxx0xxx	No parity
xxx01xxx	ODD Parity
xxx11xxx	EVEN Parity
xxxxx0xx	1 stop bit
xxxxx1xx	2 stop bits (1.5 if WL = 5)
xxxxxx00	Word length = 5
xxxxxx01	Word length = 6
xxxxxx10	Word length = 7
xxxxxx11	Word length = 8

The CH1 code provides a 512-byte ring buffer for incoming data; the buffer size should be adequate for reception at speeds up to 2400 bps without loss of data during scrolling.

The exception-handler module

For the ISR handler of CH1 to be usable, an exception handler is needed to prevent return of control to MS-DOS before _u_m restores the ISR vector to its original value. If a program using this code returns to MS-DOS without calling _u_m, the system is virtually certain to crash when line noise causes a received-data interrupt and the ISR code is no longer in memory.

A replacement exception handler (CH1A.ASM), including routines for installation, access, and removal, is shown in Figure 6-6. Like the ISR, this module is designed to work with Microsoft C (again, the small memory model only).

Note: This module does not provide for fatal disk errors; if one occurs, immediate restarting is necessary. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

```
 1 :            TITLE   CH1A.ASM
 2 :
 3 : ; CH1A.ASM -- support file for CTERM.C terminal emulator
 4 : ;        this set of routines replaces Ctrl-C/Ctrl-BREAK
 5 : ;        usage: void set_int(), rst_int();
 6 : ;                   int broke();     /* boolean if BREAK      */
 7 : ;        for use with Microsoft C and SMALL model only...
 8 :
 9 : _TEXT     segment byte public 'CODE'
10 : _TEXT     ends
11 : _DATA     segment byte public 'DATA'
12 : _DATA     ends
13 : CONST     segment byte public 'CONST'
14 : CONST     ends
15 : _BSS      segment byte public 'BSS'
16 : _BSS      ends
17 :
18 : DGROUP    GROUP   CONST, _BSS, _DATA
19 :           ASSUME  CS:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP
20 :
21 : _DATA     SEGMENT BYTE PUBLIC 'DATA'
22 :
23 : OLDINT1B DD       0                ; storage for original INT 1BH vector
24 :
25 : _DATA     ENDS
26 :
27 : _TEXT     SEGMENT
28 :
29 :           PUBLIC  _set_int,_rst_int,_broke
30 :
31 : myint1b:
32 :           mov     word ptr cs:brkflg,1Bh       ; make it nonzero
33 :           iret
```

Figure 6-6. CH1A.ASM.   *(more)*

```
34 :
35 : myint23:
36 :          mov      word ptr cs:brkflg,23h           ; make it nonzero
37 :          iret
38 :
39 : brkflg  dw       0                       ; flag that BREAK occurred
40 :
41 : _broke  proc     near                    ; returns 0 if no break
42 :          xor      ax,ax                   ; prepare to reset flag
43 :          xchg     ax,cs:brkflg            ; return current flag value
44 :          ret
45 : _broke  endp
46 :
47 : _set_int proc near
48 :          mov      ax,351bh                ; get interrupt vector for 1BH
49 :          int      21h                     ; (don't need to save for 23H)
50 :          mov      word ptr oldint1b,bx    ; save offset in first word
51 :          mov      word ptr oldint1b+2,es  ; save segment in second word
52 :
53 :          push     ds                      ; save our data segment
54 :          mov      ax,cs                   ; set DS to CS for now
55 :          mov      ds,ax
56 :          lea      dx,myint1b              ; DS:DX points to new routine
57 :          mov      ax,251bh                ; set interrupt vector
58 :          int      21h
59 :          mov      ax,cs                   ; set DS to CS for now
60 :          mov      ds,ax
61 :          lea      dx,myint23              ; DS:DX points to new routine
62 :          mov      ax,2523h                ; set interrupt vector
63 :          int      21h
64 :          pop      ds                      ; restore data segment
65 :          ret
66 : _set_int endp
67 :
68 : _rst_int proc near
69 :          push     ds                      ; save our data segment
70 :          lds      dx,oldint1b             ; DS:DX points to original
71 :          mov      ax,251bh                ; set interrupt vector
72 :          int      21h
73 :          pop      ds                      ; restore data segment
74 :          ret
75 : _rst_int endp
76 :
77 : _TEXT   ends
78 :
79 :          END
```

Figure 6-6. Continued.

The three functions in CH1A are _set_int, which saves the old vector value for Interrupt
1BH (ROM BIOS Control-Break) and then resets both that vector and the one for Interrupt
23H (Control-C Handler Address) to internal ISR code; _rst_int, which restores the

original value for the Interrupt 1BH vector; and _broke, which returns the present value of
an internal flag (and always clears the flag, just in case it had been set). The internal flag is
set to a nonzero value in response to either of the revectored interrupts and is tested from
the main C program via the _broke function.

The video display module

The final assembly-language module (CH2.ASM) used by the second package is shown
in Figure 6-7. This module provides convenient screen clearing and cursor positioning via
direct calls to the IBM BIOS, but this can be eliminated with minor rewriting of the rou-
tines that call its functions. In the original, more complex program (DT115.EXE, available
from DL6 in the CLMFORUM of CompuServe) from which CTERM was derived, this mod-
ule provided windowing capability in addition to improved display speed.

```
 1 :          TITLE   CH2.ASM
 2 :
 3 : ; CH2.ASM -- support file for CTERM.C terminal emulator
 4 : ;          for use with Microsoft C and SMALL model only...
 5 :
 6 : _TEXT    segment byte public 'CODE'
 7 : _TEXT    ends
 8 : _DATA    segment byte public 'DATA'
 9 : _DATA    ends
10 : CONST    segment byte public 'CONST'
11 : CONST    ends
12 : _BSS     segment byte public 'BSS'
13 : _BSS     ends
14 :
15 : DGROUP   GROUP   CONST, _BSS, _DATA
16 :          assume  CS:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP
17 :
18 : _TEXT    segment
19 :
20 :          public  __cls,__color,__deol,__i_v,__key,__wrchr,__wrpos
21 :
22 : atrib    DB      0               ; attribute
23 : _colr    DB      0               ; color
24 : v_bas    DW      0               ; video segment
25 : v_ulc    DW      0               ; upper left corner cursor
26 : v_lrc    DW      184Fh           ; lower right corner cursor
27 : v_col    DW      0               ; current col/row
28 :
29 : __key    proc    near            ; get keystroke
30 :          PUSH    BP
31 :          MOV     AH,1            ; check status via BIOS
32 :          INT     16h
33 :          MOV     AX,0FFFFh
34 :          JZ      key00           ; none ready, return EOF
35 :          MOV     AH,0            ; have one, read via BIOS
```

Figure 6-7. CH2.ASM.   (*more*)

```
36 :          INT      16h
37 : key00:   POP      BP
38 :          RET
39 : __key    endp
40 :
41 : __wrchr  proc     near
42 :          PUSH     BP
43 :          MOV      BP,SP
44 :          MOV      AL,[BP+4]         ; get char passed by C
45 :          CMP      AL,' '
46 :          JNB      prchr            ; printing char, go do it
47 :          CMP      AL,8
48 :          JNZ      notbs
49 :          DEC      BYTE PTR v_col   ; process backspace
50 :          MOV      AL,byte ptr v_col
51 :          CMP      AL,byte ptr v_ulc
52 :          JB       nxt_c           ; step to next column
53 :          JMP      norml
54 :
55 : notbs:   CMP      AL,9
56 :          JNZ      notht
57 :          MOV      AL,byte ptr v_col       ; process HTAB
58 :          ADD      AL,8
59 :          AND      AL,0F8h
60 :          MOV      byte ptr v_col,AL
61 :          CMP      AL,byte ptr v_lrc
62 :          JA       nxt_c
63 :          JMP      SHORT    norml
64 :
65 : notht:   CMP      AL,0Ah
66 :          JNZ      notlf
67 :          MOV      AL,byte ptr v_col+1      ; process linefeed
68 :          INC      AL
69 :          CMP      AL,byte ptr v_lrc+1
70 :          JBE      noht1
71 :          CALL     scrol
72 :          MOV      AL,byte ptr v_lrc+1
73 : noht1:   MOV      byte ptr v_col+1,AL
74 :          JMP      SHORT    norml
75 :
76 : notlf:   CMP      AL,0Ch
77 :          JNZ      ck_cr
78 :          CALL     __cls             ; process formfeed
79 :          JMP      SHORT    ignor
80 :
81 : ck_cr:   CMP      AL,0Dh
82 :          JNZ      ignor            ; ignore all other CTL chars
83 :          MOV      AL,byte ptr v_ulc       ; process CR
84 :          MOV      byte ptr v_col,AL
85 :          JMP      SHORT    norml
86 :
```

Figure 6-7. Continued.   *(more)*

```
 87 : prchr:   MOV.   AH,_colr         ; process printing char
 88 :          PUSH   AX
 89 :          XOR    AH,AH
 90 :          MOV    AL,byte ptr v_col+1
 91 :          PUSH   AX
 92 :          MOV    AL,byte ptr v_col
 93 :          PUSH   AX
 94 :          CALL   wrtvr
 95 :          MOV    SP,BP
 96 : nxt_c:   INC    BYTE PTR v_col   ; advance to next column
 97 : ·        MOV    AL,byte ptr v_col
 98 :          CMP    AL,byte ptr v_lrc
 99 :          JLE    norml
100 :          MOV    AL,0Dh           ; went off end, do CR/LF
101 :          PUSH   AX
102 :          CALL   __wrchr
103 :          POP    AX
104 :          MOV    AL,0Ah
105 :          PUSH   AX
106 :          CALL   __wrchr
107 :          POP    AX
108 : norml:   CALL   set_cur
109 : ignor:   MOV    SP,BP
110 :          POP    BP
111 :          RET
112 : __wrchr endp
113 :
114 : __i_v    proc   near             ; establish video base segment
115 :          PUSH   BP
116 :          MOV    BP,SP
117 :          MOV    AX,0B000h        ; mono, B800 for CGA
118 :          MOV    v_bas,AX         ; could be made automatic
119 :          MOV    SP,BP
120 :          POP    BP
121 :          RET
122 : __i_v    endp
123 :
124 : __wrpos proc    near             ; set cursor position
125 :          PUSH   BP
126 :          MOV    BP,SP
127 :          MOV    DH,[BP+4]        ; row from C program
128 :          MOV    DL,[BP+6]        ; col from C program
129 :          MOV    v_col,DX         ; cursor position
130 :          MOV    BH,atrib         ; attribute
131 :          MOV    AH,2
132 :          PUSH   BP
133 :          INT    10h
134 :          POP    BP
135 :          MOV    AX,v_col         ; return cursor position
136 :          MOV    SP,BP
137 :          POP    BP
```

Figure 6-7. Continued.   *(more)*

```
138 :           RET
139 : __wrpos endp
140 :
141 : set_cur proc    near            ; set cursor to v_col
142 :           PUSH    BP
143 :           MOV     BP,SP
144 :           MOV     DX,v_col        ; use where v_col says
145 :           MOV     BH,atrib
146 :           MOV     AH,2
147 :           PUSH    BP
148 :           INT     10h
149 :           POP     BP
150 :           MOV     AX,v_col
151 :           MOV     SP,BP
152 :           POP     BP
153 :           RET
154 : set_cur endp
155 :
156 : __color proc    near            ; _color(fg, bg)
157 :           PUSH    BP
158 :           MOV     BP,SP
159 :           MOV     AH,[BP+6]       ; background from C
160 :           MOV     AL,[BP+4]       ; foreground from C
161 :           MOV     CX,4
162 :           SHL     AH,CL
163 :           AND     AL,0Fh
164 :           OR      AL,AH           ; pack up into 1 byte
165 :           MOV     _colr,AL        ; store for handler's use
166 :           XOR     AH,AH
167 :           MOV     SP,BP
168 :           POP     BP
169 :           RET
170 : __color endp
171 :
172 : scrol   proc    near            ; scroll CRT up by one line
173 :           PUSH    BP
174 :           MOV     BP,SP
175 :           MOV     AL,1            ; count of lines to scroll
176 :           MOV     CX,v_ulc
177 :           MOV     DX,v_lrc
178 :           MOV     BH,_colr
179 :           MOV     AH,6
180 :           PUSH    BP
181 :           INT     10h             ; use BIOS
182 :           POP     BP
183 :           MOV     SP,BP
184 :           POP     BP
185 :           RET
186 : scrol   endp
187 :
188 : __cls   proc    near            ; clear CRT
```

Figure 6-7. Continued.   *(more)*

```
189 :             PUSH    BP
190 :             MOV     BP,SP
191 :             MOV     AL,0            ; flags CLS to BIOS
192 :             MOV     CX,v_ulc
193 :             MOV     v_col,CX        ; set to HOME
194 :             MOV     DX,v_lrc
195 :             MOV     BH,_colr
196 :             MOV     AH,6
197 :             PUSH    BP
198 :             INT     10h             ; use BIOS scroll up
199 :             POP     BP
200 :             CALL    set_cur         ; cursor to HOME
201 :             MOV     SP,BP
202 :             POP     BP
203 :             RET
204 : __cls     endp
205 :
206 : __deol    proc    near            ; delete to end of line
207 :             PUSH    BP
208 :             MOV     BP,SP
209 :             MOV     AL,' '
210 :             MOV     AH,_colr        ; set up blanks
211 :             PUSH    AX
212 :             MOV     AL,byte ptr v_col+1
213 :             XOR     AH,AH           ; set up row value
214 :             PUSH    AX
215 :             MOV     AL,byte ptr v_col
216 :
217 : deol1:    CMP     AL,byte ptr v_lrc
218 :             JA      deol2           ; at RH edge
219 :             PUSH    AX              ; current location
220 :             CALL    wrtvr           ; write a blank
221 :             POP     AX
222 :             INC     AL              ; next column
223 :             JMP     deol1           ; do it again
224 :
225 : deol2:    MOV     AX,v_col        ; return cursor position
226 :             MOV     SP,BP
227 :             POP     BP
228 :             RET
229 : __deol    endp
230 :
231 : wrtvr     proc    near            ; write video RAM (col, row, char/atr)
232 :             PUSH    BP
233 :             MOV     BP,SP           ; set up arg ptr
234 :             MOV     DL,[BP+4]       ; column
235 :             MOV     DH,[BP+6]       ; row
236 :             MOV     BX,[BP+8]       ; char/atr
237 :             MOV     AL,80           ; calc offset
238 :             MUL     DH
239 :             XOR     DH,DH
```

Figure 6-7. Continued.   *(more)*

```
240 :          ADD     AX,DX
241 :          ADD     AX,AX       ; adjust bytes to words
242 :          PUSH    ES          ; save seg reg
243 :          MOV     DI,AX
244 :          MOV     AX,v_bas    ; set up segment
245 :          MOV     ES,AX
246 :          MOV     AX,BX       ; get the data
247 :          STOSW               ; put on screen
248 :          POP     ES          ; restore regs
249 :          MOV     SP,BP
250 :          POP     BP .
251 :          RET
252 : wrtvr    endp
253 :
254 : _TEXT    ends
255 :
256 :          END
```

Figure 6-7. Continued.

The sample smarter terminal emulator: CTERM.C

Given the interrupt handler (CH1), exception handler (CH1A), and video handler (CH2), a simple terminal emulation program (CTERM.C) can be presented. The major functions of the program are written in Microsoft C; the listing is shown in Figure 6-8.

```
 1 : /* Terminal Emulator    (cterm.c)
 2 : *        Jim Kyle, 1987
 3 : *
 4 : *        Uses files CH1, CH1A, and CH2 for MASM support...
 5 : */
 6 :
 7 : #include <stdio.h>
 8 : #include <conio.h>                /* special console i/o   */
 9 : #include <stdlib.h>               /* misc definitions      */
10 : #include <dos.h>                  /* defines intdos()      */
11 : #include <string.h>
12 : #define BRK   'C'-'@'             /* control characters    */
13 : #define ESC   '['-'@'
14 : #define XON   'Q'-'@'
15 : #define XOFF  'S'-'@'
16 :
17 : #define True   1
18 : #define False  0
19 :
20 : #define Is_Function_Key(C) ( (C) == ESC )
21 :
22 : static char capbfr [ 4096 ];     /* capture buffer        */
23 : static int wh,
24 :      ws;
```

Figure 6-8. CTERM.C.   *(more)*