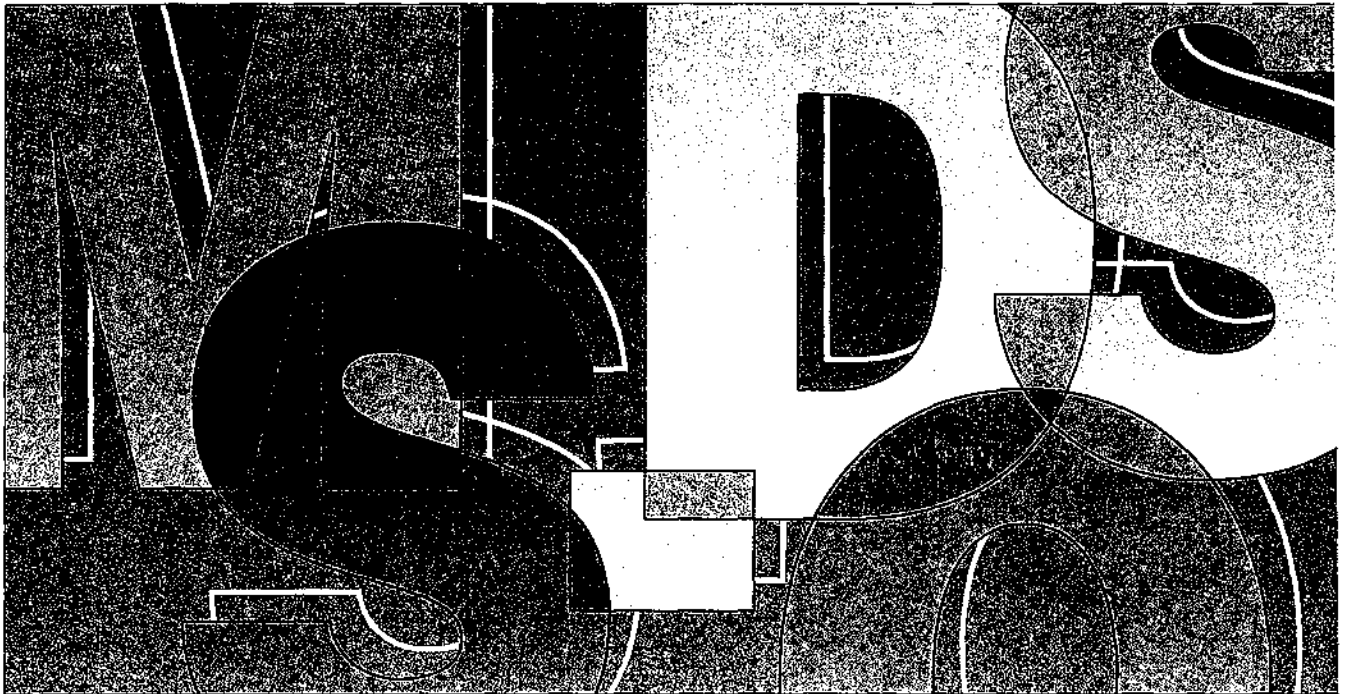


Unabridged

The MS-DOS[®] Encyclopedia



Foreword, Bill Gates
General Editor, Ray Duncan

OLYMPUS EX. 1010 - 1/1582

The

MS-DOS[®]

Encyclopedia

Published by
Microsoft Press
A Division of Microsoft Corporation
16011 NE 36th Way, Box 97017, Redmond, Washington 98073-9717
Copyright © 1988 by Microsoft Press
All rights reserved. No part of the contents of this book
may be reproduced or transmitted in any form or by any means
without the written permission of the publisher.

Library of Congress Cataloging in Publication Data
The MS-DOS encyclopedia : versions 1.0 through 3.2 /
editor, Ray Duncan.
p. cm.

Includes indexes.
1. MS-DOS (Computer operating system) I. Duncan, Ray, 1952-
II. Microsoft Press.
QA76.76.063M74 1988 87-21452
005.4'46--dc19 CIP
ISBN 1-55615-174-8

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 RMRM 3 2 1 0 9 8

Distributed to the book trade in the
United States by Harper & Row.

Distributed to the book trade in
Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the
United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging in Publication Data available

IBM®, IBM AT®, PS/2®, and TopView® are registered trademarks of International Business Machines Corporation.
GW-BASIC®, Microsoft®, MS®, MS-DOS®, SOFTCARD®, and XENIX® are registered trademarks of
Microsoft Corporation.

Microsoft Press gratefully acknowledges permission to reproduce material listed below.

Page 4: Courtesy The Computer Museum.

Pages 5, 11, 42: Intel 4004, 8008, 8080, 8086, and 80286 microprocessor photographs. Courtesy Intel Corporation.

Page 6: Reprinted from *Popular Electronics*, January 1975 Copyright © 1975 Ziff Communications Company.

Page 13: Reprinted with permission of Rod Brock.

Page 16: Reprinted with permission of The Seattle Times Copyright © 1983.

Pages 19, 34, 42: IBM PC advertisements and photographs of the PC, PC/XT, and PC/AT reproduced with
permission of International Business Machines Corporation Copyright © 1981, 1982, 1984. All rights reserved.

Page 21: "Big IBM's Little Computer" Copyright © 1981 by The New York Times Company. Reprinted by
permission.

"IBM Announces New Microcomputer System" Reprinted with permission of InfoWorld Copyright © 1981.

"IBM really gets personal" Reprinted with permission of Personal Computing Copyright © 1981.

"Personal Computer from IBM" Reprinted from DATAMATION Magazine, October 1981 Copyright © by Cahners
Publishing Company.

"IBM's New Line Likely to Shake up the Market for Personal Computers" Reprinted by permission of The Wall
Street Journal Copyright © Dow Jones & Company, Inc. 1981. All Rights Reserved.

Page 36: "Irresistible DOS 3.0" and "The Ascent of DOS" Reprinted from *PC Tech Journal*,
December 1984 and October 1986. Copyright © 1984, 1986 Ziff Communications Company.

"MS-DOS 2.00: A Hands-On Tutorial" Reprinted by permission of PC World from Volume 1, Issue 3, March 1983,
published at 501 Second Street, Suite 600, San Francisco, CA 94107.

Special thanks to Bob O'Rear, Aaron Reynolds, and Kenichi Ikeda.

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. Brunzman

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

Foreword by Bill Gates	<i>xiii</i>
Preface by Ray Duncan	<i>xv</i>
Introduction	<i>xvii</i>
Section I: The Development of MS-DOS	1
Section II: Programming in the MS-DOS Environment	47
Part A: Structure of MS-DOS	
Article 1: An Introduction to MS-DOS	51
Article 2: The Components of MS-DOS	61
Article 3: MS-DOS Storage Devices	85
Part B: Programming for MS-DOS	
Article 4: Structure of an Application Program	107
Article 5: Character Device Input and Output	149
Article 6: Interrupt-Driven Communications	167
Article 7: File and Record Management	247
Article 8: Disk Directories and Volume Labels	279
Article 9: Memory Management	297
Article 10: The MS-DOS EXEC Function	321
Part C: Customizing MS-DOS	
Article 11: Terminate-and-Stay-Resident Utilities	347
Article 12: Exception Handlers	385
Article 13: Hardware Interrupt Handlers	409
Article 14: Writing MS-DOS Filters	429
Article 15: Installable Device Drivers	447
Part D: Directions of MS-DOS	
Article 16: Writing Applications for Upward Compatibility	489
Article 17: Windows	499
Part E: Programming Tools	
Article 18: Debugging in the MS-DOS Environment	541
Article 19: Object Modules	643
Article 20: The Microsoft Object Linker	701

Section III: User Commands **723**

Introduction 725

User commands are listed in alphabetic order. This section includes ANSI.SYS, BATCH, CONFIG.SYS, DRIVER.SYS, EDLIN, RAMDRIVE.SYS, and VDISK.SYS.

Section IV: Programming Utilities **961**

Introduction 963

CREF 967

EXE2BIN 971

EXEMOD 974

EXEPACK 977

LIB 980

LINK 987

MAKE 999

MAPSYM 1004

MASM 1007

Microsoft Debuggers:

DEBUG 1020

SYMDEB 1054

CodeView 1157

Section V: System Calls **1175**

Introduction 1177

System calls are listed in numeric order.

Appendixes **1431**

Appendix A: MS-DOS Version 3.3 1433

Appendix B: Critical Error Codes 1459

Appendix C: Extended Error Codes 1461

Appendix D: ASCII and IBM Extended ASCII Character Sets 1465

Appendix E: EBCDIC Character Set 1469

Appendix F: ANSI.SYS Key and Extended Key Codes 1471

Appendix G: File Control Block (FCB) Structure 1473

Appendix H: Program Segment Prefix (PSP) Structure 1477

Appendix I: 8086/8088/80286/80386 Instruction Sets 1479

Appendix J: Common MS-DOS Filename Extensions 1485

Appendix K: Segmented (New) .EXE File Header Format 1487

Appendix L: Intel Hexadecimal Object File Format 1499

Appendix M: 8086/8088 Software Compatibility Issues 1507

Appendix N: An Object Module Dump Utility 1509

Appendix O: IBM PC BIOS Calls 1513

Indexes**1531**

Subject 1533
Commands and System Calls 1565

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 ANSL.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.

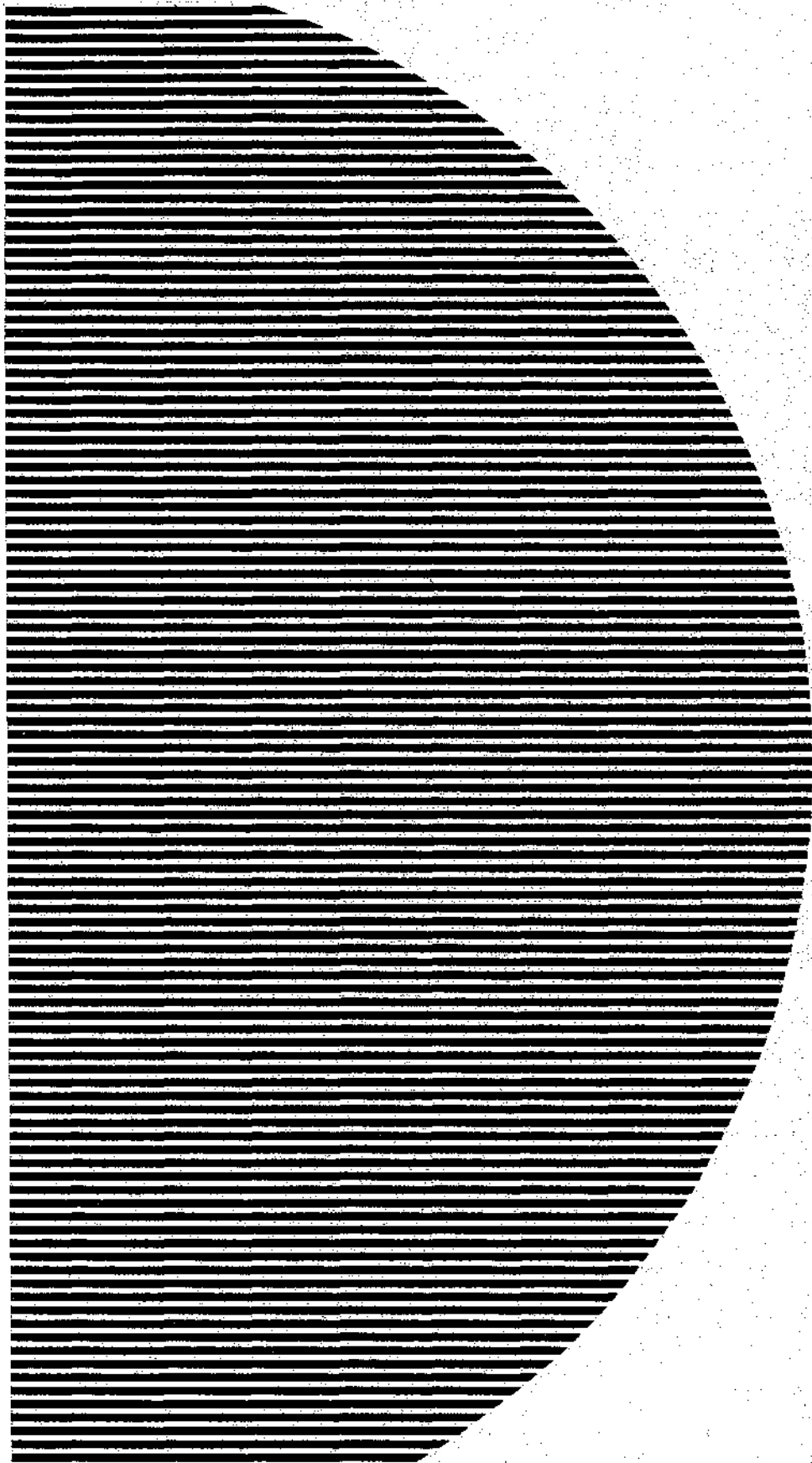
Typography and Terminology

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

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

[TEXTTAB] zero (1 byte)
 pointer to next line (2 bytes)
 binary line # (2 bytes)
 character on line (see note 11)
 zero (1 byte)

<Repeat above for each line>

[VARTAB] zero (2 bytes)
 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 -

[STREND] Repeats for each array
 lowest location for strings

[STKTOP] Free space (STKTOP is here)
 most recent stack entry

[FRETOP] bottom of stack / top of location for strings

[FRETOP] free space
 current string usage

[MEMSIZ] STRING\$
 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: 0W GETNEM
          (2 bytes-$1 low byte of
           GETNEM address
           $2 high byte of
           GETNEM address)

START:  LXI H,0
        GETNEM: 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)
    
```

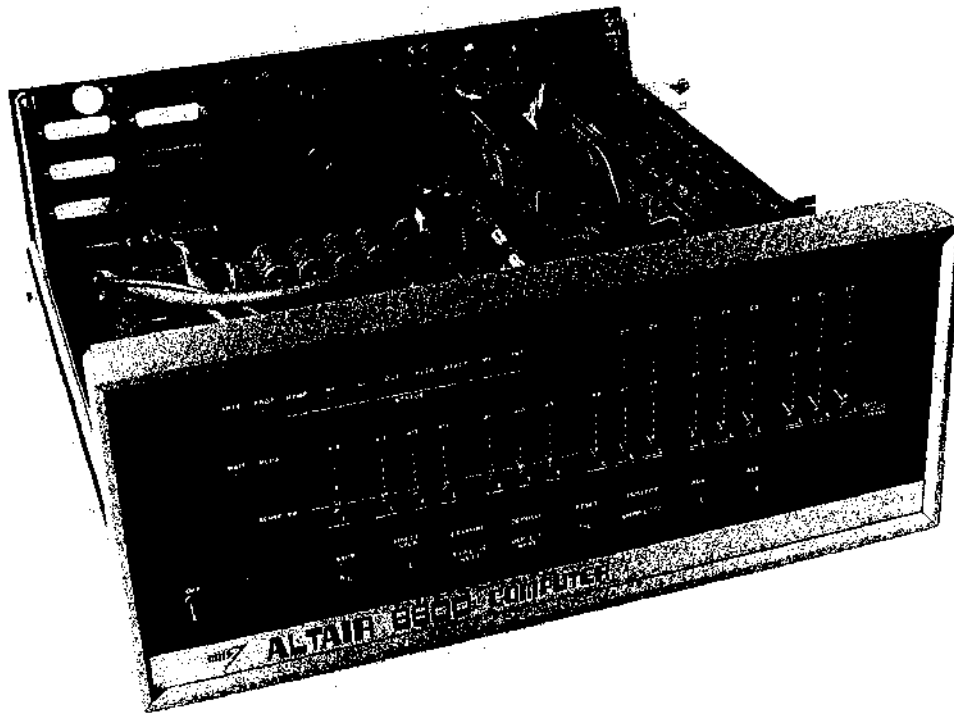
Punch a paper tape with loader. a 043 start byte. the byte to be stored at loc 0, the byte to be stored at 1, - - etc. Start at START, making sure the memory the loader is in is unprotected. Make sure you don't wipe out the loader by loading on top of it.

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

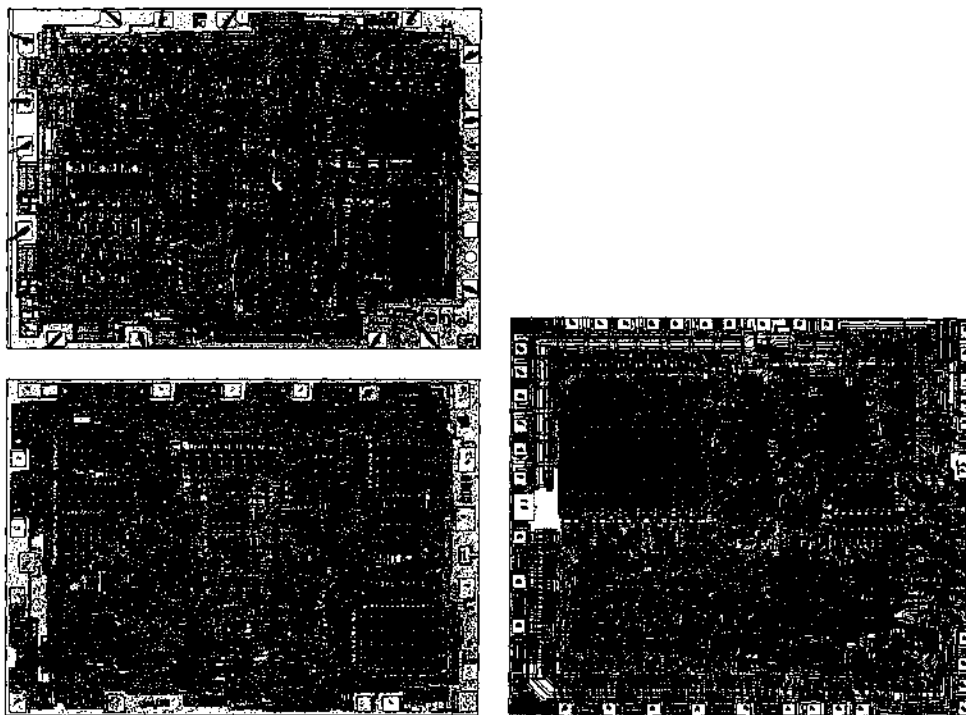
Gates and Allen's early BASIC for the Altair was loaded from paper tape after the bootstrap to load the tape was entered into memory by flipping switches on the front panel of the computer. In late 1975, however, MITS decided to release a floppy-disk system for the Altair—the first retail floppy-disk 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.



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.

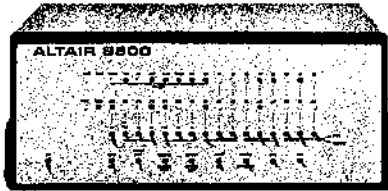
Gates and Allen had become friends in their teens, while attending Lakeside School in Seattle. They shared an intense interest in computers, and by the time Gates was in the tenth grade, they and another friend named Paul Gilbert had formed a company called Traf-O-Data to produce a machine that automated 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!

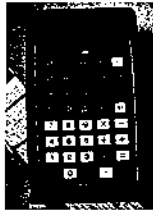
**World's First Minicomputer Kit
to Rival Commercial Models...**

"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

[LOWMEM] (low memory)

zero (2 bytes)

pointer to next line (2 bytes)

binary line # (2 bytes)

character on line (see note 11) (1 byte)

zero (1 byte)

<Repeat above for each line>

[VARTAB] zero (2 bytes)

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

lowest location for stack

[STRTOP] Free space (SP can be in here)

[STKTOP] most recent stack entry

[FREETOP] bottom of stack / top of location for strings

[FREETOP] 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 <04> = INX B;
          RNZ
          INR A
          STA CHGLOC
          RET
          (?? 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, Maria 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.

MICROSOFT

the standard for microcomputer software

Some of our latest developments include:

MACRO-80 PACKAGE Our relocatable assembler now has a complete MACRO facility including IF, RPT, REPEAT, local variables and EXIT, listing control and conditional assembly have been greatly enhanced. Another plus - the assembler is now twice as fast as previous versions. The MACRO-80 Package, including Microsoft's Linking Loader and Cross Reference Program, may now be purchased separately from FORTRAN-80. Single copy \$200. Manual \$75. (MACRO-80 is included in FORTRAN-80, Version 3.1.)

MBASIC - NEW RELEASE The new version 5.0 MBASIC includes long variable names, variable length records, dynamic string space allocation, WHILE/UNTIL, corrected files, and ordering with COM-MON. Version 5.0 is fully ANSI compatible. Our MBASIC documentation has been completely rewritten and is significantly improved. Single copy \$330. Manual \$20.

EDIT-80 PACKAGE (CP/M version only) the lowest cost editor on the market. No more searching through files or cryptic commands. The random access, line-oriented editor is similar to those used on large computers like the FCP-41. Also includes P/COM, the file compare utility, which allows comparison of source and binary files. Single copy \$220. Manual \$70.

ANSI '74 COBOL-80 a new available with fully tested ISAM, improved interactive ACCEPT/DISPLAY, COPY and EXTEND. Single copy \$750. Manual \$20.

PREVIEW OF UPCOMING PRODUCTS An 8080/8085 BASIC compiler supporting the same features as our interpreter, the long-awaited 8080/8085 APL interpreter, and a complete set of systems software products for both the 8086 and 28600.

Only one company sets the pace with software for microprocessors.

THAT'S MICROSOFT.

Whether it's BASIC, FORTRAN, or COBOL, the largest-selling microcomputer systems use software by Microsoft.

Radio Shack, Itek, NCR, Apple, Commodore, Citicorp, Eikon, Eikon, Imtek, QED, Science Center, Comshare, ALOS, Zap, Mostek, National, Rockwell, and many others.

And at Microsoft, new things are happening all the time.

All software available at single-copy prices for OEM. Dealer agreement price.

MICROSOFT IS MOVING!

2400 NE Eighth, Suite 809
Bellevue, Washington 98008
206-453-8500

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.

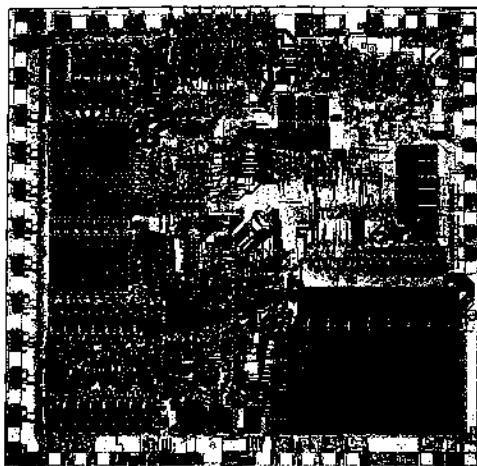
Trading some programming for a disk drive from Shugart, Kildall first attempted to build a drive controller on his own. Lacking the necessary engineering ability, he contacted a friend, John Torode, who agreed to handle the hardware aspects of interfacing the computer and the disk drive while Kildall 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.

GO 16-BIT NOW — WE HAVE MADE IT EASY

8086

8 Mhz. 2-card CPU Set

\$595

WITH 86-DOS[®]

ASSEMBLED, TESTED, GUARANTEED

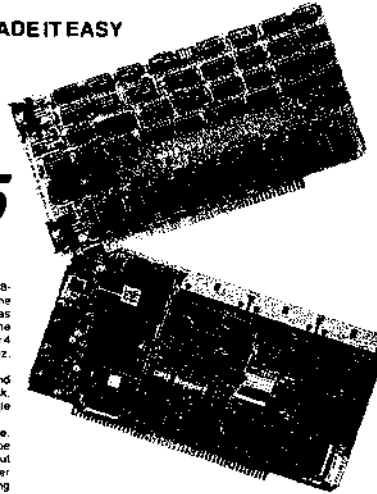
With our 2-card 8086 CPU set you can upgrade your Z80 8-bit S-100 system to run three times as fast by swapping the CPUs. If you use our 16-bit memory, it will run five times as fast. Up to 64K of your static 8-bit memory may be used in the 8086's 1-megabyte addressing range. A switch allows either 4 or 8 Mhz operation. Memory access requirements at 4 Mhz, extended 500 nsec.

The EPROM monitor allows you to display, alter, and search memory, do inputs and outputs, and boot your disk. Debugging aids include register display and change, single stepping, and execute with breakpoints.

The set includes a serial port with programmable baud rate, four independent programmable 16-bit timers (two may be combined for a time-of-day clock), a parallel in and parallel out port, and an interrupt controller with 15 inputs. External power may be applied to the timers to maintain the clock during system power-off time. Total power: 2 amps at +8V, less than 100 ma. at +16V and at -16V.

86-DOS[™], our \$195 8086 single user disk operating system, is provided without additional charge. It allows functions such as console I/O of characters and strings, and random or sequential reading and writing to named disk files. While it has a different format from CP/M, it performs similar calls plus some extensions (CP/M is a registered trademark of Digital Research Corporation). Its construction allows relatively easy configuration of I/O to different hardware. Directly supported are the Terabeek and Cromemco disk controllers.

The 86-DOS[™] package includes an 8086 resident assembler, a Z80 to 8086 source code translator, a utility to reape files written in CP/M and convert them to the 86-DOS format, a line editor, and disk maintenance utilities. Of significance to Z80 users is the ability of the translator to accept Z80 source



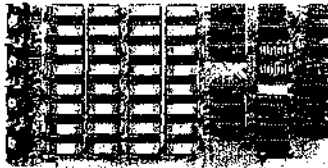
code written for CP/M. Translate this to 8086 source code, assemble the source code, and then run the program on the 8086 processor under 86-DOS. This allows the conversion of any Z80 program, for which source code is available, to run on the much higher performance 8086.

BASIC-86 by Microsoft is available for the 8086 at \$350. Several firms are working on application programs. Call for current software status.

All software licensed for use on a single computer only. Non-disclosure agreements required. Shipping from stock to one week. Bank cards, personal checks, CODs okay. There is a 10-day return privilege. All boards are guaranteed one year — both parts and labor. Shipped prepaid by air in US and Canada. Foreign purchases must be prepaid in US funds. Also add \$10 per board for overseas air shipment.

8/16 16-BIT MEMORY

This board was designed for the 1980s. It is configured as 16K by 8 bits when accessed by an 8-bit processor and configured 8K by 16 bits when used with a 16-bit processor. The configuration switching is automatic and is done by the card sampling the "strobe request" signal sent out by all S-100 IEEE 16-bit CPU boards. The card has all the high noise immunity features of our well-known PLUS-RAM cards as well as "extended addressing". Extended addressing is a replacement for bank select. It makes use of a total of 24 address lines to give a directly addressable range of over 16 megabytes. (For older systems, a switch will cause the card to ignore the top 8 address lines.) This card ensures that your memory board purchase will not soon be obsolete. It is guaranteed to run without wait states with our 8086 CPU set using an 8 Mhz clock. Shipped from stock. Prices: 1-4, \$280; 5-9, \$260; 10-up, \$240.



Seattle Computer Products, Inc.
1114 Industry Drive, Seattle, WA 98108
(206) 575-1830

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.

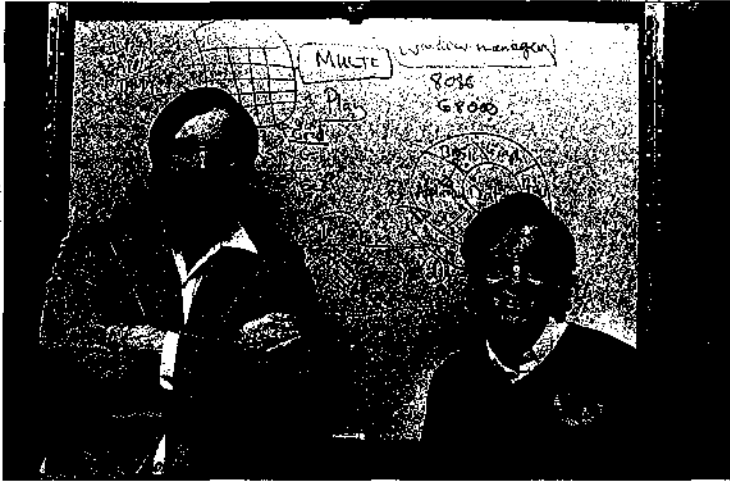
In October 1980, with 86-DOS in hand, Microsoft submitted another proposal to IBM. This time the plan included both an operating system and the languages for the new computer. Time was short and the boundaries between the languages and the operating system were unclear, so Microsoft explained that it 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.



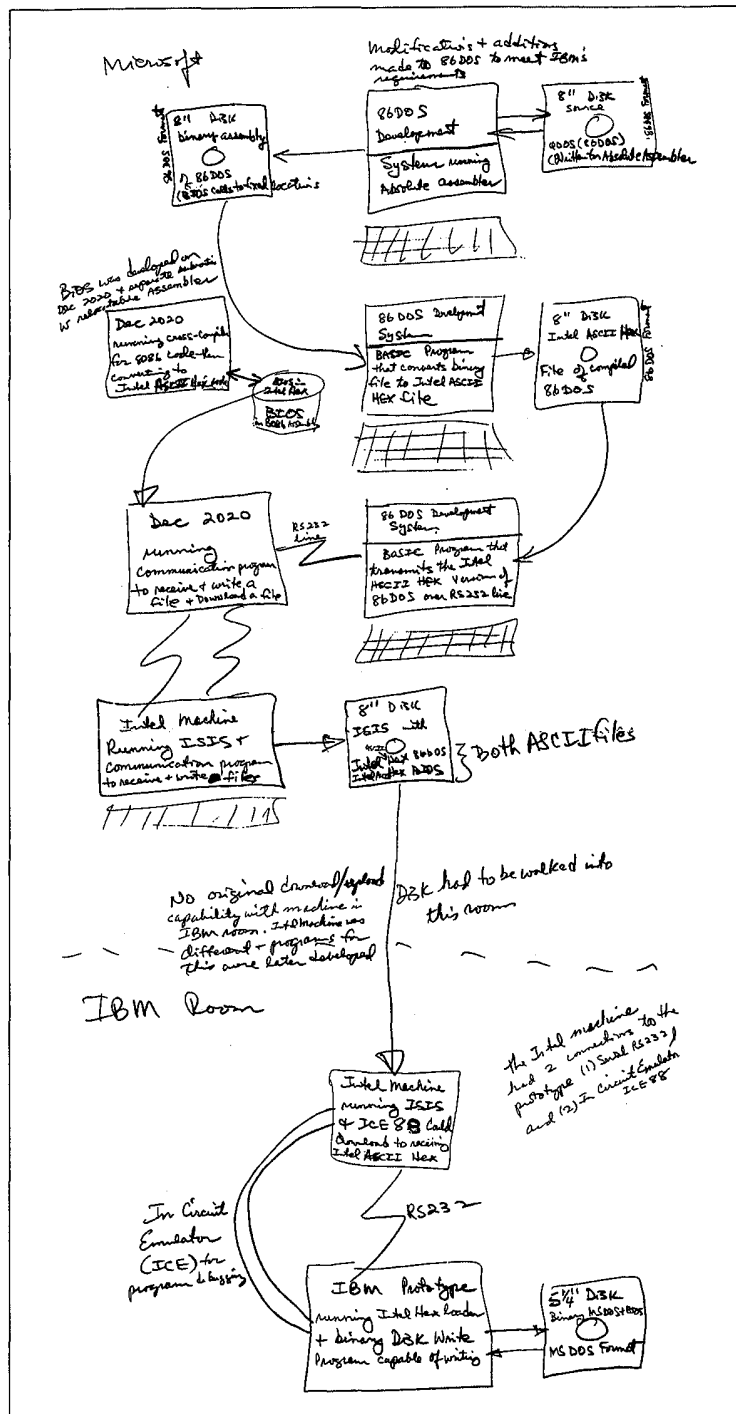
*Paul Allen and
Bill Gates (1982).*

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.

DOS Changes & Fixes

- 4/20 ~~13~~ Single drive support, i.e. six copy + prompt if same disk
- 4/20 ~~13~~ Modify "Format" to do a prompt to allow user to replace disk if formatting a single drive system
- 4/2 ~~13~~ Move origin of BIOS to 60:0 and origin of 86 DOS to C0:0. 86 DOS moved to 100:0 due to #21.
- 4/2 15. Change BOOT program to load the BIOS & DOS into the correct segments and further to locate and identify "if" these routines are on the disk. ~~that is part of the changes necessary to effect data diskette.~~ BOOT should print an error message indicating on what available if ~~data disks have the b~~

- 4/2 ~~16~~ ~~of sectors~~
- 4/2 ~~17~~ ~~Full disked~~ see #21
- 4/2 ~~18~~ ~~make sure~~ instruction
- 4/2 ~~19~~ ~~Modify the~~ ~~define m~~ ~~the ms acc~~
- 4/2 ~~20~~ ~~Set final sin~~ ~~support of a~~
- 4/2 ~~21~~ ~~Modify~~ ~~CHK~~ ~~that cross~~ ~~it to the~~

DOS Changes & Fixes

- 4/2 ~~22~~ Move 'date' to known location → 50:2
50:3

```

format 50:2 76543210
        50:3 77777777
    
```

new register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Requires mode to 86DOS to ~~main~~ address date correctly & will take out 86DOS request for date & move to COMMANDS

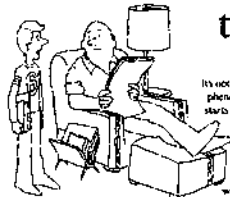
- 4/2 2. modify COMMAND to search for AUTOEXEC.BAT & if found do a submit on this file. If AUTOEXEC.BAT not found print banner and request date
- 4/2 ~~3~~ Fix DEBUG to do disassembly correctly. Strange problem. This works correctly on the CMC machine.
- 4/2 ~~4~~ ~~Modify~~ ~~DEBUG~~ to frame its output so that it's readable on both 40X25 & 80X25 screens
- 4/2 ~~5~~ ~~Modify~~ ~~FORMAT~~ to allocate detected bad tracks to file BADTRK.
- 4/2 ~~6~~ ~~Fix~~ ~~problem~~ with ZIBO RUN SPACE where a random read request (function 39) bombs.
- 4/2 ~~7~~ ~~check~~ ~~out~~ RS-232 support in the BIOS
- 4/2 ~~8~~ ~~check~~ ~~out~~ SUBDIRT command
- 4/2 ~~9~~ ~~check~~ ~~out~~ EDIEN edit of file larger than available memory. depends on # 21
- 4/2 ~~10~~ ~~check~~ ~~out~~ why F9 function key does not work correctly in the DOS
- 4/2 11. Indication from CHKDSK of available disk entries.

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 car. Or when your daughter

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 in or 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. Unusually, 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 in 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.

to use your metal exosquelet. 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 compare 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

The IBM Personal Computer and me.



A system and the IBM Personal Computer. The IBM Personal Computer is a system that includes a monitor, keyboard, and system unit. It is designed for home and office use. The IBM Personal Computer is a system that includes a monitor, keyboard, and system unit. It is designed for home and office use.

IBM Personal Computer. The IBM Personal Computer is a system that includes a monitor, keyboard, and system unit. It is designed for home and office use. The IBM Personal Computer is a system that includes a monitor, keyboard, and system unit. It is designed for home and office use.

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

LINE COMPANY	VALUE OF SALES
IBM	21.5
APPLE	11.0
COMMODORE	11.0
HP	10.0
INTEGRATED	10.0
CONQUEST	10.0
PERIPHERALS	10.0
DATA	10.0
SOFTWARE	10.0
SYSTEMS	10.0
PERIPHERALS	10.0
SOFTWARE	10.0
SYSTEMS	10.0
PERIPHERALS	10.0
SOFTWARE	10.0
SYSTEMS	10.0

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

By GEORGE ARONSON
NEW YORK—International Business Machines Corp. has made its bold entry into the personal computer market, and experts believe the company 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

range from \$1,995 for the base system to \$5,995 for the more powerful "big" model. These new computers are designed to be used in homes, schools and offices. They are also designed to be used in homes, schools and offices. They are also designed to be used in homes, schools and offices.

far greater, equivalent to more than 1,000 typewriters. 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 output on their video screens in greater detail.

IBM acknowledges that a fully loaded computer will cost \$4,000 or more. For basic 1,995 machine comes with 16,000 characters

of memory. The new IBM computers are designed to be used in homes, schools and offices. They are also designed to be used in homes, schools and offices. They are also designed to be used in homes, schools and offices.

InfoWorld

News For Microcomputer Users

IBM Announces New Microcomputer System

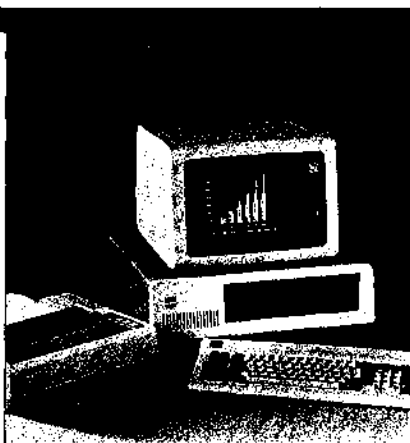
It's Official; One surprise

By Peter Dinklage, IBM Staff
NEW YORK—IBM announced yesterday that it has entered the personal computer market. The company's new Personal Computer System, which is designed to be used in homes, schools and offices, is expected to be a major success.

IBM's new Personal Computer System is designed to be used in homes, schools and offices. It is expected to be a major success. The company's new Personal Computer System, which is designed to be used in homes, schools and offices, is expected to be a major success.

OUTLOOK

IBM really gets personal.



PERSONAL COMPUTERS

PERSONAL COMPUTER FROM IBM

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

With characteristics that resonating features spanned the summer's most popular guessing game for the industry by introducing its Personal Computer. Highly comparable to offerings from arch-rival Apple and Radio Shack, the machine represents several new tactics 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 \$1,995 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, she 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 broiled probes and shone lights as the machine's Waldorf Astoria introduction, declines to say how many personnel have been dedicated to the national remarketing 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 we 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 Packline Software operation, and Information Unlimited's EasyWriter word processing system. Although that wouldn't say, more independently developed packages are certain to be offered for the computer as well as packages

recently 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 you'd expect.

The machine is impressive. Its starting price is a mere \$1,995. For that price the buyer gets the 85-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 external

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 professional 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

Continued on page 77

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

MICROSOFT QUARTERLY

The policy is especially advantageous when a large number of programs is developed 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 3.2 and earlier, too.

The 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 COMAN with COMMON, and the implementation of the runtime module make BASCOM a much more flexible and powerful tool for the application programmer.

BASCOM 3.5, the available now for CP/M systems, including the Apple II with the Microsoft Software Development System, 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 opens up the arena of the industry that have been



...on the verge of changing for the past 10 months. First, the industry is reacting 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 when set in an enhanced environment. For example, we've taken an advantage of the expanded addressing in our MS-DOS, a kernel for Pascal or FORTRAN programs that run up to a megabyte in size in 80K of memory. The Microsoft 8086 BASIC interpreter can execute a 64K program, almost double the size available on an 8-bit runtime. Applications programs can be more sophisticated in their features, human engineering factors, and solving problems that involve large 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 require much faster. The speed of the graphics routines in IBMASIC makes it very easy to construct a graphics application with the machine language.

With the IBM announcement of the Personal Computer, it looks as though the industry is finally opening 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 "next pin" of Microsoft is now 16-bit products for the 8086/8088 is our complete, reliable operating system, MS-DOS. MS-DOS is the primary operating system on the IBM Personal Computer. We've maintained compatibility with existing CP/M 2.2 operating system calls, so it's a straight forward process to convert 8086 and 286 programs to run under MS-DOS. MS-DOS also provides a future upgrade path to the 80286 multi-user, multi-tasking 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 its products. The United States Government, the largest user of computer equipment and software in the world, has developed tests for compliance with an implementation of standards for computers. Testing of computers 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-maintenance implementation of the 1974 ANSI standard for COBOL.

Why is Microsoft concerned about standards, and why did we submit Microsoft COBOL for validation? Mike Or, COBOL product manager, offered the following reasons (continued on page 5):

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.

<p>Package Contents</p> <p>1 diskette, with the following files: COMMAND.COM MSDOS.COM EDLIN.COM DEBUG.COM FILCOM.COM</p> <p>1 MS-DOS Disk Operating System Manual</p> <p>System Requirements</p> <p>The MS-DOS Operating System requires 8K bytes of memory.</p>	<p>Contents</p> <p>Introduction</p> <p>Features and Benefits of MS-DOS Using This Manual Syntax Notation MS-DOS Structure and Characteristics</p> <p>Chapter 1 General MS-DOS Commands</p> <p>1.1 Control Function Characters</p> <p>1.2 Special Editing Commands</p> <p>1.3 Disk Errors</p> <p>Chapter 2 COMMAND.COM</p> <p>2.1 Prompt</p> <p>2.2 Filenames</p> <p>2.3 Commands</p> <p>2.3.1 Internal Commands</p> <p>2.3.2 External Commands</p> <p>Chapter 3 EDLIN</p> <p>3.1 Invoking EDLIN</p> <p>3.2 Commands</p> <p>3.2.1 Command Parameters</p> <p>3.2.2 Interline Commands</p> <p>3.3 Error Messages</p> <p>Chapter 4 DEBUG</p> <p>4.1 Invoking DEBUG</p> <p>4.2 Commands</p> <p>4.2.1 Command Parameters</p> <p>4.2.2 Command Descriptions</p> <p>4.3 Error Messages</p> <p>Chapter 5 FILCOM</p> <p>5.1 Invoking FILCOM</p> <p>5.2 Commands</p> <p>5.2.1 Filenames</p> <p>5.2.2 Switches</p> <p>5.3 Examples</p> <p>Chapter 6 Instructions for Single Disk Drive Users</p>
--	---

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 Structures**
The format eliminates the need for "searches," minimizing access to the directory track, and provides for duplicate directory information and verify after write.
- **No Need to Log In/Exit**
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 removable disks.
- **No Physical File Disk Size Limitation**
Limits 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-16-Byte Physical Sectors**
One does not have to worry about different physical sector sizes when writing a BIOS.
- **Time/Date Stamp**
This alleviates, for instance, the need to recompile a file if its time on the relocatable file is more recent than on the source file.
- **Litko/Bit 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, Litko/Bit 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 increase the 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 transparency 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 final 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 IBM PC 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 (specifies 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 COMMAIND 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 subette, 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, it could 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 16 bytes. This alleviates the need for forcing Control-Z's or 0's at the end of a file.

The Future of MS-DOS

Microsoft plans to enhance MS-DOS. The additional addressing space of the 8088 processor makes multitasking 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, menu support, multi-user and hard disk support, and networking.



Microsoft, Inc.
18800 E. Eighth, Suite 819
Bellevue, WA 98004
206-465-8000 Telex 328945

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

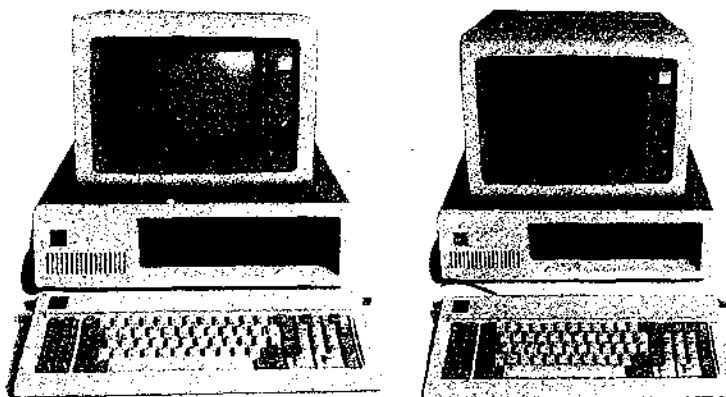
At IBM's request, version 2.0 of MS-DOS also possessed the undocumented ability to perform rudimentary background processing — an interim solution to a growing awareness of the potentials of multitasking.

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

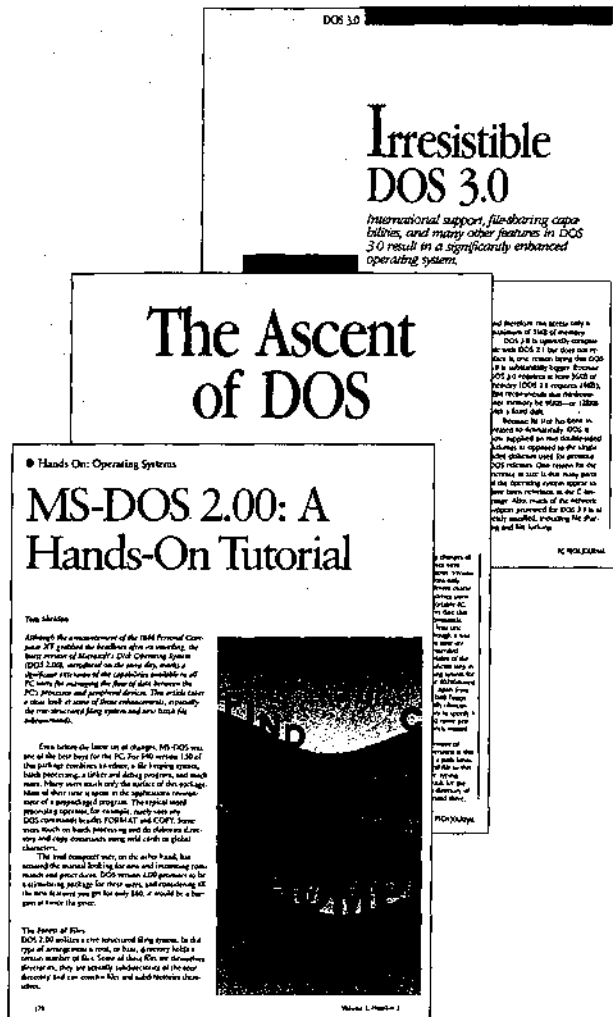
When version 2.0 was released, IBM was already planning to introduce its PCjr. The PCjr would have the ability to run programs from ROM cartridges and, in addition to using half-height 5¼-inch drives, would employ a slightly different disk-controller architecture. Because of these differences from the 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.



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 KARAI_RYU
ディレクトリは A:\BIN
CHKDSK  EXE  COPY2  COM  ASSIGN  COM  ATTRIB  EXE  BACKUP  EXE
FC       EXE  FIND   EXE  COPYA   COM  DISKCOPY COM  MOUSE   SYS
MORE    COM  SPEED  COM  FORMAT  EXE  KEY      COM  LABEL   EXE
                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 interface 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 request 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 program 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 version 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."

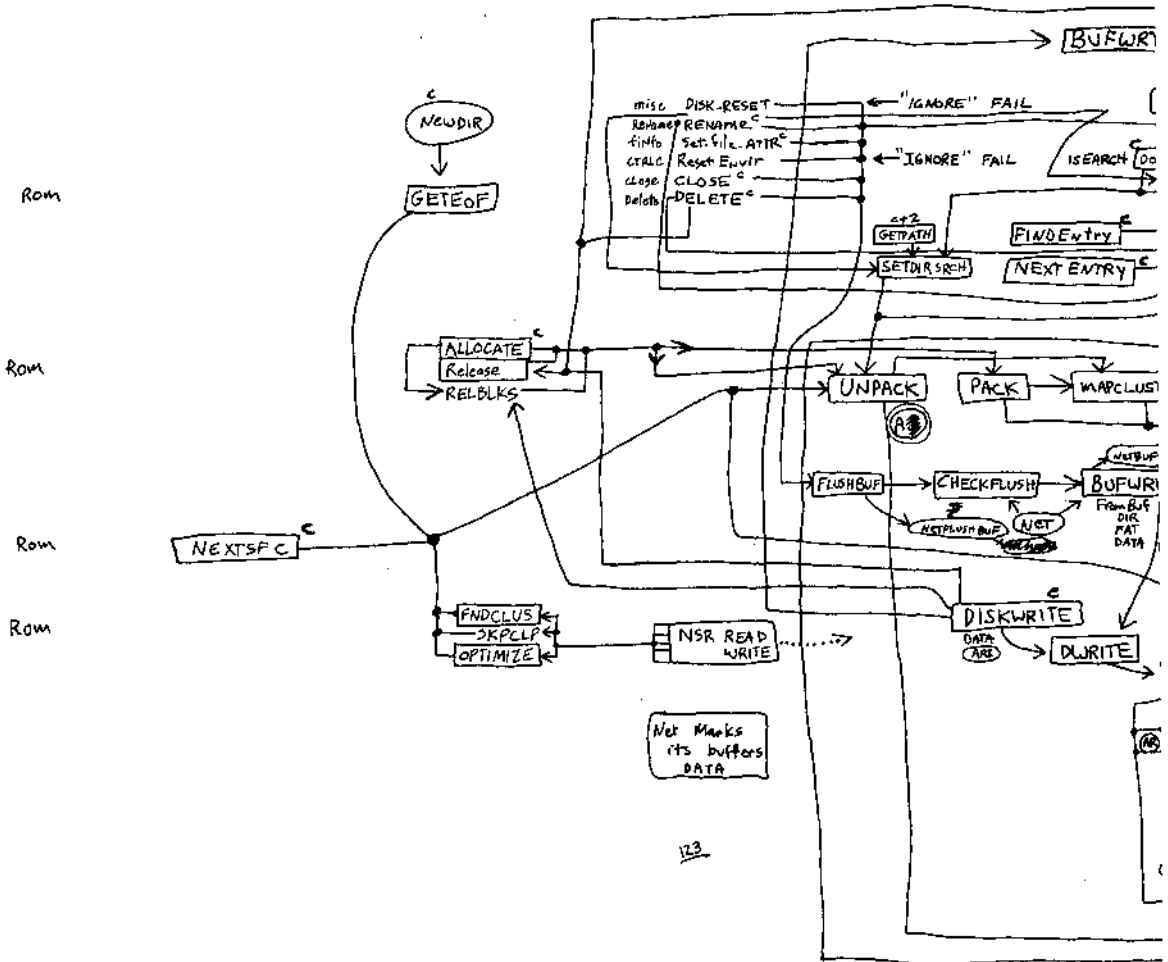
In fact, the great variability in programs and programming approaches that MS-DOS supported eventually proved to be one of the biggest obstacles to the development of a sophisticated networking system and, in the longer term, to the addition of true multitasking.

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.

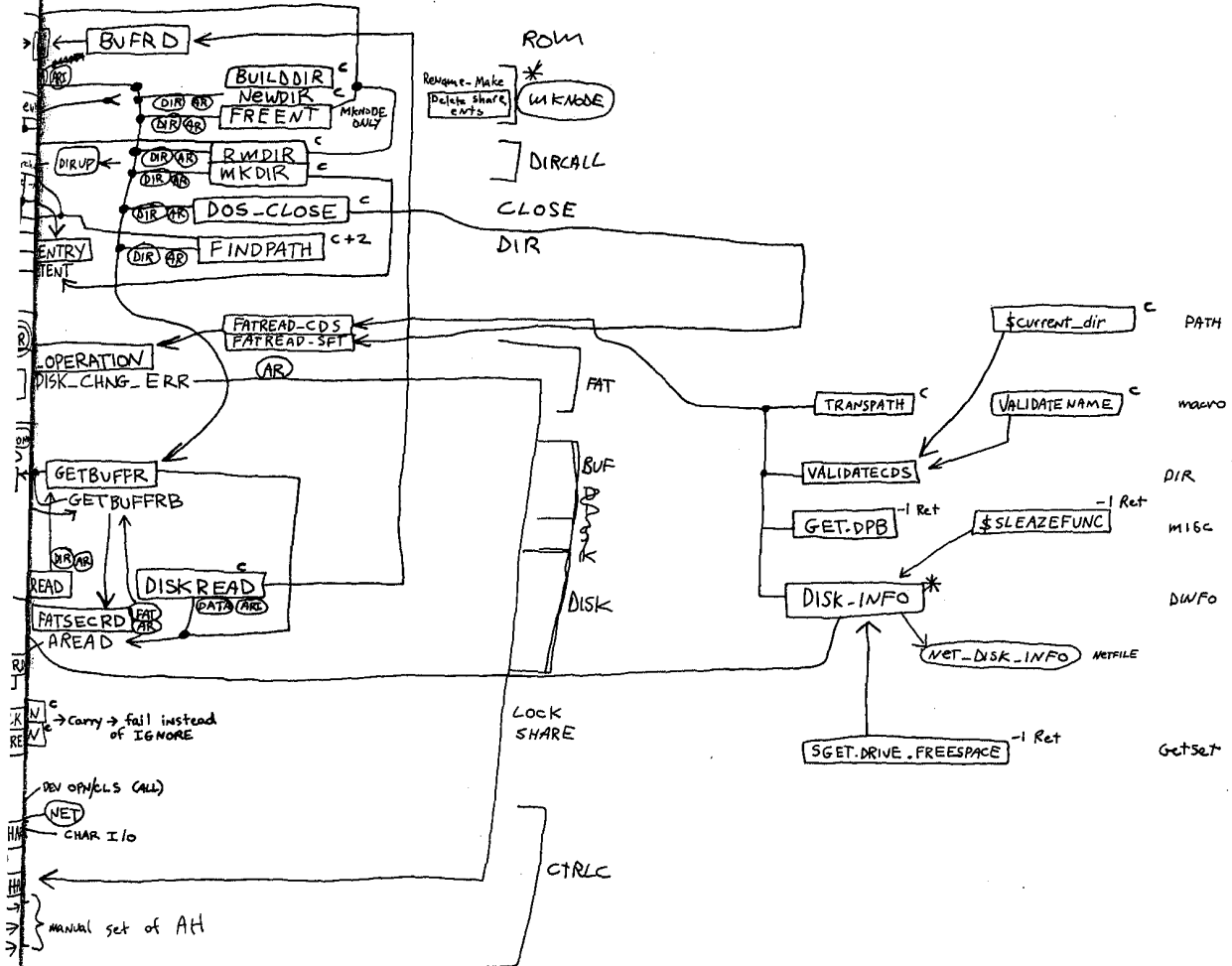
Version 3.0

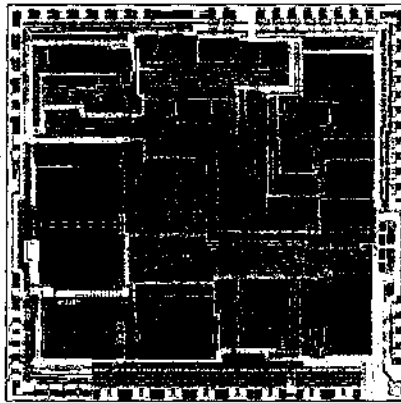
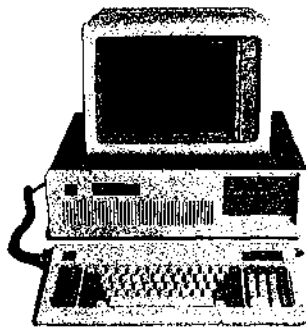
All told, the work on version 3.0 of MS-DOS proved to be long and difficult. For a year and a half, Microsoft grappled with problems of software incompatibility, remote file management, and logical device independence at the network level. Even so, when IBM was ready to announce its new Personal Computer 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.



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 microprocessor, 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 networking, 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 Interconnect 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 approach 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

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 sub-multiple 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 specialized 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 initialization 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 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.

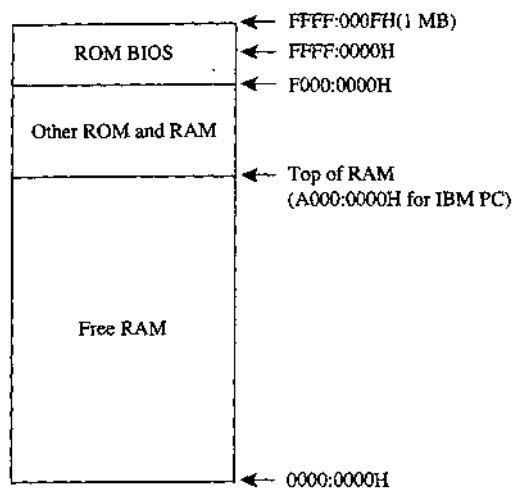


Figure 2-1. Memory layout at startup.

The initialization routine in the ROM BIOS—the POST procedure—typically determines 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 operation. First, it initializes the interrupt vector table at the beginning of memory and any interrupt 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 memory (starting at address 0000:0400H) is used for table storage by various ROM BIOS routines (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 signature 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

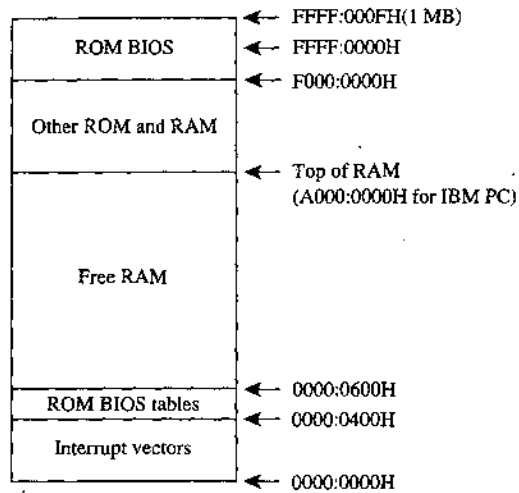


Figure 2-2. The interrupt vector table and the ROM BIOS table.

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

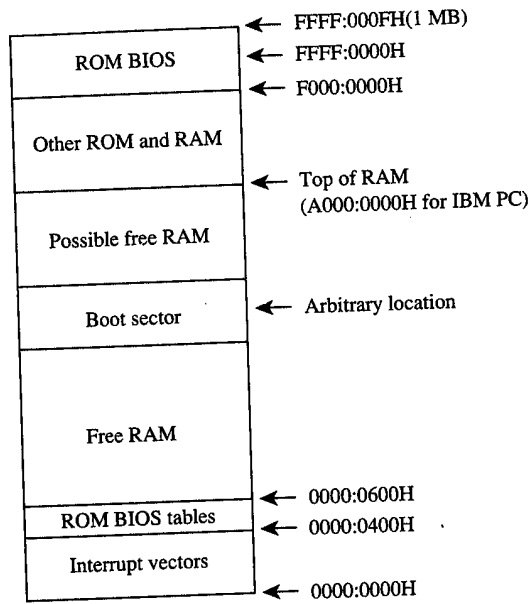


Figure 2-3. A loaded boot sector.

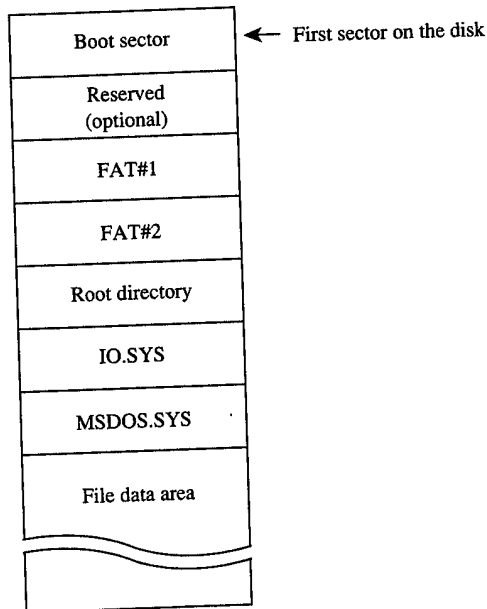


Figure 2-4. Boot-disk configuration.

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.

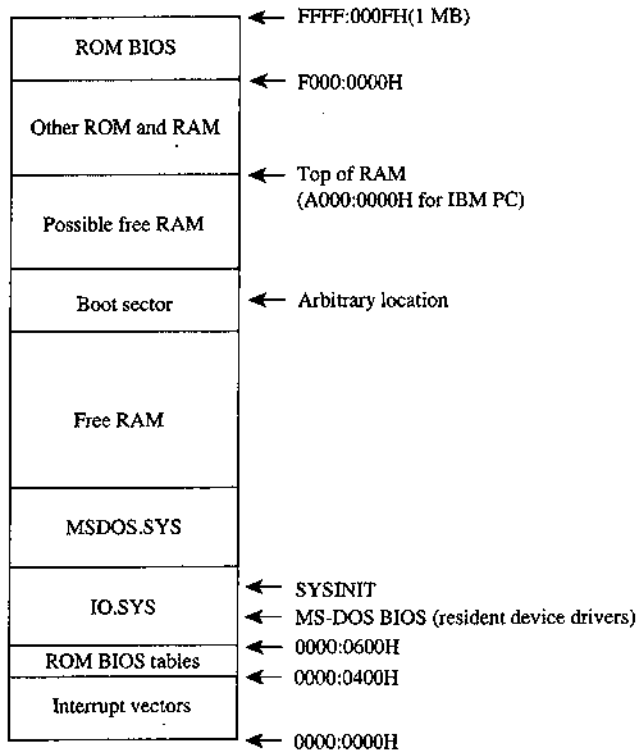


Figure 2-5. IO.SYS and MSDOS.SYS loaded.

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.

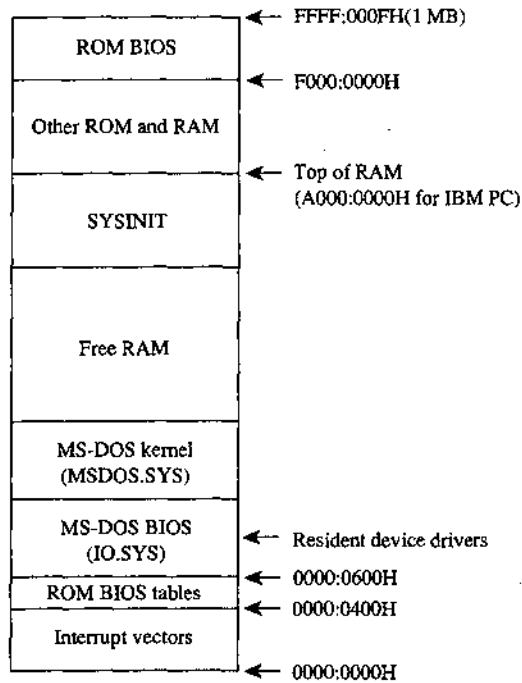


Figure 2-6. SYSINIT and MSDOS.SYS relocated.

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.

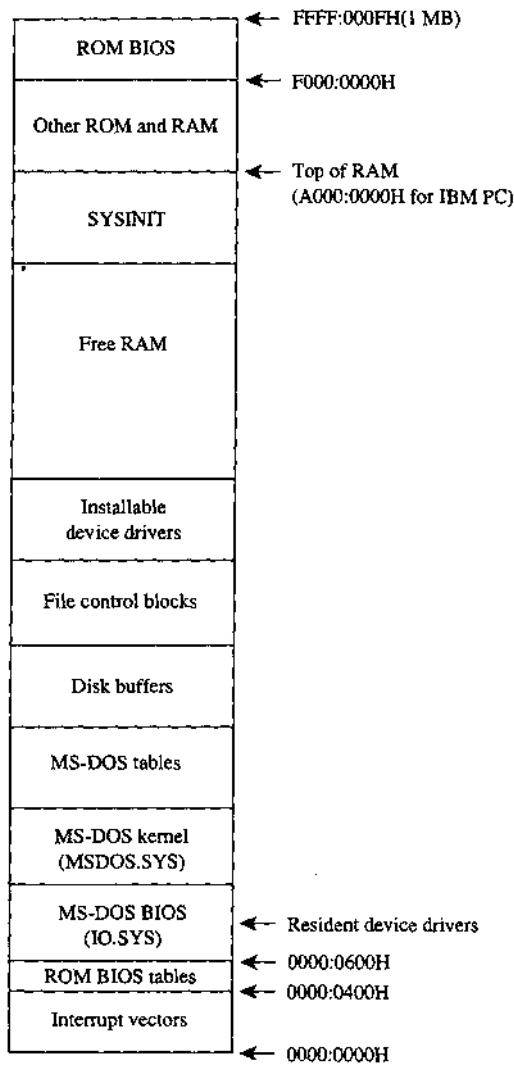


Figure 2-7. Tables allocated and installable device drivers loaded.

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.

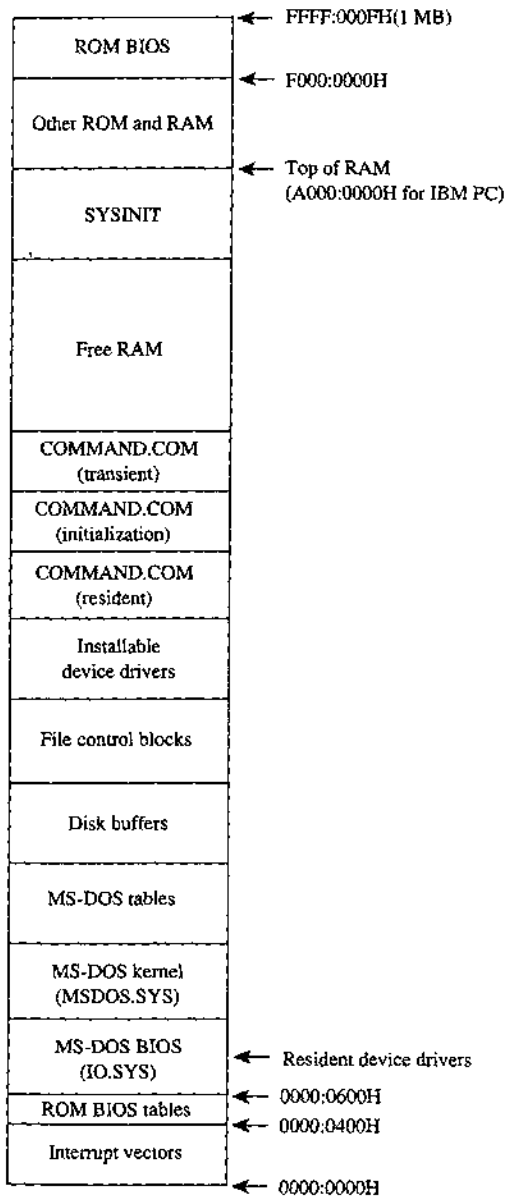


Figure 2-8. COMMAND.COM loaded.

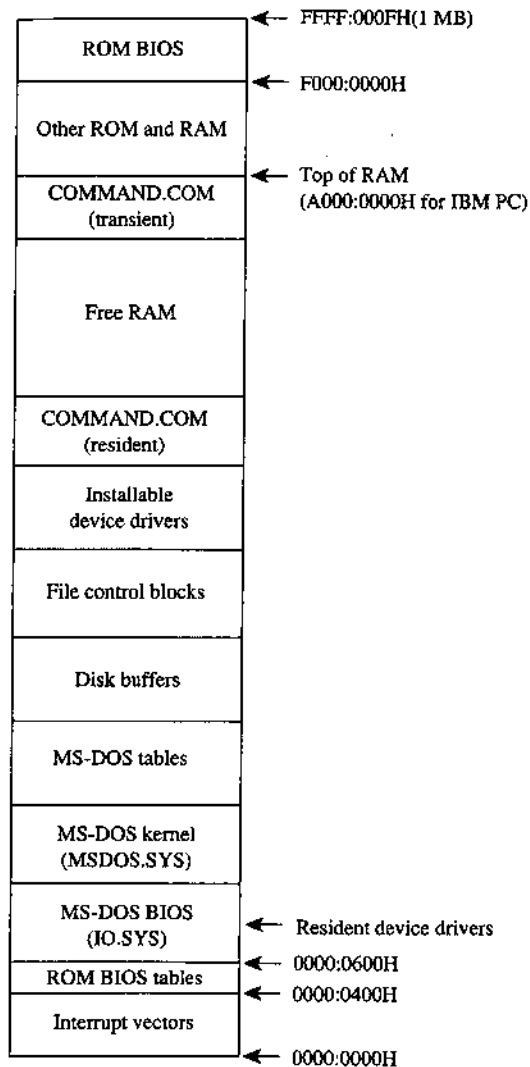


Figure 2-9. COMMAND.COM after relocation.

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.

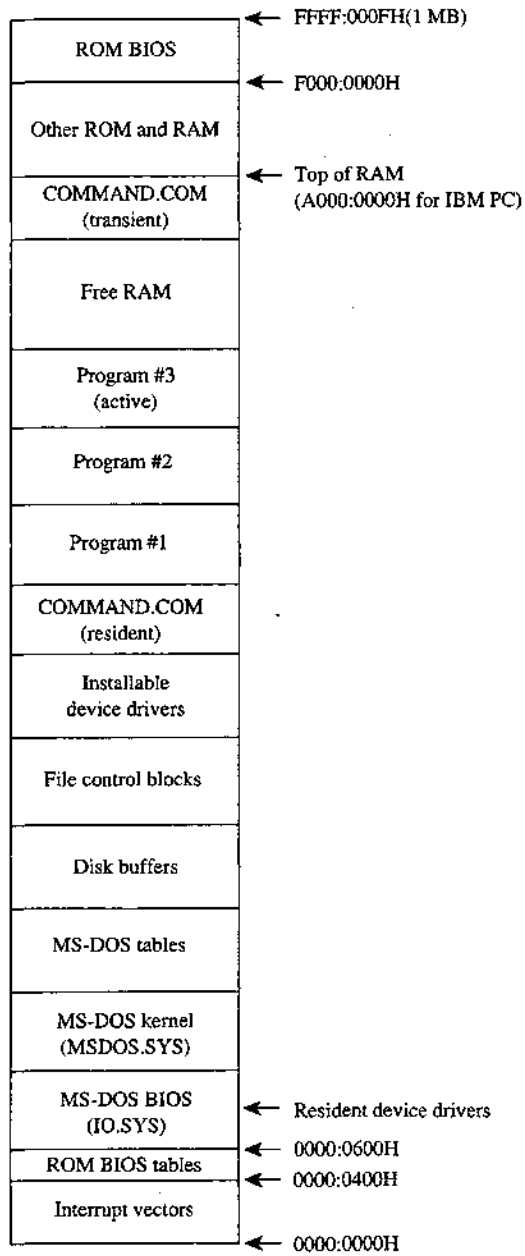


Figure 2-10. Multiple programs loaded.

```

; 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 command 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).

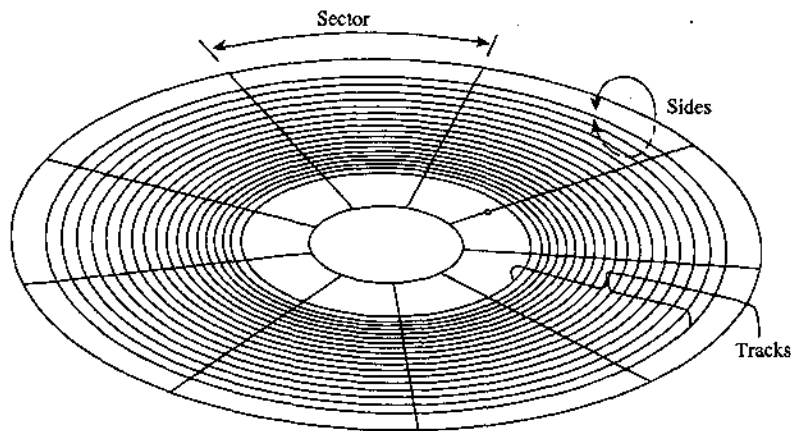


Figure 3-1. The physical layout of a CAV 9-sector, 5.25-inch floppy disk.

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).

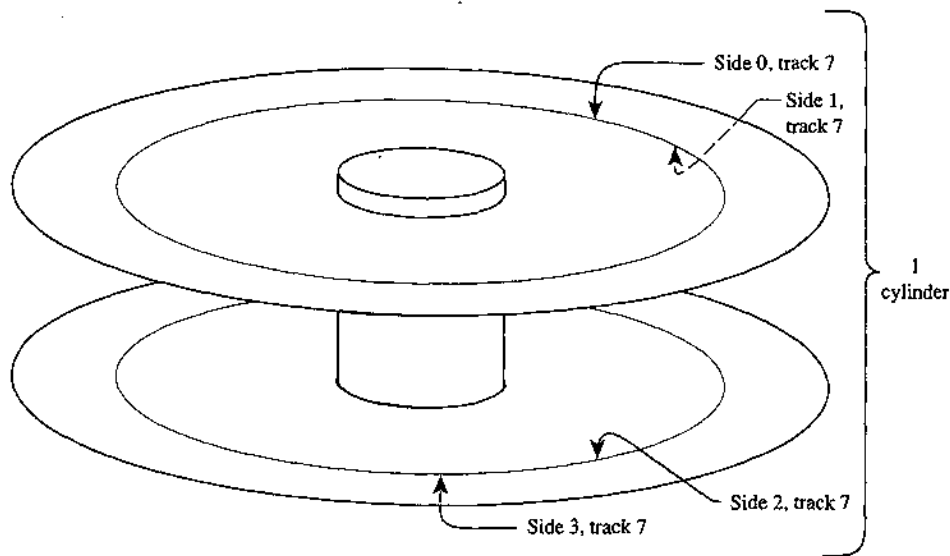


Figure 3-2. Tracks and cylinders on a fixed-disk system.

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

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.

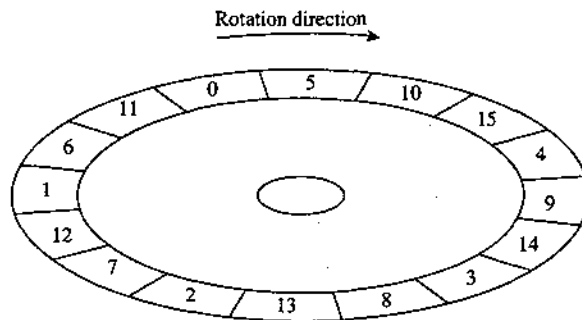


Figure 3-3. A 3:1 interleave.

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.)

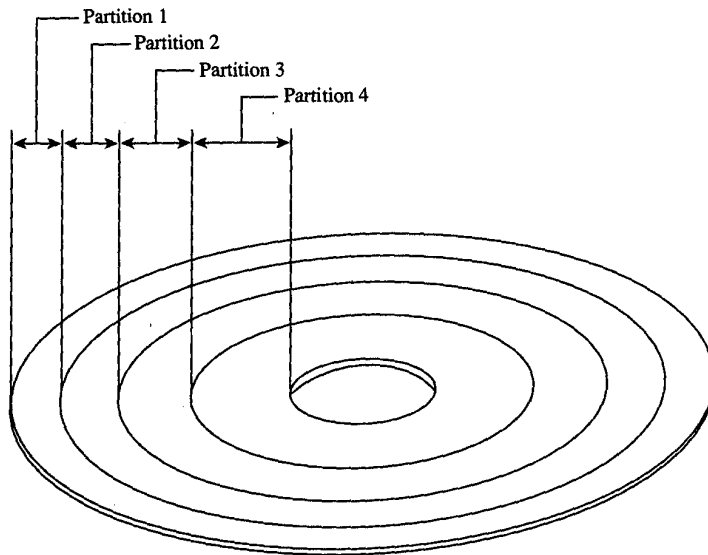


Figure 3-4. A partitioned disk.

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.

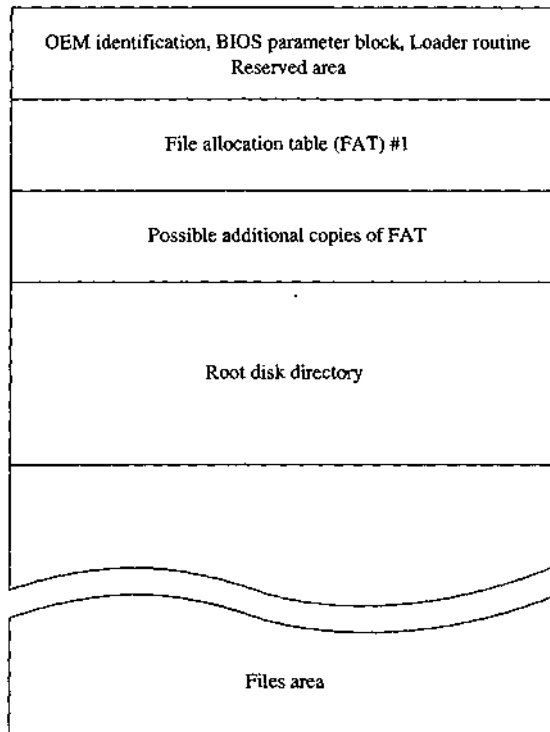


Figure 3-5. The MS-DOS file system.

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.

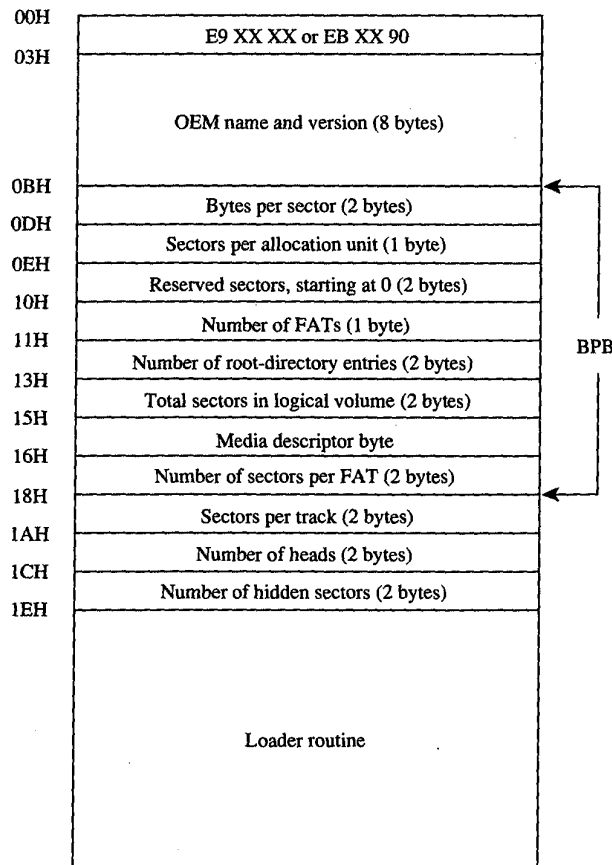


Figure 3-6. Map of the boot sector of an MS-DOS disk. Bytes 0BH through 17H are the BIOS parameter block (BPB).

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

Part A: Structure of MS-DOS

```

      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 80..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 .....
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.

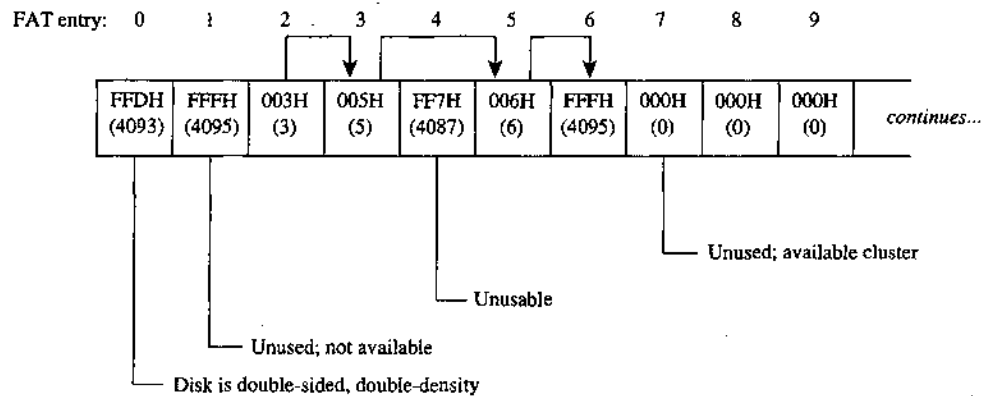


Figure 3-8. Space allocation in the FAT for a typical MS-DOS disk.

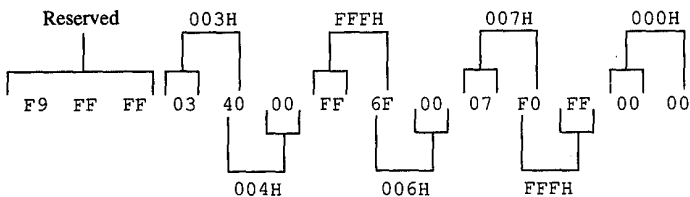
Free FAT entries contain a link value of zero; a link value of 1 is never used. Thus, the first allocatable link number, associated with the first available cluster in the file data area, is 2, which is the number assigned to the first *physical* cluster in the file data area. Figure 3-9 shows the relationship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

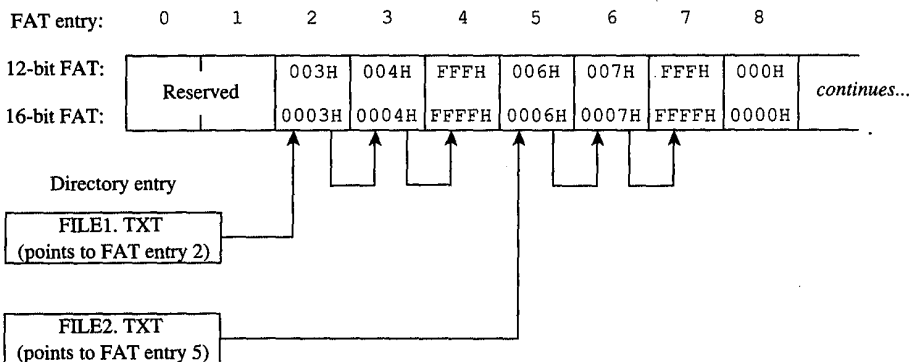
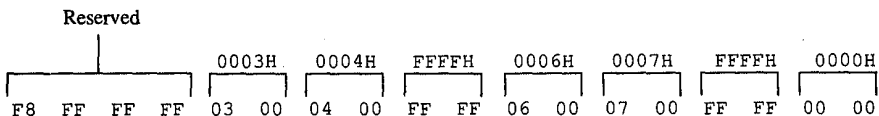
Special considerations

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:



File data area	Corresponding FAT entry
FILE1. TXT	2
FILE1. TXT	3
FILE1. TXT	4
FILE2. TXT	5
FILE2. TXT	6
FILE2. TXT	7
Unused (available)	8

Figure 3-9. Correspondence between the FAT and the file data area.

Part A: Structure of MS-DOS

```
; ---- 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	00YS....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	00AIR...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	00AIR...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	00w}8.#.....	
0080	4C	41	42	45	4C	20	20	20	20	20	08	00	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	10	00	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	00F.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	00Ems.#./...	
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	00ae.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	00AIR.'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	002..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...f...	
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.

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.

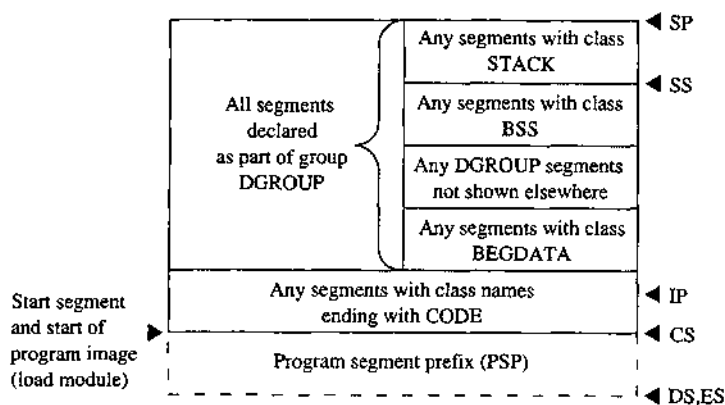


Figure 4-1. The .EXE program: memory map diagram with register pointers.

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

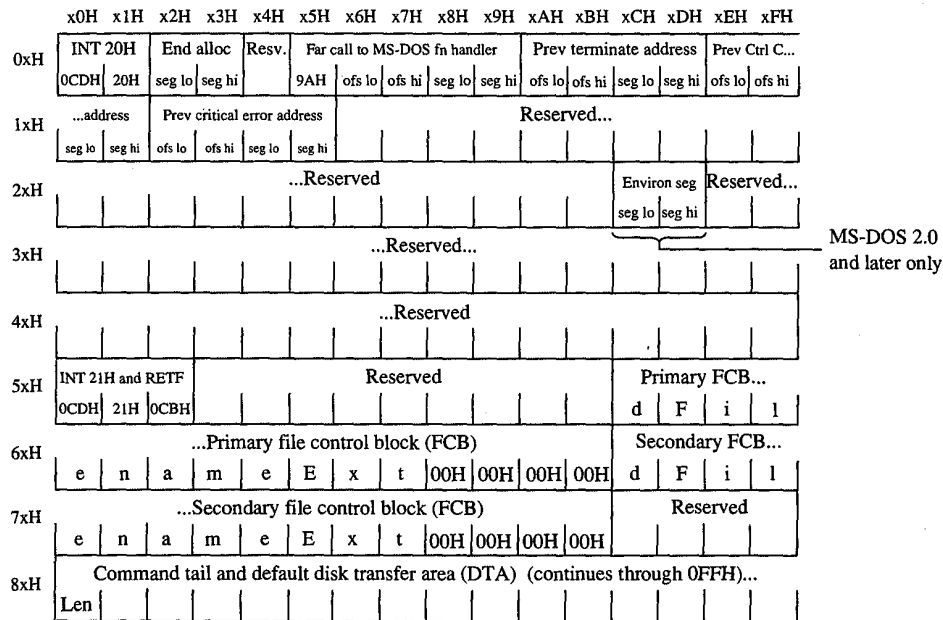


Figure 4-2. The program segment prefix (PSP).

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.

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 program, on the other hand, executes with CS pointing to various code segments of the program, 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 Program 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

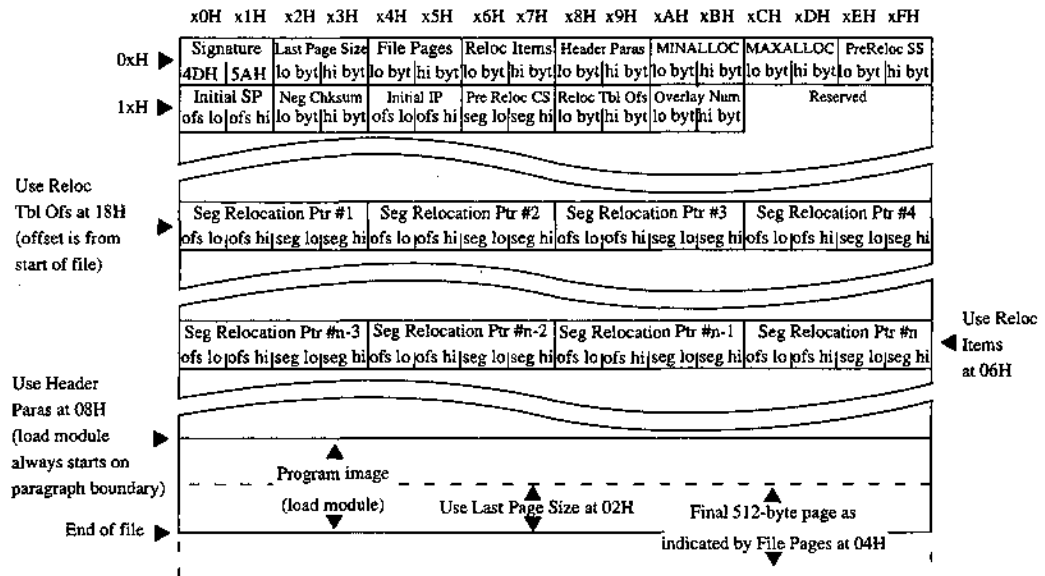


Figure 4-5. Structure of a .EXE file.

(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 $C8CH + D8EH + 10BAH + 00B4H = 1B288H$. (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

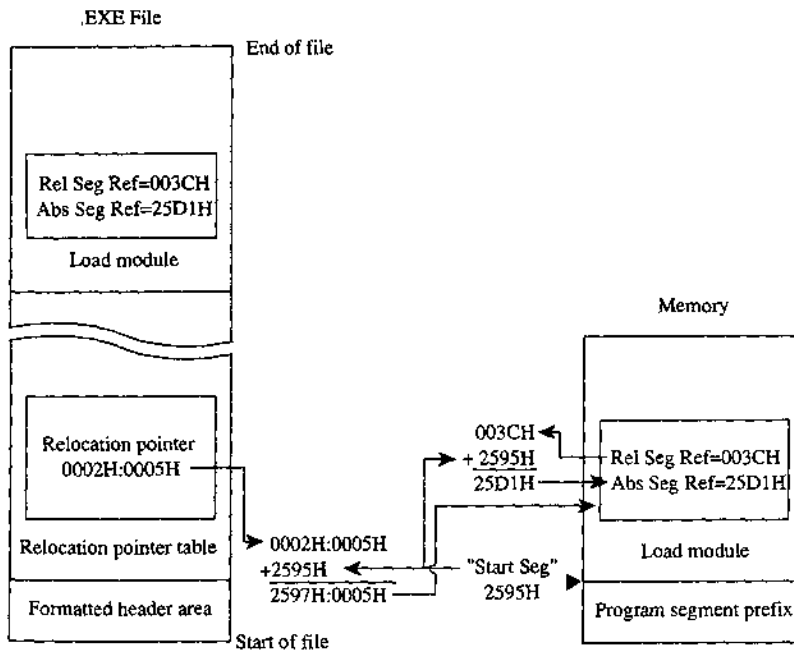


Figure 4-6. The .EXE file relocation procedure.

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.

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:

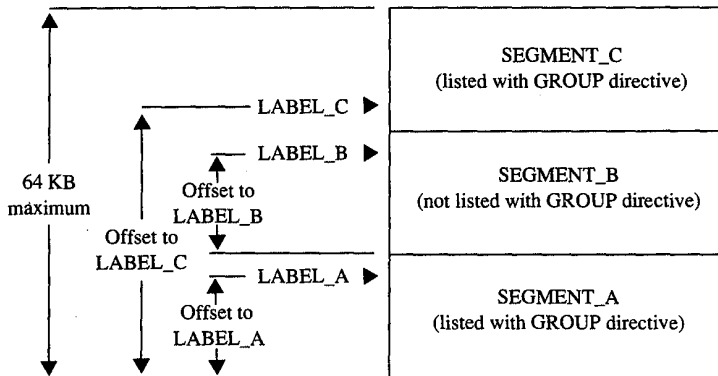


Figure 4-7. Noncontiguous segments in the same GROUP.

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.

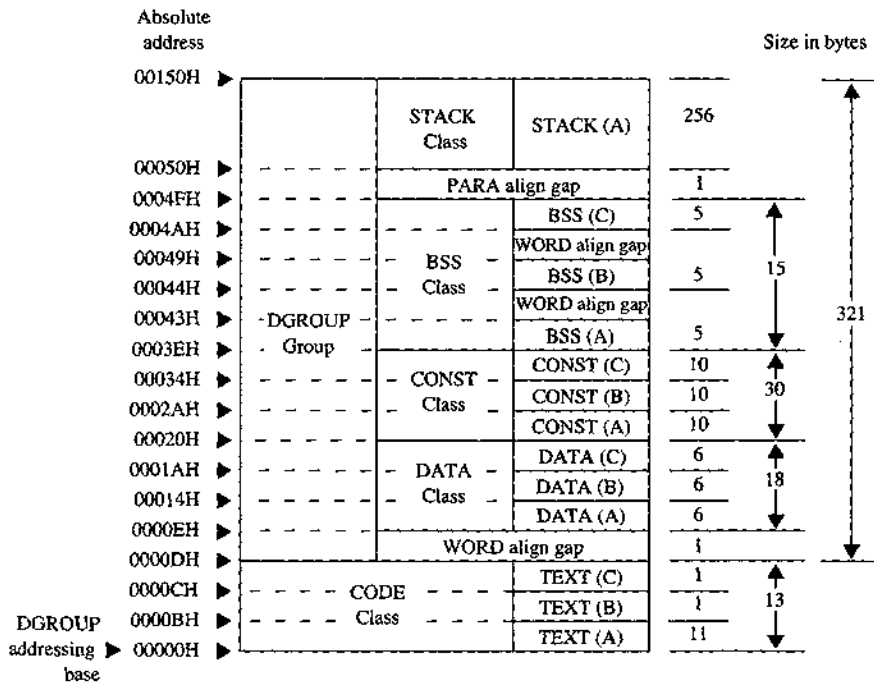


Figure 4-12. Structure of the sample .EXE program.

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.

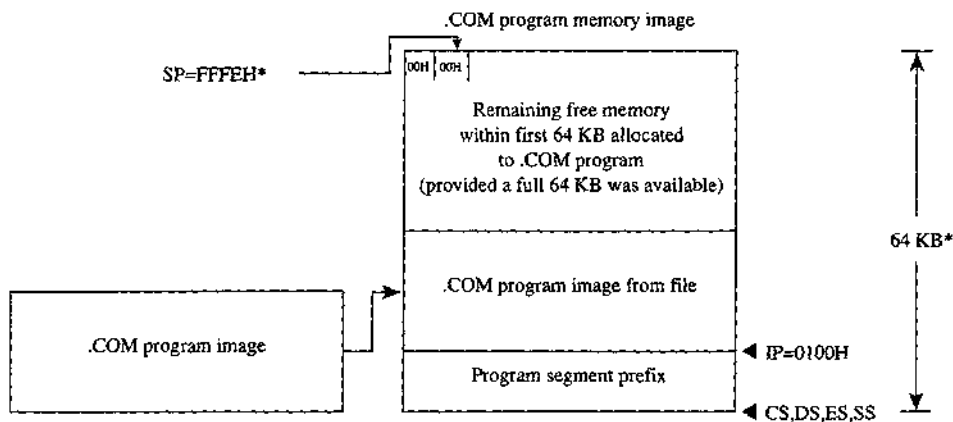
Giving control to the .COM program

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 corresponds 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* permitted 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 (<i>stdin</i>)	0	CON
Standard output (<i>stdout</i>)	1	CON
Standard error (<i>stderr</i>)	2	CON
Standard auxiliary (<i>stdaux</i>)	3	AUX
Standard printer (<i>stdprn</i>)	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
< <i>file</i>	Causes read operations from standard input to obtain data from <i>file</i> .
> <i>file</i>	Causes data written to standard output to be placed in <i>file</i> .
>> <i>file</i>	Causes data written to standard output to be appended to <i>file</i> .
<i>p1</i> <i>p2</i>	Causes data written to standard output by program <i>p1</i> to appear as the standard input of program <i>p2</i> .

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,08h           ; 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 misnamed 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.

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-adaptor 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 ENVIRONMENT: 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 terminology. 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 specific 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 COMx 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    1000000b ; divisor latch access bit
104 : SetBrk  Equ    0100000b ; send break control bit
105 : StkPar  Equ    0010000b ; stick parity control bit
106 : EvnPar  Equ    0001000b ; even parity bit
107 : GenPar  Equ    0000100b ; generate parity bit
108 : Xstop   Equ    0000010b ; 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    0100000b ; xmt SR empty
115 : xhre    Equ    0010000b ; 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 : Rl1vl Equ      01000000b ; ring indicator level
133 : DSrlvl Equ     00100000b ; DSR level
134 : CTS1vl Equ     00010000b ; CTS level
135 : CDchg Equ      00001000b ; Carrier Detect change
136 : Rl1chg 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       Offh ; 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    Struct      ; 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 : ;dbpptr 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

```

Figure 6-1. Continued.

(more)

```
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 : Strategy 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 : Strategy 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 : BldBPP:
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 :

```

Figure 6-1. Continued.

(more)

```

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 :      Cld
518 :      Jnc    $+2
519 :      Inc     Dx              ; mdm ctrl
520 :      Mov     Al,es:[di]
521 :      Or     Al,Usr2         ; Int Gate
522 :      Out    Dx,al

```

Figure 6-1. Continued.

(m)


```

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 :      Jump   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 :      Jump   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,CD1vl           ; 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,DSR1vl          ; 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 G-1. Continued.

(more)

```

727 : Dstson:
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_no_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],LinKof     ; 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],LinKof     ; yes, waiting?
770 :     Jz     Int_exit2            ; no, ignore it
771 :     Xor    OtStat[si],LinKof     ; 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 :      Jump    Int_exit
791 : Nocode:
792 :      Or      InStat[si],BadInp
793 :      Jump    Int_exit3
794 : Int_done:
795 :      Cld
796 :      Jnc     S+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 :        Cmc
838 :        Jnc   $+2
839 :        Mov   Dl,RxBuf
840 :        Mov   Ax,Baud[si]   ; set baud
841 :        Out   Dx,Al
842 :        Cmc
843 :        Jnc   $+2
844 :        Inc   Dx
845 :        Mov   Al,Ah
846 :        Out   Dx,Al
847 :        Cmc
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 :        Cmc
855 :        Jnc   $+2
856 :        Mov   Dl,IntEn       ; IER
857 :        Mov   Al,AllInt      ; enable ints in 8250
858 :        Out   Dx,Al
859 :        Cmc
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 :      Cmc
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 *Badlnp* 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    kbuf,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 kbuf+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 kbuf
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 mbuf
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 : mbuf    db     256 dup (0)
79 : kbuf    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 *mbuf* (where *getmdm* put them) to the display, and then it returns.

Similarly, *getkbd* gets keystrokes from the keyboard, stores them in *kbuf*, and posts a count in *kblen*; *putmdm* checks *kblen* and, if the count is not zero, sends the required number of bytes from *kbuf* 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 *kbuf* 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)

```

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, OverRun */
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 :     putchar ( ' ' );
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 ) iobuf;
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, iobuf [ 4 ],          /* baud rate          */
181 :           5 + ( iobuf [ 0 ] & m_WL ), /* word length        */
182 :           ( iobuf [ 0 ] & m_PG ?
183 :             ( iobuf [ 0 ] & m_PE ? 'E' : 'O' ) : 'N' ),
184 :           ( iobuf [ 0 ] & m_XS ? '2' : '1' )); /* stop bits        */
185 :   cputs ( temp );
186 :
187 :   cputs ( "Hardware Interrupts are" );
188 :   cputs ( onoff ( iobuf [ 0 ] & HWINT ));
189 :   cputs ( ", Data Terminal Rdy" );
190 :   cputs ( onoff ( iobuf [ 0 ] & o_DTR ));
191 :   cputs ( ", Rqst To Send" );
192 :   cputs ( onoff ( iobuf [ 0 ] & o_RTS ));
193 :   cputs ( ".\r\n" );
194 :
195 :   cputs ( "Carrier Detect" );
196 :   cputs ( onoff ( iobuf [ 1 ] & i_CD ));
197 :   cputs ( ", Data Set Rdy" );
198 :   cputs ( onoff ( iobuf [ 1 ] & i_DSR ));
199 :   cputs ( ", Clear to Send" );
200 :   cputs ( onoff ( iobuf [ 1 ] & i_CTS ));
201 :   cputs ( ", Ring Indicator" );
202 :   cputs ( onoff ( iobuf [ 1 ] & i_RI ));
203 :   cputs ( ".\r\n" );
204 :
205 :   cputs ( l_SRE & iobuf [ 1 ] ? "Xmtr SR Empty, " : "" );
206 :   cputs ( l_HRE & iobuf [ 1 ] ? "Xmtr HR Empty, " : "" );
207 :   cputs ( l_BRK & iobuf [ 1 ] ? "Break Received, " : "" );
208 :   cputs ( l_ER1 & iobuf [ 1 ] ? "Framing Error, " : "" );
209 :   cputs ( l_ER2 & iobuf [ 1 ] ? "Parity Error, " : "" );
210 :   cputs ( l_ER3 & iobuf [ 1 ] ? "Overrun Error, " : "" );
211 :   cputs ( l_RRF & iobuf [ 1 ] ? "Rcvr DR Full, " : "" );
212 :   cputs ( "\b\b.\r\n" );
213 :
214 :   cputs ( "Reception errors " );
215 :   if ( iobuf [ 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 ( iobuf [ 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	<i>main()</i>	Conventional entry point.
47-150	<i>disp()</i>	Main dispatching loop.
152-158	<i>center()</i>	Centers text on CRT.
160-166	<i>iocwr()</i>	Writes control string to driver with IOCTL Write.
168-170	<i>onoff()</i>	Returns pointer to ON or OFF.
172-233	<i>report()</i>	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 : imrnsk   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

```

Figure 6-5. Continued.

(more)

```

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	<code>_set_mdm</code>	Initializes UART as specified by parameter passed from C.
86-114	<code>_wrtmdm</code>	Outputs character to UART.
87	<code>_Send_Byte</code>	Entry point for use if flow control is added to system.
116-140	<code>_rdmdm</code>	Gets character from buffer where ISR put it, or signals that no character available.
142-155	<code>w_tmr</code>	Wait timer; internal routine used to prevent infinite wait in case of problems.
157-182	<code>rts_m</code>	Hardware ISR; installed by <code>_i_m</code> and removed by <code>_u_m</code> .
184-240	<code>_i_m</code>	Installs ISR, saving old interrupt vector.
242-265	<code>_u_m</code>	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 revectoring 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 routines 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 module 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     noht
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 : noht:   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)

```

25 :
26 : static int I,
27 :     waitchr = 0,
28 :     vflag = False,
29 :     capbp,
30 :     capbc,
31 :     Ch,
32 :     Want_7_Bit = True,
33 :     ESC_Seq_State = 0;          /* escape sequence state variable */
34 :
35 : int _cx ,
36 :     _cy,
37 :     _atr = 0x07,                /* white on black */
38 :     _pag = 0,
39 :     oldtop = 0,
40 :     oldbot = 0x184f;
41 :
42 : FILE * in_file = NULL;         /* start with keyboard input */
43 : FILE * cap_file = NULL;
44 :
45 : #include "cterm.h"            /* external declarations, etc. */
46 :
47 : int Wants_To_Abort ()         /* checks for interrupt of script */
48 : { return broke ();
49 : }
50 : void
51 :
52 : main ( argc, argv ) int argc ; /* main routine */
53 : char * argv [];
54 : { char * cp,
55 :     * addext ();
56 :   if ( argc > 1 )              /* check for script filename */
57 :     in_file = fopen ( addext ( argv [ 1 ], ".SCR" ), "r" );
58 :   if ( argc > 2 )              /* check for capture filename */
59 :     cap_file = fopen ( addext ( argv [ 2 ], ".CAP" ), "w" );
60 :   set_int ();                  /* install CH1 module */
61 :   Set_Vid ();                  /* get video setup */
62 :   cls ();                      /* clear the screen */
63 :   cputs ( "Terminal Emulator" ); /* tell who's working */
64 :   cputs ( "\r\n< ESC for local commands >\r\n\n" );
65 :   Want_7_Bit = True;
66 :   ESC_Seq_State = 0;
67 :   Init_Comm ();               /* set up drivers, etc. */
68 :   while ( 1 )                 /* main loop */
69 :     { if ( ( Ch = kb_file () ) > 0 ) /* check local */
70 :       { if ( Is_Function_Key ( Ch ) )
71 :         { if ( docmd () < 0 ) /* command */
72 :           break;
73 :         }
74 :       else
75 :         Send_Byte ( Ch & 0x7F ); /* else send it */

```

Figure 6-8. Continued.

(more)

```

76 :     )
77 :     if (( Ch = Read_Modem ()) >= 0 ) /* check remote      */
78 :     { if ( Want_7_Bit )
79 :       Ch &= 0x7F; /* trim off high bit      */
80 :       switch ( ESC_Seq_State ) /* state machine      */
81 :       {
82 :       case 0 : /* no Esc sequence      */
83 :         switch ( Ch )
84 :         {
85 :         case ESC : /* Esc char received      */
86 :           ESC_Seq_State = 1;
87 :           break;
88 :
89 :         default :
90 :           if ( Ch == waitchr ) /* wait if required      */
91 :             waitchr = 0;
92 :           if ( Ch == 12 ) /* clear screen on FF      */
93 :             cls ();
94 :           else
95 :             if ( Ch != 127 ) /* ignore rubouts      */
96 :               { putchx ( (char) Ch ); /* handle all others      */
97 :                 put_cap ( (char) Ch );
98 :               }
99 :         }
100 :        break;
101 :
102 :        case 1 : /* ESC -- process any escape sequences here      */
103 :          switch ( Ch )
104 :          {
105 :          case 'A' : /* VT52 up      */
106 :            ; /* nothing but stubs here      */
107 :            ESC_Seq_State = 0;
108 :            break;
109 :
110 :          case 'B' : /* VT52 down      */
111 :            ;
112 :            ESC_Seq_State = 0;
113 :            break;
114 :
115 :          case 'C' : /* VT52 left      */
116 :            ;
117 :            ESC_Seq_State = 0;
118 :            break;
119 :
120 :          case 'D' : /* VT52 right      */
121 :            ;
122 :            ESC_Seq_State = 0;
123 :            break;
124 :
125 :          case 'E' : /* VT52 Erase CRT      */
126 :            cls (); /* actually do this one      */

```

Figure 6-8. Continued.

(more)


```

127 :             ESC_Seq_State = 0;
128 :             break;
129 :
130 :             case 'H' :             /* VT52 home cursor      */
131 :                 locate ( 0, 0 );
132 :                 ESC_Seq_State = 0;
133 :                 break;
134 :
135 :             case 'j' :             /* VT52 Erase to EOS  */
136 :                 deos ();
137 :                 ESC_Seq_State = 0;
138 :                 break;
139 :
140 :             case '[' :             /* ANSI.SYS - VT100 sequence */
141 :                 ESC_Seq_State = 2;
142 :                 break;
143 :
144 :             default :
145 :                 putchar ( ESC );             /* pass thru all others */
146 :                 putchar ( (char) Ch );
147 :                 ESC_Seq_State = 0;
148 :             }
149 :             break;
150 :
151 :             case 2 :             /* ANSI 3.64 decoder   */
152 :                 ESC_Seq_State = 0;             /* not implemented    */
153 :             }
154 :         }
155 :         if ( broke () )             /* check CH1A handlers */
156 :             { cputs ( "\r\n***BREAK***\r\n" );
157 :             break;
158 :         }
159 :     }
160 :     if ( cap_file )             /* save any capture    */
161 :         cap_flush ();
162 :     Term_Comm ();             /* restore when done   */
163 :     rst_int ();             /* restore break handlers */
164 :     exit ( 0 );             /* be nice to MS-DOS  */
165 : }
166 :
167 : docmd ()             /* local command shell */
168 : { FILE * getfil ();
169 :   int wp;
170 :   wp = True;
171 :   if ( ! in_file || vflag )
172 :       cputs ( "\r\n\tCommand: " );             /* ask for command    */
173 :   else
174 :       wp = False;
175 :   Ch = toupper ( kbd_wait () );             /* get response      */
176 :   if ( wp )
177 :       putchar ( (char) Ch );

```

Figure 6-8. Continued.

(more)

```
178 : switch ( Ch ) /* and act on it */
179 : {
180 : case 'S' :
181 :     if ( wp )
182 :         cputs ( "low speed\r\n" );
183 :         Set_Baud ( 300 );
184 :         break;
185 :
186 : case 'D' :
187 :     if ( wp )
188 :         cputs ( "elay (1-9 sec): " );
189 :         Ch = kbd_wait ();
190 :         if ( wp )
191 :             putchx ( (char) Ch );
192 :             Delay ( 1000 * ( Ch - '0' ) );
193 :             if ( wp )
194 :                 putchx ( '\n' );
195 :                 break;
196 :
197 : case 'E' :
198 :     if ( wp )
199 :         cputs ( "ven Parity\r\n" );
200 :         Set_Parity ( 2 );
201 :         break;
202 :
203 : case 'F' :
204 :     if ( wp )
205 :         cputs ( "ast speed\r\n" );
206 :         Set_Baud ( 1200 );
207 :         break;
208 :
209 : case 'H' :
210 :     if ( wp )
211 :         { cputs ( "\r\n\tINVALID COMMANDS:\r\n" );
212 :           cputs ( "\tD = delay 0-9 seconds.\r\n" );
213 :           cputs ( "\tE = even parity.\r\n" );
214 :           cputs ( "\tF = (fast) 1200-baud.\r\n" );
215 :           cputs ( "\tN = no parity.\r\n" );
216 :           cputs ( "\tO = odd parity.\r\n" );
217 :           cputs ( "\tQ = quit, return to DOS.\r\n" );
218 :           cputs ( "\tR = reset modem.\r\n" );
219 :           cputs ( "\tS = (slow) 300-baud.\r\n" );
220 :           cputs ( "\tU = use script file.\r\n" );
221 :           cputs ( "\tV = verify file input.\r\n" );
222 :           cputs ( "\tW = wait for char." );
223 :         }
224 :         break;
225 :
226 : case 'N' :
227 :     if ( wp )
```

Figure 6-8. Continued.

(more)

```
228 :         cputs ( "o Parity\r\n" );
229 :         Set_Parity ( 1 );
230 :         break;
231 :
232 :     case 'O' :
233 :         if ( wp )
234 :             cputs ( "dd Parity\r\n" );
235 :         Set_Parity ( 3 );
236 :         break;
237 :
238 :     case 'R' :
239 :         if ( wp )
240 :             cputs ( "ESET Comm Port\r\n" );
241 :         Init_Comm ();
242 :         break;
243 :
244 :     case 'Q' :
245 :         if ( wp )
246 :             cputs ( " = QUIT Command\r\n" );
247 :         Ch = ( - 1 );
248 :         break;
249 :
250 :     case 'U' :
251 :         if ( in_file && ! vflag )
252 :             putchar ( 'U' );
253 :         cputs ( "se file: " );
254 :         getfil ();
255 :         cputs ( "File " );
256 :         cputs ( in_file ? "Open\r\n" : "Bad\r\n" );
257 :         waitchr = 0;
258 :         break;
259 :
260 :     case 'V' :
261 :         if ( wp )
262 :             { cputs ( "erify flag toggled " );
263 :               cputs ( vflag ? "OFF\r\n" : "ON\r\n" );
264 :             }
265 :         vflag = vflag ? False : True;
266 :         break;
267 :
268 :     case 'W' :
269 :         if ( wp )
270 :             cputs ( "ait for: <" );
271 :         waitchr = kbd_wait ();
272 :         if ( waitchr == ' ' )
273 :             waitchr = 0;
274 :         if ( wp )
275 :             { if ( waitchr )
276 :               putchar ( (char) waitchr );
277 :               else
278 :                 cputs ( "no wait" );
```

Figure 6-8. Continued.

(more)

```

279 :         cputs ( ">\r\n" );
280 :     }
281 :     break;
282 :
283 :     default :
284 :         if ( wp )
285 :             { cputs ( "Don't know " );
286 :               putchar ( (char) Ch );
287 :               cputs ( "\r\nUse 'H' command for Help.\r\n" );
288 :             }
289 :         Ch = '?';
290 :     }
291 :     if ( wp )                               /* if window open... */
292 :         { cputs ( "\r\n[any key]\r" );
293 :           while ( Read_Keyboard () == EOF ) /* wait for response */
294 :               ;
295 :         }
296 :     return Ch ;
297 : }
298 :
299 : kbd_wait ()                                /* wait for input */
300 : { int c ;
301 :   while (( c = kb_file () ) == ( - 1 ))
302 :       ;
303 :   return c & 255;
304 : }
305 :
306 : kb_file ()                                  /* input from kb or file */
307 : { int c ;
308 :   if ( in_file )                            /* USING SCRIPT */
309 :       { c = Wants_To_Abort ();              /* use first as flag */
310 :         if ( waitchr && ! c )
311 :             c = ( - 1 );                    /* then for char */
312 :         else
313 :             if ( c || ( c = getc ( in_file )) == EOF || c == 26 )
314 :                 { fclose ( in_file );
315 :                   cputs ( "\r\nScript File Closed\r\n" );
316 :                   in_file = NULL;
317 :                   waitchr = 0;
318 :                   c = ( - 1 );
319 :                 }
320 :         else
321 :             if ( c == '\n' )                /* ignore LFs in file */
322 :                 c = ( - 1 );
323 :             if ( c == '\\')                 /* process Esc sequence */
324 :                 c = esc ();
325 :             if ( vflag && c != ( - 1 ))      /* verify file char */
326 :                 { putchar ( '(' );
327 :                   putchar ( (char) c );
328 :                   putchar ( ')' );
329 :                 }

```

Figure 6-8. Continued.

(more)

```

330 :     }
331 :     else                               /* USING CONSOLE      */
332 :         c = Read_Keyboard ();           /* if not using file    */
333 :     return ( c );
334 : }
335 :
336 : esc ()                                   /* script translator   */
337 : { int c ;
338 :   c = getc ( in_file );                 /* control chars in file */
339 :   switch ( toupper ( c ))
340 :   {
341 :     case 'E' :
342 :       c = ESC;
343 :       break;
344 :
345 :     case 'N' :
346 :       c = '\n';
347 :       break;
348 :
349 :     case 'R' :
350 :       c = '\r';
351 :       break;
352 :
353 :     case 'T' :
354 :       c = '\t';
355 :       break;
356 :
357 :     case '^' :
358 :       c = getc ( in_file ) & 31;
359 :       break;
360 :   }
361 :   return ( c );
362 : }
363 :
364 : FILE * getfil ()
365 : { char fnm [ 20 ];
366 :   getnam ( fnm, 15 );                   /* get the name        */
367 :   if ( ! ( strchr ( fnm, '.' )))
368 :     strcat ( fnm, ".SCR" );
369 :   return ( in_file = fopen ( fnm, "r" ));
370 : }
371 :
372 : void getnam ( b, s ) char * b;          /* take input to buffer */
373 : int s ;
374 : { while ( s -- > 0 )
375 :   { if ( ( * b = (char) kbd_wait ( ) ) != '\r' )
376 :     putchar ( * b ++ );
377 :     else
378 :       break ;
379 :   }
380 :   putchar ( '\n' );

```

Figure 6-8. Continued.

(more)

```

381 :   * b = 0;
382 : }
383 :
384 : char * addext ( b,           /* add default EXTension */
385 :   e ) char * b,
386 :   * e;
387 : { static char bfr [ 20 ];
388 :   if ( strchr ( b, '.' ))
389 :     return ( b );
390 :   strcpy ( bfr, b );
391 :   strcat ( bfr, e );
392 :   return ( bfr );
393 : }
394 :
395 : void put_cap ( c ) char c ;
396 : { if ( cap_file && c != 13 )      /* strip out CRs      */
397 :   fputc ( c, cap_file );        /* use MS-DOS buffering */
398 : }
399 :
400 : void cap_flush ()                /* end Capture mode   */
401 : { if ( cap_file )
402 :   { fclose ( cap_file );
403 :     cap_file = NULL;
404 :     cputs ( "\r\nCapture file closed\r\n" );
405 :   }
406 : }
407 :
408 : /*      TIMER SUPPORT STUFF (IBMP/MSDOS)      */
409 : static long timr;                /* timeout register    */
410 :
411 : static union REGS rgv ;
412 :
413 : long getmr ()
414 : { long now ;                      /* msec since midnite */
415 :   rgv.x.ax = 0x2c00;
416 :   intdos ( & rgv, & rgv );
417 :   now = rgv.h.ch;                 /* hours              */
418 :   now *= 60L;                     /* to minutes         */
419 :   now += rgv.h.cl;                /* plus min           */
420 :   now *= 60L;                     /* to seconds         */
421 :   now += rgv.h.dh;                /* plus sec           */
422 :   now *= 100L;                    /* to 1/100           */
423 :   now += rgv.h.dl;                /* plus 1/100        */
424 :   return ( 10L * now );           /* msec value         */
425 : }
426 :
427 : void Delay ( n ) int n ;          /* sleep for n msec   */
428 : { long wakeup ;
429 :   wakeup = getmr () + ( long ) n; /* wakeup time        */
430 :   while ( getmr () < wakeup )
431 :     ;                             /* now sleep          */

```

Figure 6-8. Continued.

(more)

```

432 : }
433 :
434 : void Start_Timer ( n ) int n ;          /* set timeout for n sec */
435 : { timr = getmr () + ( long ) n * 1000L;
436 : }
437 :
438 : Timer_Expired ()          /* if timeout return 1 else return 0 */
439 : { return ( getmr () > timr );
440 : }
441 :
442 : Set_Vid ()
443 : { _i_v ();                /* initialize video */
444 :   return 0;
445 : }
446 :
447 : void locate ( row, col ) int row ,
448 :           col;
449 : { _cy = row % 25;
450 :   _cx = col % 80;
451 :   _wrpos ( row, col );    /* use ML from CH2.ASM */
452 : }
453 :
454 : void deol ()
455 : { _deol ();              /* use ML from CH2.ASM */
456 : }
457 :
458 : void deos ()
459 : { deol ();
460 :   if ( _cy < 24 )        /* if not last, clear */
461 :     { rgv.x.ax = 0x0600;
462 :       rgv.x.bx = ( _atr << 8 );
463 :       rgv.x.cx = ( _cy + 1 ) << 8;
464 :       rgv.x.dx = 0x184F;
465 :       int86 ( 0x10, & rgv, & rgv );
466 :     }
467 :   locate ( _cy, _cx );
468 : }
469 :
470 : void cls ()
471 : { _cls ();              /* use ML */
472 : }
473 :
474 : void cursor ( yn ) int yn ;
475 : { rgv.x.cx = yn ? 0x0607 : 0x2607;    /* ON/OFF */
476 :   rgv.x.ax = 0x0100;
477 :   int86 ( 0x10, & rgv, & rgv );
478 : }
479 :
480 : void revvid ( yn ) int yn ;
481 : { if ( yn )
482 :   _atr = _color ( 8, 7 );    /* black on white */

```

Figure 6-8. Continued.

(more)

```

483 :   else
484 :       _atr = _color ( 15, 0 );           /* white on black      */
485 :   )
486 :
487 :   putchar ( c ) char c ;                 /* put char to CRT    */
488 :   { if ( c == '\n' )
489 :       putchar ( '\r' );
490 :   putchar ( c );
491 :   return c ;
492 :   }
493 :
494 :   Read_Keyboard ()                       /* get keyboard character
495 :       returns -1 if none present */
496 :   { int c ;
497 :     if ( kbhit ()                       /* no char at all    */
498 :         return ( getch () );
499 :     return ( EOF );
500 :   }
501 :
502 :   /*      MODEM SUPPORT                    */
503 :   static char mparm,
504 :       wrk [ 80 ];
505 :
506 :   void Init_Comm ()                      /* initialize comm port stuff
507 :       { static int ft = 0;                /* firsttime flag    */
508 :         if ( ft ++ == 0 )
509 :             i_m ();
510 :         Set_Parity ( 1 );                 /* 8,N,1              */
511 :         Set_Baud ( 1200 );                /* 1200 baud          */
512 :     }
513 :
514 :   #define B1200 0x80                      /* baudrate codes    */
515 :   #define B300 0x40
516 :
517 :   Set_Baud ( n ) int n ;                  /* n is baud rate    */
518 :   { if ( n == 300 )
519 :       mparm = ( mparm & 0x1F ) + B300;
520 :     else
521 :         if ( n == 1200 )
522 :             mparm = ( mparm & 0x1F ) + B1200;
523 :     else
524 :         return 0;                        /* invalid speed     */
525 :     sprintf ( wrk, "Baud rate = %d\r\n", n );
526 :     cputs ( wrk );
527 :     set_mdm ( mparm );
528 :     return n ;
529 :   }
530 :
531 :   #define PAREVN 0x18                     /* MCR bits for commands
532 :   #define PARODD 0x10
533 :   #define PAROFF 0x00

```

Figure 6-8. Continued.

(more)


```

534 : #define STOP2 0x40
535 : #define WORD8 0x03
536 : #define WORD7 0x02
537 : #define WORD6 0x01
538 :
539 : Set_Parity ( n ) int n ;           /* n is parity code      */
540 : { static int mmode;
541 :   if ( n == 1 )
542 :     mmode = ( WORD8 | PAROFF );    /* off                    */
543 :   else
544 :     if ( n == 2 )
545 :       mmode = ( WORD7 | PAREVN );  /* on and even            */
546 :   else
547 :     if ( n == 3 )
548 :       mmode = ( WORD7 | PARODD );  /* on and odd             */
549 :   else
550 :     return 0;                      /* invalid code           */
551 :   mparm = ( mparm & 0xE0 ) + mmode;
552 :   sprintf ( wrk, "Parity is %s\r\n", ( n == 1 ? "OFF" :
553 :     ( n == 2 ? "EVEN" : "ODD" ) ));
554 :   cputs ( wrk );
555 :   set_mdm ( mparm );
556 :   return n ;
557 : }
558 :
559 : Write_Modem ( c ) char c ;         /* return 1 if ok, else 0 */
560 : { wrtmdm ( c );
561 :   return ( 1 );                   /* never any error        */
562 : }
563 :
564 : Read_Modem ( )
565 : { return ( rdmdm ());              /* from int bfr           */
566 : }
567 :
568 : void Term_Comm ( )                 /* uninstall comm port drivers */
569 : { u_m ( );
570 : }
571 :
572 : /* end of cterm.c */

```

Figure 6-8. Continued.

CTERM features file-capture capabilities, a simple yet effective script language, and a number of stub (that is, incompletely implemented) actions, such as emulation of the VT52 and VT100 series terminals, indicating various directions in which it can be developed.

The names of a script file and a capture file can be passed to CTERM in the command line. If no filename extensions are included, the default for the script file is .SCR and that for the capture file is .CAP. If extensions are given, they override the default values. The capture feature can be invoked only if a filename is supplied in the command line, but a script file can be called at any time via the Esc command sequence, and one script file can call for another with the same feature.

The functions included in CTERM.C are listed and summarized in Table 6-13.

Table 6-13. CTERM.C Functions.

Lines	Name	Description
1-5		Program documentation.
7-11		<i>Include</i> files.
12-20		Definitions.
22-43		Global data areas.
45		External prototype declaration.
47-49	<i>Wants_To_Abort()</i>	Checks for Ctrl-Break or Ctrl-C being pressed.
52-165	<i>main()</i>	Main program loop; includes modem engine and sequential state machine to decode remote commands.
167-297	<i>docmd()</i>	Gets, interprets, and performs local (console or script) command.
299-304	<i>kbd_wait()</i>	Waits for input from console or script file.
306-334	<i>kb_file()</i>	Gets keystroke from console or script; returns EOF if no character available.
336-362	<i>esc()</i>	Translates script escape sequence.
364-370	<i>getfil()</i>	Gets name of script file and opens the file.
372-382	<i>getnam()</i>	Gets string from console or script into designated buffer.
384-393	<i>addext()</i>	Checks buffer for extension; adds one if none given.
395-398	<i>put_cap()</i>	Writes character to capture file if capture in effect.
400-406	<i>cap_flush()</i>	Closes capture file and terminates capture mode if capture in effect.
408-411		Timer data locations.
413-425	<i>getmr()</i>	Returns time since midnight, in milliseconds.
427-432	<i>Delay()</i>	Sleeps <i>n</i> milliseconds.
434-436	<i>Start_Timer()</i>	Sets timer for <i>n</i> seconds.
438-440	<i>Timer_Expired()</i>	Checks timer versus clock.
442-445	<i>Set_Vid()</i>	Initializes video data.
447-452	<i>locate()</i>	Positions cursor on display.
454-456	<i>deol()</i>	Deletes to end of line.
458-468	<i>deos()</i>	Deletes to end of screen.
470-472	<i>cls()</i>	Clears screen.
474-478	<i>cursor()</i>	Turns cursor on or off.
480-485	<i>revvid()</i>	Toggles inverse/normal video display attributes.
487-492	<i>putchx()</i>	Writes char to display using <i>putch()</i> (Microsoft C library).

(more)

Table 6-13. *Continued.*

Lines	Name	Description
494-500	<i>Read_Keyboard()</i>	Gets keystroke from keyboard.
502-504		Modem data areas.
506-512	<i>Init_Comm()</i>	Installs ISR and so forth and initializes modem.
514-515		Baud-rate definitions.
517-529	<i>Set_Baud()</i>	Changes bps rate of UART.
531-537		Parity, WL definitions.
539-557	<i>Set_Parity()</i>	Establishes UART parity mode.
559-562	<i>Write_Modem()</i>	Sends character to UART.
564-566	<i>Read_Modem()</i>	Gets character from ISR's buffer.
568-570	<i>Term_Comm()</i>	Uninstalls ISR and so forth and restores original vectors.

For communication with the console, CTERM uses the special Microsoft C library functions defined by CONIO.H, augmented with the functions in the CH2.ASM handler. Much of the code may require editing if used with other compilers. CTERM also uses the function prototype file CTERM.H, listed in Figure 6-9, to optimize function calling within the program.

```

/* CTERM.H - function prototypes for CTERM.C */
int Wants_To_Abort(void);
void main(int ,char * *);
int docmd(void);
int kbd_wait(void);
int kb_file(void);
int esc(void);
FILE *getfil(void);
void getnam(char *,int );
char *addext(char *,char *);
void put_cap(char );
void cap_flush(void);
long getmr(void);
void Delay(int );
void Start_Timer(int );
int Timer_Expired(void);
int Set_Vid(void);
void locate(int ,int );
void deol(void);
void deos(void);
void cls(void);
void cursor(int );
void revvid(int );
int putchx(char );

```

*Figure 6-9. CTERM.H.**(more)*

```
int Read_Keyboard(void);
void Init_Comm(void);
int Set_Baud(int );
int Set_Parity(int );
int Write_Modem(char );
int Read_Modem(void);
void Term_Comm(void);

/* CH1.ASM functions - modem interfacing */
void i_m(void);
void set_mdm(int);
void wrtmdm(int);
void Send_Byte(int);
int rdmdm(void);
void u_m(void);

/* CH1A.ASM functions - exception handlers */
void set_int (void);
void rst_int (void);
int broke (void);

/* CH2.ASM functions - video interfacing */
void _i_v(void);
int _wrpos(int, int);
void _deol(void);
void _cls(void);
int _color(int, int);
```

Figure 6-9. Continued.

Program execution begins at the entry to *main()*, line 52. CTERM first checks (lines 56 through 59) whether any filenames were passed in the command line; if they were, CTERM opens the corresponding files. Next, the program installs the exception handler (line 60), initializes the video handler (line 61), clears the display (line 62), and announces its presence (lines 63 and 64). The serial driver is installed and initialized to 1200 bps and no parity (lines 65 through 67), and the program enters its main modem-engine loop (lines 68 through 159).

This loop is functionally the same as that used in ENGINE, but it has been extended to detect an Esc from the keyboard as signalling the start of a local command sequence (lines 70 through 73) and to include a state-machine technique (lines 80 through 153) to recognize incoming escape sequences, such as the VT52 or VT100 codes. To specify a local command from the keyboard, press the Escape (Esc) key, then the first letter of the local command desired. After the local command has been selected, press any key (such as Enter or the spacebar) to continue. To get a listing of all the commands available, press Esc-H.

The *kb_file()* routine of CTERM (called in the main loop at line 69) can get its input from either a script file or the keyboard. If a script file is open (lines 308 through 330), it is used until EOF is reached or until the operator presses Ctrl-C to stop script-file input. Otherwise,

input is taken from the keyboard (lines 331 and 332). If a script file is in use, its input is echoed to the display (lines 325 through 329) if the V command has been given.

To permit the Esc character itself to be placed in script files, the backslash (\) character serves as a secondary escape signal. When a backslash is detected (lines 323 and 324) in the input stream, the next character input is translated according to the following rules:

Character	Interpretation
E or e	Translates to Esc.
N or n	Translates to Linefeed.
R or r	Translates to Enter (CR).
T or t	Translates to Tab.
^	Causes the <i>next</i> character input to be converted into a control character.

Any other character, including another \, is not translated at all.

When the Esc character is detected from either the console or a script file, the *docmd()* function (lines 167 through 297) is called to prompt for and decode the next input character as a command and to perform appropriate actions. Valid command characters, and the actions they invoke, are as follows:

Command Character	Action
D	Delay 0–9 seconds, then proceed. Must be followed by a decimal digit that indicates how long to delay.
E	Set EVEN parity.
F	Set (fast) 1200 baud.
H	Display list of valid commands.
N	Set no parity.
O	Set ODD parity.
Q	Quit; return to MS-DOS command prompt.
R	Reset modem.
S	Set (slow) 300 baud.
U	Use script file (CTERM prompts for filename).
V	Verify file input. Echoes each script-file byte.
W	Wait for character; the next input character is the one that must be matched.

Any other character input after an Esc and the resulting Command prompt generates the message *Don't know X* (where *X* stands for the actual input character) followed by the prompt *Use 'H' command for Help*.

If input is taken from a script and the V flag is off, *docmd()* performs its task quietly, with no output to the screen. If input is received from the console, however, the command letter, followed by a descriptive phrase, is echoed to the screen. Input, detection, and execution of the local commands are accomplished much as in CDVUTL, by way of a large *switch()* statement (lines 178 through 290).

Although the listed commands are only a subset of the features available in CDVUTL for the device-driver program, they are more than adequate for creating useful scripts. The predecessor of CTERM (DT115.EXE), which included the CompuServe B-Protocol file-transfer capability but had no additional commands, has been in use since early 1986 to handle automatic uploading and downloading of files from the CompuServe Information Service by means of script files. In conjunction with an auto-dialing modem, DT115.EXE handles the entire transaction, from login through logout, without human intervention.

All the bits and pieces of CTERM are put together by assembling the three handlers with MASM, compiling CTERM with Microsoft C, and linking all four object modules into an executable file. Figure 6-10 shows the complete sequence and also the three ways of using the finished program.

Compiling:

```
C>MASM CH1; <Enter>
C>MASM CH1A; <Enter>
C>MASM CH2; <Enter>
C>MSC CTERM; <Enter>
```

Linking:

```
C>LINK CTERM+CH1+CH1A+CH2; <Enter>
```

Use:

(no files)

```
C>CTERM <Enter>
```

or

(script only)

```
C>CTERM scriptfile <Enter>
```

or

```
C>CTERM scriptfile capturefile <Enter>
```

Figure 6-10. Putting CTERM together and using it.

*Jim Kyle
Chip Rabinowitz*

Article 7

File and Record Management

The core of most application programs is the reading, processing, and writing of data stored on magnetic disks. This data is organized into files, which are identified by name; the files, in turn, can be organized by grouping them into directories. Operating systems provide application programs with services that allow them to manipulate these files and directories without regard to the hardware characteristics of the disk device. Thus, applications can concern themselves solely with the form and content of the data, leaving the details of the data's location on the disk and of its retrieval to the operating system.

The disk storage services provided by an operating system can be categorized into file functions and record functions. The file functions operate on entire files as named entities, whereas the record functions provide access to the data contained within files. (In some systems, an additional class of directory functions allows applications to deal with collections of files as well.) This article discusses the MS-DOS function calls that allow an application program to create, open, close, rename, and delete disk files; read data from and write data to disk files; and inspect or change the information (such as attributes and date and time stamps) associated with disk filenames in disk directories. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS; MS-DOS Storage Devices; PROGRAMMING FOR MS-DOS: Disk Directories and Volume Labels.

Historical Perspective

Current versions of MS-DOS provide two overlapping sets of file and record management services to support application programs: the handle functions and the file control block (FCB) functions. Both sets are available through Interrupt 21H (Table 7-1). *See* SYSTEM CALLS: INTERRUPT 21H. The reasons for this surprising duplication are strictly historical.

The earliest versions of MS-DOS used FCBs for all file and record access because CP/M, which was the dominant operating system on 8-bit microcomputers, used FCBs. Microsoft chose to maintain compatibility with CP/M to aid programmers in converting the many existing CP/M application programs to the 16-bit MS-DOS environment; consequently, MS-DOS versions 1.x included a set of FCB functions that were a functional superset of those present in CP/M. As personal computers evolved, however, the FCB access method did not lend itself well to the demands of larger, faster disk drives.

Accordingly, MS-DOS version 2.0 introduced the handle functions to provide a file and record access method similar to that found in UNIX/XENIX. These functions are easier to use and more flexible than their FCB counterparts and fully support a hierarchical (tree-like) directory structure. The handle functions also allow character devices, such as the

console or printer, to be treated for some purposes as though they were files. MS-DOS version 3.0 introduced additional handle functions, enhanced some of the existing handle functions for use in network environments, and provided improved error reporting for all functions.

The handle functions, which offer far more capability and performance than the FCB functions, should be used for all new applications. Therefore, they are discussed first in this article.

Table 7-1. Interrupt 21H Function Calls for File and Record Management.

Operation	Handle Function	FCB Function
Create file.	3CH	16H
Create new file.	5BH	
Create temporary file.	5AH	
Open file.	3DH	0FH
Close file.	3EH	10H
Delete file.	41H	13H
Rename file.	56H	17H
Perform sequential read.	3FH	14H
Perform sequential write.	40H	15H
Perform random record read.	3FH	21H
Perform random record write.	40H	22H
Perform random block read.		27H
Perform random block write.		28H
Set disk transfer area address.		1AH
Get disk transfer area address.		2FH
Parse filename.		29H
Position read/write pointer.	42H	
Set random record number.		24H
Get file size.	42H	23H
Get/Set file attributes.	43H	
Get/Set date and time stamp.	57H	
Duplicate file handle.	45H	
Redirect file handle.	46H	

Using the Handle Functions

The initial link between an application program and the data stored on disk is the name of a disk file in the form

drive:path\filename.ext

where *drive* designates the disk on which the file resides, *path* specifies the directory on that disk in which the file is located, and *filename.ext* identifies the file itself. If *drive* and/or *path* is omitted, MS-DOS assumes the default disk drive and current directory. Examples of acceptable pathnames include

```
C:\PAYROLL\TAXES.DAT
LETTERS\MEMO.TXT
BUDGET.DAT
```

Pathnames can be hard-coded into a program as part of its data. More commonly, however, they are entered by the user at the keyboard, either as a command-line parameter or in response to a prompt from the program. If the pathname is provided as a command-line parameter, the application program must extract it from the other information in the command line. Therefore, to allow a program to distinguish between pathnames and other parameters when the two are combined in a command line, the other parameters, such as switches, usually begin with a slash (/) or dash (-) character.

All handle functions that use a pathname require the name to be in the form of an ASCIIZ string — that is, the name must be terminated by a null (zero) byte. If the pathname is hard-coded into a program, the null byte must be part of the ASCIIZ string. If the pathname is obtained from keyboard input or from a command-line parameter, the null byte must be appended by the program. See *Opening an Existing File* below.

To use a disk file, a program opens or creates the file by calling the appropriate MS-DOS function with the ASCIIZ pathname. MS-DOS checks the pathname for invalid characters and, if the open or create operation is successful, returns a 16-bit handle, or identification code, for the file. The program uses this handle for subsequent operations on the file, such as record reads and writes.

The total number of handles for simultaneously open files is limited in two ways. First, the per-process limit is 20 file handles. The process's first five handles are always assigned to the standard devices, which default to the CON, AUX, and PRN character devices:

Handle	Service	Default
0	Standard input	Keyboard (CON)
1	Standard output	Video display (CON)
2	Standard error	Video display (CON)
3	Standard auxiliary	First communications port (AUX)
4	Standard list	First parallel printer port (PRN)

Ordinarily, then, a process has only 15 handles left from its initial allotment of 20; however, when necessary, the 5 standard device handles can be redirected to other files and devices or closed and reused.

In addition to the per-process limit of 20 file handles, there is a system-wide limit. MS-DOS maintains an internal table that keeps track of all the files and devices opened with file handles for all currently active processes. The table contains such information as the current file pointer for read and write operations and the time and date of the last write to the file. The size of this table, which is set when MS-DOS is initially loaded into memory, determines the system-wide limit on how many files and devices can be open simultaneously. The default limit is 8 files and devices; thus, this system-wide limit usually overrides the per-process limit.

To increase the size of MS-DOS's internal handle table, the statement *FILES=nnn* can be included in the CONFIG.SYS file. (CONFIG.SYS settings take effect the next time the system is turned on or restarted.) The maximum value for FILES is 99 in MS-DOS versions 2.x and 255 in versions 3.x. See USER COMMANDS: CONFIG.SYS: FILES.

Error handling and the handle functions

When a handle-based file function succeeds, MS-DOS returns to the calling program with the carry flag clear. If a handle function fails, MS-DOS sets the carry flag and returns an error code in the AX register. The program should check the carry flag after each operation and take whatever action is appropriate when an error is encountered. Table 7-2 lists the most frequently encountered error codes for file and record I/O (exclusive of network operations).

Table 7-2. Frequently Encountered Error Diagnostics for File and Record Management.

Code	Error
02	File not found
03	Path not found
04	Too many open files (no handles left)
05	Access denied
06	Invalid handle
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid disk drive letter
17	Not same device
18	No more files

The error codes used by MS-DOS in versions 3.0 and later are a superset of the MS-DOS version 2.0 error codes. See APPENDIX B: CRITICAL ERROR CODES; APPENDIX C: EXTENDED ERROR CODES. Most MS-DOS version 3 error diagnostics relate to network operations, which provide the program with a greater chance for error than does a single-user system.

Programs that are to run in a network environment need to anticipate network problems. For example, the server can go down while the program is using shared files.

Under MS-DOS versions 3.x, a program can also use Interrupt 21H Function 59H (Get Extended Error Information) to obtain more details about the cause of an error after a failed handle function. The information returned by Function 59H includes the type of device that caused the error and a recommended recovery action.

Warning: Many file and record I/O operations discussed in this article can result in or be affected by a hardware (critical) error. Such errors can be intercepted by the program if it contains a custom critical error exception handler (Interrupt 24H). See PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

Creating a file

MS-DOS provides three Interrupt 21H handle functions for creating files:

Function	Name
3CH	Create File with Handle (versions 2.0 and later)
5AH	Create Temporary File (versions 3.0 and later)
5BH	Create New File (versions 3.0 and later)

Each function is called with the segment and offset of an ASCII pathname in the DS:DX registers and the attribute to be assigned to the new file in the CX register. The possible attribute values are

Code	Attribute
00H	Normal file
01H	Read-only file
02H	Hidden file
04H	System file

Files with more than one attribute can be created by combining the values listed above. For example, to create a file that has both the read-only and system attributes, the value 05H is placed in the CX register.

If the file is successfully created, MS-DOS returns a file handle in AX that must be used for subsequent access to the new file and sets the file read/write pointer to the beginning of the file; if the file is not created, MS-DOS sets the carry flag (CF) and returns an error code in AX.

Function 3CH is the only file-creation function available under MS-DOS versions 2.x. It must be used with caution, however, because if a file with the specified name already exists, Function 3CH will open it and truncate it to zero length, eradicating the previous contents of the file. This complication can be avoided by testing for the previous existence of the file with an open operation before issuing the create call.

Under MS-DOS versions 3.0 and later, Function 5BH is the preferred function in most cases because it will fail if a file with the same name already exists. In networking environments, this function can be used to implement semaphores, allowing the synchronization of programs running in different network nodes.

Function 5AH is used to create a temporary work file that is guaranteed to have a unique name. This capability is important in networking environments, where several copies of the same program, running in different nodes, may be accessing the same logical disk volume on a server. The function is passed the address of a buffer that can contain a drive and/or path specifying the location for the created file. MS-DOS generates a name for the created file that is a sequence of alphanumeric characters derived from the current time and returns the entire ASCIIZ pathname to the program in the same buffer, along with the file's handle in AX. The program must save the filename so that it can delete the file later, if necessary; the file created with Function 5AH is not destroyed when the program exits.

Example: Create a file named MEMO.TXT in the \LETTERS directory on drive C using Function 3CH. Any existing file with the same name is truncated to zero length and opened.

```

fname db 'C:\LETTERS\MEMO.TXT',0
fhandle dw ?
.
.
.
mov dx,seg fname ; DS:DX = address of
mov ds,dx ; pathname for file
mov dx,offset fname
xor cx,cx ; CX = normal attribute
mov ah,3ch ; Function 3CH = create
int 21h ; transfer to MS-DOS
jc error ; jump if create failed
mov fhandle,ax ; else save file handle
.
.
.

```

Example: Create a temporary file using Function 5AH and place it in the \TEMP directory on drive C. MS-DOS appends the filename it generates to the original path in the buffer named *fname*. The resulting file specification can be used later to delete the file.

```

fname db 'C:\TEMP\' ; generated ASCIIZ filename
db 13 dup (0) ; is appended by MS-DOS
fhandle dw ?
.
.
.

```

(more)

```

mov    dx,seg fname    ; DS:DX = address of
mov    ds,dx           ; path for temporary file
mov    dx,offset fname
xor    cx,cx           ; CX = normal attribute
mov    ah,5ah          ; Function 5AH = create
                           ; temporary file
int    21h             ; transfer to MS-DOS
jc     error           ; jump if create failed
mov    fhandle,ax      ; else save file handle

```

Opening an existing file

Function 3DH (Open File with Handle) opens an existing normal, system, or hidden file in the current or specified directory. When calling Function 3DH, the program supplies a pointer to the ASCII pathname in the DS:DX registers and a 1-byte access code in the AL register. This access code includes the read/write permissions, the file-sharing mode, and an inheritance flag. The bits of the access code are assigned as follows:

Bit(s)	Description
0-2	Read/write permissions (versions 2.0 and later)
3	Reserved
4-6	File-sharing mode (versions 3.0 and later)
7	Inheritance flag (versions 3.0 and later)

The read/write permissions field of the access code specifies how the file will be used and can take the following values:

Bits 0-2	Description
000	Read permission desired
001	Write permission desired
010	Read and write permission desired

For the open to succeed, the permissions field must be compatible with the file's attribute byte in the disk directory. For example, if the program attempts to open an existing file that has the read-only attribute when the permissions field of the access code byte is set to write or read/write, the open function will fail and an error code will be returned in AX.

The sharing-mode field of the access code byte is important in a networking environment. It determines whether other programs will also be allowed to open the file and, if so, what operations they will be allowed to perform. Following are the possible values of the file-sharing mode field:

Bits 4–6 Description

000	Compatibility mode. Other programs can open the file and perform read or write operations as long as no process specifies any sharing mode other than compatibility mode.
001	Deny all. Other programs cannot open the file.
010	Deny write. Other programs cannot open the file in compatibility mode or with write permission.
011	Deny read. Other programs cannot open the file in compatibility mode or with read permission.
100	Deny none. Other programs can open the file and perform both read and write operations but cannot open the file in compatibility mode.

When file-sharing support is active (that is, SHARE.EXE has previously been loaded), the result of any open operation depends on both the contents of the permissions and file-sharing fields of the access code byte and the permissions and file-sharing requested by other processes that have already successfully opened the file.

The inheritance bit of the access code byte controls whether a child process will inherit that file handle. If the inheritance bit is cleared, the child can use the inherited handle to access the file without performing its own open operation. Subsequent operations performed by the child process on inherited file handles also affect the file pointer associated with the parent's file handle. If the inheritance bit is set, the child process does not inherit the handle.

If the file is opened successfully, MS-DOS returns its handle in AX and sets the file read/write pointer to the beginning of the file; if the file is not opened, MS-DOS sets the carry flag and returns an error code in AX.

Example: Copy the first parameter from the program's command tail in the program segment prefix (PSP) into the array *fname* and append a null character to form an ASCIIZ filename. Attempt to open the file with compatibility sharing mode and read/write access. If the file does not already exist, create it and assign it a normal attribute.

```
cmdtail equ      80h          ; PSP offset of command tail
fname  db       64 dup (?)
fhandle dw      ?

;
;
;
; assume that DS already
; contains segment of PSP
```

(more)

```

mov     si,cmdtail      ; prepare to copy filename...
mov     di,seg fname    ; DS:SI = command tail
mov     es,di           ; ES:DI = buffer to receive
mov     di,offset fname ; filename from command tail
cld

lodsb                    ; check length of command tail
or      al,al
jz      error           ; jump, command tail empty

label1:                  ; scan off leading spaces
lodsb                    ; get next character
cmp     al,20h          ; is it a space?
jz      label1         ; yes, skip it

label2:
cmp     al,0dh          ; look for terminator
jz      label3         ; quit if return found
cmp     al,20h          ; quit if space found
jz      label3         ; else copy this character
stosb                    ; get next character
lodsb
jmp     label2

label3:
xor     al,al           ; store final NULL to
stosb                    ; create ASCIIIZ string

; now open the file...
mov     dx,seg fname    ; DS:DX = address of
mov     ds,dx           ; pathname for file
mov     dx,offset fname
mov     ax,3d02h        ; Function 3DH = open r/w
int     21h            ; transfer to MS-DOS
jnc     label4         ; jump if file found

cmp     ax,2           ; error 2 = file not found
jnz     error         ; jump if other error
; else make the file...
xor     cx,cx          ; CX = normal attribute
mov     ah,3ch         ; Function 3CH = create
int     21h            ; transfer to MS-DOS
jc      error         ; jump if create failed

label4:
mov     fhandle,ax     ; save handle for file
.
.
.

```

Closing a file

Function 3EH (Close File) closes a file created or opened with a file handle function. The program must place the handle of the file to be closed in BX. If a write operation was performed on the file, MS-DOS updates the date, time, and size in the file's directory entry.

Closing the file also flushes the internal MS-DOS buffers associated with the file to disk and causes the disk's file allocation table (FAT) to be updated if necessary.

Good programming practice dictates that a program close files as soon as it finishes using them. This practice is particularly important when the file size has been changed, to ensure that data will not be lost if the system crashes or is turned off unexpectedly by the user. A method of updating the FAT without closing the file is outlined below under Duplicating and Redirecting Handles.

Reading and writing with handles

Function 3FH (Read File or Device) enables a program to read data from a file or device that has been opened with a handle. Before calling Function 3FH, the program must set the DS:DX registers to point to the beginning of a data buffer large enough to hold the requested transfer, put the file handle in BX, and put the number of bytes to be read in CX. The length requested can be a maximum of 65535 bytes. The program requesting the read operation is responsible for providing the data buffer.

If the read operation succeeds, the data is read, beginning at the current position of the file read/write pointer, to the specified location in memory. MS-DOS then increments its internal read/write pointer for the file by the length of the data transferred and returns the length to the calling program in AX with the carry flag cleared. The only indication that the end of the file has been reached is that the length returned is less than the length requested. In contrast, when Function 3FH is used to read from a character device that is *not* in raw mode, the read will terminate at the requested length or at the receipt of a carriage return character, whichever comes first. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output. If the read operation fails, MS-DOS returns with the carry flag set and an error code in AX.

Function 40H (Write File or Device) writes from a buffer to a file (or device) using a handle previously obtained from an open or create operation. Before calling Function 40H, the program must set DS:DX to point to the beginning of the buffer containing the source data, put the file handle in BX, and put the number of bytes to write in CX. The number of bytes to write can be a maximum of 65535.

If the write operation is successful, MS-DOS puts the number of bytes written in AX and increments the read/write pointer by this value; if the write operation fails, MS-DOS sets the carry flag and returns an error code in AX.

Records smaller than one sector (512 bytes) are not written directly to disk. Instead, MS-DOS stores the record in an internal buffer and writes it to disk when the internal buffer is full, when the file is closed, or when a call to Interrupt 21H Function 0DH (Disk Reset) is issued.

Note: If the destination of the write operation is a disk file and the disk is full, the only indication to the calling program is that the length returned in AX is not the same as the length requested in CX. *Disk full* is not returned as an error with the carry flag set.

A special use of the Write function is to truncate or extend a file. If Function 40H is called with a record length of zero in CX, the file size will be adjusted to the current location of the file read/write pointer.

Example: Open the file MYFILE.DAT, create the file MYFILE.BAK, copy the contents of the .DAT file into the .BAK file using 512-byte reads and writes, and then close both files.

```

file1 db 'MYFILE.DAT',0
file2 db 'MYFILE.BAK',0

handle1 dw ? ; handle for MYFILE.DAT
handle2 dw ? ; handle for MYFILE.BAK

buff db 512 dup (?) ; buffer for file I/O
.
.
.
; open MYFILE.DAT...
mov dx,seg file1 ; DS:DX = address of filename
mov ds,dx
mov dx,offset file1
mov ax,3d00h ; Function 3DH = open (read-only)
int 21h ; transfer to MS-DOS
jc error ; jump if open failed
mov handle1,ax ; save handle for file

; create MYFILE.BAK...
mov dx,offset file2 ; DS:DX = address of filename
mov cx,0 ; CX = normal attribute
mov ah,3ch ; Function 3CH = create
int 21h ; transfer to MS-DOS
jc error ; jump if create failed
mov handle2,ax ; save handle for file

loop: ; read MYFILE.DAT
mov dx,offset buff ; DS:DX = buffer address
mov cx,512 ; CX = length to read
mov bx,handle1 ; BX = handle for MYFILE.DAT
mov ah,3fh ; Function 3FH = read
int 21h ; transfer to MS-DOS
jc error ; jump if read failed
or ax,ax ; were any bytes read?
jz done ; no, end of file reached

; write MYFILE.BAK
mov dx,offset buff ; DS:DX = buffer address
mov cx,ax ; CX = length to write
mov bx,handle2 ; BX = handle for MYFILE.BAK
mov ah,40h ; Function 40H = write
int 21h ; transfer to MS-DOS
jc error ; jump if write failed
cmp ax,cx ; was write complete?
jne error ; jump if disk full
jmp loop ; continue to end of file

```

(more)

```

done:
    mov     bx,handle1      ; now close files...
    mov     ah,3eh         ; handle for MYFILE.DAT
    int     21h           ; Function 3EH = close file
    jc     error          ; transfer to MS-DOS
                                ; jump if close failed

    mov     bx,handle2    ; handle for MYFILE.BAK
    mov     ah,3eh         ; Function 3EH = close file
    int     21h           ; transfer to MS-DOS
    jc     error          ; jump if close failed

```

Positioning the read/write pointer

Function 42H (Move File Pointer) sets the position of the read/write pointer associated with a given handle. The function is called with a signed 32-bit offset in the CX and DX registers (the most significant half in CX), the file handle in BX, and the positioning mode in AL:

Mode	Significance
00	Supplied offset is relative to beginning of file.
01	Supplied offset is relative to current position of read/write pointer.
02	Supplied offset is relative to end of file.

If Function 42H succeeds, MS-DOS returns the resulting absolute offset (in bytes) of the file pointer relative to the beginning of the file in the DX and AX registers, with the most significant half in DX; if the function fails, MS-DOS sets the carry flag and returns an error code in AX.

Thus, a program can obtain the size of a file by calling Function 42H with an offset of zero and a positioning mode of 2. The function returns a value in DX:AX that represents the offset of the end-of-file position relative to the beginning of the file.

Example: Assume that the file MYFILE.DAT was previously opened and its handle is saved in the variable *fhandle*. Position the file pointer 32768 bytes from the beginning of the file and then read 512 bytes of data starting at that file position.

```

fhandle dw    ?           ; handle from previous open
buff    db    512 dup (?) ; buffer for data from file

```

(more)

```

mov     cx,0           ; position the file pointer...
mov     dx,32768      ; CX = high part of file offset
mov     bx,fhandle    ; DX = low part of file offset
mov     bx,fhandle    ; BX = handle for file
mov     al,0         ; AL = positioning mode
mov     ah,42h       ; Function 42H = position
int     21h         ; transfer to MS-DOS
jc     error        ; jump if function call failed

; now read 512 bytes from file
mov     dx,offset buff ; DS:DX = address of buffer
mov     cx,512       ; CX = length of 512 bytes
mov     bx,fhandle    ; BX = handle for file
mov     ah,3fh       ; Function 3FH = read
int     21h         ; transfer to MS-DOS
jc     error        ; jump if read failed
cmp     ax,512       ; was 512 bytes read?
jne     error        ; jump if partial rec. or EOF
.
.
.

```

Example: Assume that the file MYFILE.DAT was previously opened and its handle is saved in the variable *fhandle*. Find the size of the file in bytes by positioning the file pointer to zero bytes relative to the end of the file. The returned offset, which is relative to the beginning of the file, is the file's size.

```

fhandle dw    ?           ; handle from previous open
.
.
.
; position the file pointer
; to the end of file...
mov     cx,0           ; CX = high part of offset
mov     dx,0           ; DX = low part of offset
mov     bx,fhandle    ; BX = handle for file
mov     al,2         ; AL = positioning mode
mov     ah,42h       ; Function 42H = position
int     21h         ; transfer to MS-DOS
jc     error        ; jump if function call failed

; if call succeeded, DX:AX
; now contains the file size
.
.
.

```

Other handle operations

MS-DOS provides other handle-oriented functions to rename (or move) a file, delete a file, read or change a file's attributes, read or change a file's date and time stamp, and duplicate or redirect a file handle. The first three of these are "file-handle-like" because they use an ASCII string to specify the file; however, they do not return a file handle.

Renaming a file

Function 56H (Rename File) renames an existing file and/or moves the file from one location in the hierarchical file structure to another. The file to be renamed cannot be a hidden or system file or a subdirectory and must not be currently open by any process; attempting to rename an open file can corrupt the disk. MS-DOS renames a file by simply changing its directory entry; it moves a file by removing its current directory entry and creating a new entry in the target directory that refers to the same file. The location of the file's actual data on the disk is not changed.

Both the current and the new filenames must be ASCII strings and can include a drive and path specification; wildcard characters (* and ?) are not permitted in the filenames. The program calls Function 56H with the address of the current pathname in the DS:DX registers and the address of the new pathname in ES:DI. If the path elements of the two strings are not the same and both paths are valid, the file "moves" from the source directory to the target directory. If the paths match but the filenames differ, MS-DOS simply modifies the directory entry to reflect the new filename.

If the function succeeds, MS-DOS returns to the calling program with the carry flag clear. The function fails if the new filename is already in the target directory; in that case, MS-DOS sets the carry flag and returns an error code in AX.

Example: Change the name of the file MYFILE.DAT to MYFILE.OLD. In the same operation, move the file from the \WORK directory to the \BACKUP directory.

```
file1 db    '\WORK\MYFILE.DAT',0
file2 db    '\BACKUP\MYFILE.OLD',0
.
.
.
mov    dx,seg file1    ; DS:DX = old filename
mov    ds,dx
mov    es,dx
mov    dx,offset file1
mov    di,offset file2 ; ES:DI = new filename
mov    ah,56h         ; Function 56H = rename
int    21h           ; transfer to MS-DOS
jc     error         ; jump if rename failed
.
.
.
```

Deleting a file

Function 41H (Delete File) effectively deletes a file from a disk. Before calling the function, a program must set the DS:DX registers to point to the ASCII pathname of the file to be deleted. The supplied pathname cannot specify a subdirectory or a read-only file, and the file must not be currently open by any process.

If the function is successful, MS-DOS deletes the file by simply marking the first byte of its directory entry with a special character (0E5H), making the entry subsequently unrecognizable. MS-DOS then updates the disk's FAT so that the clusters that previously belonged to the file are "free" and returns to the program with the carry flag clear. If the delete function fails, MS-DOS sets the carry flag and returns an error code in AX.

The actual contents of the clusters assigned to the file are not changed by a delete operation, so for security reasons sensitive information should be overwritten with spaces or some other constant character before the file is deleted with Function 41H.

Example: Delete the file MYFILE.DAT, located in the \WORK directory on drive C.

```
fname db 'C:\WORK\MYFILE.DAT',0
      .
      .
      .
      mov dx,seg fname ; DS:DX = address of filename
      mov ds,dx
      mov dx,offset fname
      mov ah,41h ; Function 41H = delete
      int 21h ; transfer to MS-DOS
      jc error ; jump if delete failed
      .
      .
      .
```

Getting/setting file attributes

Function 43H (Get/Set File Attributes) obtains or modifies the attributes of an existing file. Before calling Function 43H, the program must set the DS:DX registers to point to the ASCII pathname for the file. To read the attributes, the program must set AL to zero; to set the attributes, it must set AL to 1 and place an attribute code in CX. See *Creating a File* above.

If the function is successful, MS-DOS reads or sets the attribute byte in the file's directory entry and returns with the carry flag clear and the file's attribute in CX. If the function fails, MS-DOS sets the carry flag and returns an error code in AX.

Function 43H cannot be used to set the volume-label bit (bit 3) or the subdirectory bit (bit 4) of a file. It also should not be used on a file that is currently open by any process.

Example: Change the attributes of the file MYFILE.DAT in the \BACKUP directory on drive C to read-only. This prevents the file from being accidentally deleted from the disk.

```
fname db 'C:\BACKUP\MYFILE.DAT',0
      .
      .
      .
      mov dx,seg fname ; DS:DX = address of filename
      mov ds,dx
      mov dx,offset fname
      mov cx,1 ; CX = attribute (read-only)
      mov al,1 ; AL = mode (0 = get, 1 = set)
```

(more)

```
mov    ah,43h        ; Function 43H = get/set attr
int    21h           ; transfer to MS-DOS
jc     error        ; jump if set attrib. failed
.
```

Getting/setting file date and time

Function 57H (Get/Set Date/Time of File) reads or sets the directory time and date stamp of an open file. To set the time and date to a particular value, the program must call Function 57H with the desired time in CX, the desired date in DX, the handle for the file (obtained from a previous open or create operation) in BX, and the value 1 in AL. To read the time and date, the function is called with AL containing 0 and the file handle in BX; the time is returned in the CX register and the date is returned in the DX register. As with other handle-oriented file functions, if the function succeeds, the carry flag is returned cleared; if the function fails, MS-DOS returns the carry flag set and an error code in AX.

The formats used for the file time and date are the same as those used in disk directory entries and FCBs. See Structure of the File-Control Block below.

The main uses of Function 57H are to force the time and date entry for a file to be updated when the file has *not* been changed and to circumvent MS-DOS's modification of a file date and time when the file *has* been changed. In the latter case, a program can use this function with AL = 0 to obtain the file's previous date and time stamp, modify the file, and then restore the original file date and time by re-calling the function with AL = 1 before closing the file.

Duplicating and redirecting handles

Ordinarily, the disk FAT and directory are not updated until a file is closed, even when the file has been modified. Thus, until the file is closed, any new data added to the file can be lost if the system crashes or is turned off unexpectedly. The obvious defense against such loss is simply to close and reopen the file every time the file is changed. However, this is a relatively slow procedure and in a network environment can cause the program to lose control of the file to another process.

Use of a second file handle, created by using Function 45H (Duplicate File Handle) to duplicate the original handle of the file to be updated, can protect data added to a disk file before the file is closed. To use Function 45H, the program must put the handle to be duplicated in BX. If the operation is successful, MS-DOS clears the carry flag and returns the new handle in AX; if the operation fails, MS-DOS sets the carry flag and returns an error code in AX.

If the function succeeds, the duplicate handle can simply be closed in the usual manner with Function 3EH. This forces the desired update of the disk directory and FAT. The original handle remains open and the program can continue to use it for file read and write operations.

Note: While the second handle is open, moving the read/write pointer associated with either handle moves the pointer associated with the other.

Example: Assume that the file MYFILE.DAT was previously opened and the handle for that file has been saved in the variable *fhandle*. Duplicate the handle and then close the duplicate to ensure that any data recently written to the file is saved on the disk and that the directory entry for the file is updated accordingly.

```
fhandle dw      ?           ; handle from previous open
.
.
.
                                ; duplicate the handle...
mov     bx,fhandle           ; BX = handle for file
mov     ah,45h              ; Function 45H = dup handle
int     21h                 ; transfer to MS-DOS
jc      error               ; jump if function call failed

                                ; now close the new handle...
mov     bx,ax                ; BX = duplicated handle
mov     ah,3eh              ; Function 3EH = close
int     21h                 ; transfer to MS-DOS
jc      error               ; jump if close failed
mov     bx,fhandle           ; replace closed handle with active handle
.
.
.
```

Function 45H is sometimes also used in conjunction with Function 46H (Force Duplicate File Handle). Function 46H forces a handle to be a duplicate for another open handle—in other words, to refer to the same file or device at the same file read/write pointer location. The handle is then said to be redirected.

The most common use of Function 46H is to change the meaning of the standard input and standard output handles before loading a child process with the EXEC function. In this manner, the input for the child program can be redirected to come from a file or its output can be redirected into a file, without any special knowledge on the part of the child program. In such cases, Function 45H is used to also create duplicates of the standard input and standard output handles before they are redirected, so that their original meanings can be restored after the child exits. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Writing MS-DOS Filters.

Using the FCB Functions

A file control block is a data structure, located in the application program's memory space, that contains relevant information about an open disk file: the disk drive, the filename and extension, a pointer to a position within the file, and so on. Each open file must have its own FCB. The information in an FCB is maintained cooperatively by both MS-DOS and the application program.

MS-DOS moves data to and from a disk file associated with an FCB by means of a data buffer called the disk transfer area (DTA). The current address of the DTA is under the control of the application program, although each program has a 128-byte default DTA at offset 80H in its program segment prefix (PSP). See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

Under early versions of MS-DOS, the only limit on the number of files that can be open simultaneously with FCBs is the amount of memory available to the application to hold the FCBs and their associated disk buffers. However, under MS-DOS versions 3.0 and later, when file-sharing support (SHARE.EXE) is loaded, MS-DOS places some restrictions on the use of FCBs to simplify the job of maintaining network connections for files. If the application attempts to open too many FCBs, MS-DOS simply closes the least recently used FCBs to keep the total number within a limit.

The CONFIG.SYS file directive FCBS allows the user to control the allowed maximum number of FCBs and to specify a certain number of FCBs to be protected against automatic closure by the system. The default values are a maximum of four files open simultaneously using FCBs and zero FCBs protected from automatic closure by the system. See USER COMMANDS: CONFIG.SYS: FCBS.

Because the FCB operations predate MS-DOS version 2.0 and because FCBs have a fixed structure with no room to contain a path, the FCB file and record services do not support the hierarchical directory structure. Many FCB operations can be performed only on files in the current directory of a disk. For this reason, the use of FCB file and record operations should be avoided in new programs.

Structure of the file control block

Each FCB is a 37-byte array allocated from its own memory space by the application program that will use it. The FCB contains all the information needed to identify a disk file and access the data within it: drive identifier, filename, extension, file size, record size, various file pointers, and date and time stamps. The FCB structure is shown in Table 7-3.

Table 7-3. Structure of a Normal File Control Block.

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	00H	1	Drive identifier
Program	01H	8	Filename
Program	09H	3	File extension
MS-DOS	0CH	2	Current block number
Program	0EH	2	Record size (bytes)
MS-DOS	10H	4	File size (bytes)
MS-DOS	14H	2	Date stamp
MS-DOS	16H	2	Time stamp
MS-DOS	18H	8	Reserved
MS-DOS	20H	1	Current record number
Program	21H	4	Random record number

Drive identifier: Initialized by the application to designate the drive on which the file to be opened or created resides. 0 = default drive, 1 = drive A, 2 = drive B, and so on. If the application supplies a zero in this byte (to use the default drive), MS-DOS alters the byte during the open or create operation to reflect the actual drive used; that is, after an open or create operation, this drive will always contain a value of 1 or greater.

Filename: Standard eight-character filename; initialized by the application; must be left justified and padded with blanks if the name has fewer than eight characters. A device name (for example, PRN) can be used; note that there is no colon after a device name.

File extension: Three-character file extension; initialized by the application; must be left justified and padded with blanks if the extension has fewer than three characters.

Current block number: Initialized to zero by MS-DOS when the file is opened. The block number and the record number together make up the record pointer during sequential file access.

Record size: The size of a record (in bytes) as used by the program. MS-DOS sets this field to 128 when the file is opened or created; the program can modify the field afterward to any desired record size. If the record size is larger than 128 bytes, the default DTA in the PSP cannot be used because it will collide with the program's own code or data.

File size: The size of the file in bytes. MS-DOS initializes this field from the file's directory entry when the file is opened. The first 2 bytes of this 4-byte field are the least significant bytes of the file size.

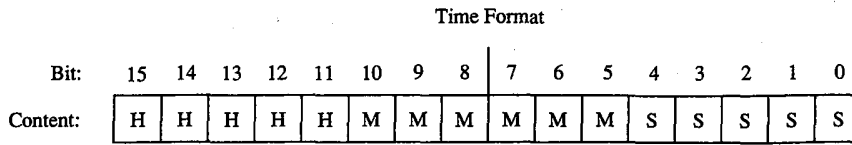
Date stamp: The date of the last write operation on the file. MS-DOS initializes this field from the file's directory entry when the file is opened. This field uses the same format used by file handle Function 57H (Get/Set/Date/Time of File):

Date Format

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Content:	Y	Y	Y	Y	Y	Y	Y	M	M	M	M	M	D	D	D	D

Bits	Contents
0-4	Day of month (1-31)
5-8	Month (1-12)
9-15	Year (relative to 1980)

Time stamp: The time of the last write operation on the file. MS-DOS initializes this field from the file's directory entry when the file is opened. This field uses the same format used by file handle Function 57H (Get/Set/Date/Time of File):



Bits	Contents
0-4	Number of 2-second increments (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)

Current record number: Together with the block number, constitutes the record pointer used during sequential read and write operations. MS-DOS does not initialize this field when a file is opened. The record number is limited to the range 0 through 127; thus, there are 128 records per block. The beginning of a file is record 0 of block 0.

Random record pointer: A 4-byte field that identifies the record to be transferred by the random record functions 21H, 22H, 27H, and 28H. If the record size is 64 bytes or larger, only the first 3 bytes of this field are used. MS-DOS updates this field after random block reads and writes (Functions 27H and 28H) but not after random record reads and writes (Functions 21H and 22H).

An extended FCB, which is 7 bytes longer than a normal FCB, can be used to access files with special attributes such as hidden, system, and read-only. The extra 7 bytes of an extended FCB are simply prefixed to the normal FCB format (Table 7-4). The first byte of an extended FCB always contains 0FFH, which could never be a legal drive code and therefore serves as a signal to MS-DOS that the extended format is being used. The next 5 bytes are reserved and must be zero, and the last byte of the prefix specifies the attributes of the file being manipulated. The remainder of an extended FCB has exactly the same layout as a normal FCB. In general, an extended FCB can be used with any MS-DOS function call that accepts a normal FCB.

Table 7-4. Structure of an Extended File Control Block.

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	00H	1	Extended FCB flag = 0FFH
MS-DOS	01H	5	Reserved
Program	06H	1	File attribute byte
Program	07H	1	Drive identifier
Program	08H	8	Filename

(more)

Table 7-4. *Continued.*

Maintained by	Offset (bytes)	Size (bytes)	Description
Program	10H	3	File extension
MS-DOS	13H	2	Current block number
Program	15H	2	Record size (bytes)
MS-DOS	17H	4	File size (bytes)
MS-DOS	1BH	2	Date stamp
MS-DOS	1DH	2	Time stamp
MS-DOS	1FH	8	Reserved
MS-DOS	27H	1	Current record number
Program	28H	4	Random record number

Extended FCB flag: When 0FFH is present in the first byte of an FCB, it is a signal to MS-DOS that an extended FCB (44 bytes) is being used instead of a normal FCB (37 bytes).

File attribute byte: Must be initialized by the application when an extended FCB is used to open or create a file. The bits of this field have the following significance:

Bit	Meaning
0	Read-only
1	Hidden
2	System
3	Volume label
4	Directory
5	Archive
6	Reserved
7	Reserved

FCB functions and the PSP

The PSP contains several items that are of interest when using the FCB file and record operations: two FCBs called the default FCBs, the default DTA, and the command tail for the program. The following table shows the size and location of these elements:

PSP Offset (bytes)	Size (bytes)	Description
5CH	16	Default FCB #1
6CH	20	Default FCB #2
80H	1	Length of command tail
81H	127	Command-tail text
80H	128	Default disk transfer area (DTA)

When MS-DOS loads a program into memory for execution, it copies the command tail into the PSP at offset 81H, places the length of the command tail in the byte at offset 80H, and parses the first two parameters in the command tail into the default FCBs at PSP offsets 5CH and 6CH. (The command tail consists of the command line used to invoke the program minus the program name itself and any redirection or piping characters and their associated filenames or device names.) MS-DOS then sets the initial DTA address for the program to PSP:0080H.

For several reasons, the default FCBs and the DTA are often moved to another location within the program's memory area. First, the default DTA allows processing of only very small records. In addition, the default FCBs overlap substantially, and the first byte of the default DTA and the last byte of the first FCB conflict. Finally, unless either the command tail or the DTA is moved beforehand, the first FCB-related file or record operation will destroy the command tail.

Function 1AH (Set DTA Address) is used to alter the DTA address. It is called with the segment and offset of the new buffer to be used as the DTA in DS:DX. The DTA address remains the same until another call to Function 1AH, regardless of other file and record management calls; it does not need to be reset before each read or write.

Note: A program can use Function 2FH (Get DTA Address) to obtain the current DTA address before changing it, so that the original address can be restored later.

Parsing the filename

Before a file can be opened or created with the FCB function calls, its drive, filename, and extension must be placed within the proper fields of the FCB. The filename can be coded into the program itself, or the program can obtain it from the command tail in the PSP or by prompting the user and reading it in with one of the several function calls for character device input.

MS-DOS automatically parses the first two parameters in the program's command tail into the default FCBs at PSP:005CH and PSP:006CH. It does not, however, attempt to differentiate between switches and filenames, so the pre-parsed FCBs are not necessarily useful to the application program. If the filenames were preceded by any switches, the program itself has to extract the filenames directly from the command tail. The program is then responsible for determining which parameters are switches and which are filenames, as well as where each parameter begins and ends.

After a filename has been located, Function 29H (Parse Filename) can be used to test it for invalid characters and separators and to insert its various components into the proper fields in an FCB. The filename must be a string in the standard form *drive:filename.ext*. Wildcard characters are permitted in the filename and/or extension; asterisk (*) wildcards are expanded to question mark (?) wildcards.

To call Function 29H, the DS:SI registers must point to the candidate filename, ES:DI must point to the 37-byte buffer that will become the FCB for the file, and AL must hold the parsing control code. See SYSTEM CALLS: INTERRUPT 21H: Function 29H.

If a drive code is not included in the filename, MS-DOS inserts the drive number of the current drive into the FCB. Parsing stops at the first terminator character encountered in the filename. Terminators include the following:

; , = + / " [] | < > | space tab

If a colon character (:) is not in the proper position to delimit the disk drive identifier or if a period (.) is not in the proper position to delimit the extension, the character will also be treated as a terminator. For example, the filename C:MEMO.TXT will be parsed correctly; however, ABC:DEF.DAY will be parsed as ABC.

If an invalid drive is specified in the filename, Function 29H returns 0FFH in AL; if the filename contains any wildcard characters, it returns 1. Otherwise, AL contains zero upon return, indicating a valid, unambiguous filename.

Note that this function simply parses the filename into the FCB. It does not initialize any other fields of the FCB (although it does zero the current block and record size fields), and it does not test whether the specified file actually exists.

Error handling and FCB functions

The FCB-related file and record functions do not return much in the way of error information when a function fails. Typically, an FCB function returns a zero in AL if the function succeeded and 0FFH if the function failed. Under MS-DOS versions 2.x, the program is left to its own devices to determine the cause of the error. Under MS-DOS versions 3.x, however, a failed FCB function call can be followed by a call to Interrupt 21H Function 59H (Get Extended Error Information). Function 59H will return the same descriptive codes for the error, including the error locus and a suggested recovery strategy, as would be returned for the counterpart handle-oriented file or record function.

Creating a file

Function 16H (Create File with FCB) creates a new file and opens it for subsequent read/write operations. The function is called with DS:DX pointing to a valid, unopened FCB. MS-DOS searches the current directory for the specified filename. If the filename is found, MS-DOS sets the file length to zero and opens the file, effectively truncating it to a zero-length file; if the filename is not found, MS-DOS creates a new file and opens it. Other fields of the FCB are filled in by MS-DOS as described below under Opening a File.

If the create operation succeeds, MS-DOS returns zero in AL; if the operation fails, it returns 0FFH in AL. This function will not ordinarily fail unless the file is being created in the root directory and the directory is full.

Warning: To avoid loss of existing data, the FCB open function should be used to test for file existence before creating a file.

Opening a file

Function 0FH opens an existing file. DS:DX must point to a valid, unopened FCB containing the name of the file to be opened. If the specified file is found in the current directory, MS-DOS opens the file, fills in the FCB as shown in the list below, and returns with AL set to 00H; if the file is not found, MS-DOS returns with AL set to 0FFH, indicating an error.

When the file is opened, MS-DOS

- Sets the drive identifier (offset 00H) to the actual drive (01 = A, 02 = B, and so on).
- Sets the current block number (offset 0CH) to zero.
- Sets the file size (offset 10H) to the value found in the directory entry for the file.
- Sets the record size (offset 0EH) to 128.
- Sets the date and time stamp (offsets 14H and 16H) to the values found in the directory entry for the file.

The program may need to adjust the FCB—change the record size and the random record pointer, for example—before proceeding with record operations.

Example: Display a prompt and accept a filename from the user. Parse the filename into an FCB, checking for an illegal drive identifier or the presence of wildcards. If a valid, unambiguous filename has been entered, attempt to open the file. Create the file if it does not already exist.

```

kbuf    db        64,0,64 dup (0)
prompt  db        0dh,0ah,'Enter filename: $'
myfcb   db        37 dup (0)

.
.
.

                                ; display the prompt...
mov     dx,seg prompt           ; DS:DX = prompt address
mov     ds,dx
mov     es,dx
mov     dx,offset prompt
mov     ah,09h                 ; Function 09H = print string
int     21h                    ; transfer to MS-DOS

                                ; now input filename...
mov     dx,offset kbuf         ; DS:DX = buffer address
mov     ah,0ah                 ; Function 0AH = enter string
int     21h                    ; transfer to MS-DOS

                                ; parse filename into FCB...
mov     si,offset kbuf+2      ; DS:SI = address of filename
mov     di,offset myfcb       ; ES:DI = address of fcb
mov     ax,2900h              ; Function 29H = parse name
int     21h                    ; transfer to MS-DOS
or     al,al                   ; jump if bad drive or
jnz     error                  ; wildcard characters in name

```

(more)

```

                                ; try to open file...
mov     dx,offset myfcb ; DS:DX = FCB address
mov     ah,0fh          ; Function 0FH = open file
int     21h             ; transfer to MS-DOS
or      al,al           ; check status
jz      proceed        ; jump if open successful

                                ; else create file...
mov     dx,offset myfcb ; DS:DX = FCB address
mov     ah,16h          ; Function 16H = create
int     21h             ; transfer to MS-DOS
or      al,al           ; did create succeed?
jnz     error          ; jump if create failed

proceed:
.
.
.
                                ; file has been opened or
                                ; created, and FCB is valid
                                ; for read/write operations...

```

Closing a file

Function 10H (Close File with FCB) closes a file previously opened with an FCB. As usual, the function is called with DS:DX pointing to the FCB of the file to be closed. MS-DOS updates the directory, if necessary, to reflect any changes in the file's size and the date and time last written.

If the operation succeeds, MS-DOS returns 00H in AL; if the operation fails, MS-DOS returns 0FFH.

Reading and writing files with FCBs

MS-DOS offers a choice of three FCB access methods for data within files: sequential, random record, and random block.

Sequential operations step through the file one record at a time. MS-DOS increments the current record and current block numbers after each file access so that they point to the beginning of the next record. This method is particularly useful for copying or listing files.

Random record access allows the program to read or write a record from any location in the file, without sequentially reading all records up to that point in the file. The program must set the random record number field of the FCB appropriately before the read or write is requested. This method is useful in database applications, in which a program must manipulate fixed-length records.

Random block operations combine the features of sequential and random record access methods. The program can set the record number to point to any record within a file, and MS-DOS updates the record number after a read or write operation. Thus, sequential operations can easily be initiated at any file location. Random block operations with a record length of 1 byte simulate file-handle access methods.

All three methods require that the FCB for the file be open, that DS:DX point to the FCB, that the DTA be large enough for the specified record size, and that the DTA address be previously set with Function 1AH if the default DTA in the program's PSP is not being used.

MS-DOS reports the success or failure of any FCB-related read operation (sequential, random record, or random block) with one of four return codes in register AL:

Code	Meaning
00H	Successful read
01H	End of file reached; no data read into DTA
02H	Segment wrap (DTA too close to end of segment); no data read into DTA
03H	End of file reached; partial record read into DTA

MS-DOS reports the success or failure of an FCB-related write operation as one of three return codes in register AL:

Code	Meaning
00H	Successful write
01H	Disk full; partial or no write
02H	Segment wrap (DTA too close to end of segment); write failed

For FCB write operations, records smaller than one sector (512 bytes) are not written directly to disk. Instead, MS-DOS stores the record in an internal buffer and writes the data to disk only when the internal buffer is full, when the file is closed, or when a call to Interrupt 21H Function 0DH (Disk Reset) is issued.

Sequential access: reading

Function 14H (Sequential Read) reads records sequentially from the file to the current DTA address, which must point to an area at least as large as the record size specified in the file's FCB. After each read operation, MS-DOS updates the FCB block and record numbers (offsets 0CH and 20H) to point to the next record.

Sequential access: writing

Function 15H (Sequential Write) writes records sequentially from memory into the file. The length written is specified by the record size field (offset 0EH) in the FCB; the memory address of the record to be written is determined by the current DTA address. After each sequential write operation, MS-DOS updates the FCB block and record numbers (offsets 0CH and 20H) to point to the next record.

Random record access: reading

Function 21H (Random Read) reads a specific record from a file. Before requesting the read operation, the program specifies the record to be transferred by setting the record size and random record number fields of the FCB (offsets 0EH and 21H). The current DTA address must also have been previously set with Function 1AH to point to a buffer of adequate size if the default DTA is not large enough.

After the read, MS-DOS sets the current block and current record number fields (offsets 0CH and 20H) to point to the same record. Thus, the program is set up to change to sequential reads or writes. However, if the program wants to continue with random record access, it must continue to update the random record field of the FCB before each random record read or write operation.

Random record access: writing

Function 22H (Random Write) writes a specific record from memory to a file. Before issuing the function call, the program must ensure that the record size and random record pointer fields at FCB offsets 0EH and 21H are set appropriately and that the current DTA address points to the buffer containing the data to be written.

After the write, MS-DOS sets the current block and current record number fields (offsets 0CH and 20H) to point to the same record. Thus, the program is set up to change to sequential reads or writes. If the program wants to continue with random record access, it must continue to update the random record field of the FCB before each random record read or write operation.

Random block access: reading

Function 27H (Random Block Read) reads a block of consecutive records. Before issuing the read request, the program must specify the file location of the first record by setting the record size and random record number fields of the FCB (offsets 0EH and 21H) and must put the number of records to be read in CX. The DTA address must have already been set with Function 1AH to point to a buffer large enough to contain the group of records to be read if the default DTA was not large enough. The program can then issue the Function 27H call with DS:DX pointing to the FCB for the file.

After the random block read operation, MS-DOS resets the FCB random record pointer (offset 21H) and the current block and current record number fields (offsets 0CH and 20H) to point to the beginning of the next record not read and returns the number of records actually read in CX.

If the record size is set to 1 byte, Function 27H reads the number of bytes specified in CX, beginning with the byte position specified in the random record pointer. This simulates (to some extent) the handle type of read operation (Function 3FH).

Random block access: writing

Function 28H (Random Block Write) writes a block of consecutive records from memory to disk. The program specifies the file location of the first record to be written by setting the record size and random record pointer fields in the FCB (offsets 0EH and 21H). If the default DTA is not being used, the program must also ensure that the current DTA address is set appropriately by a previous call to Function 1AH. When Function 28H is called, DS:DX must point to the FCB for the file and CX must contain the number of records to be written.

After the random block write operation, MS-DOS resets the FCB random record pointer (offset 21H) and the current block and current record number fields (offsets 0CH and 20H) to point to the beginning of the next block of data and returns the number of records actually written in CX.

If the record size is set to 1 byte, Function 28H writes the number of bytes specified in CX, beginning with the byte position specified in the random record pointer. This simulates (to some extent) the handle type of write operation (Function 40H).

Calling Function 28H with a record count of zero in register CX causes the file length to be extended or truncated to the current value in the FCB random record pointer field (offset 21H) multiplied by the contents of the record size field (offset 0EH).

Example: Open the file MYFILE.DAT and create the file MYFILE.BAK on the current disk drive, copy the contents of the .DAT file into the .BAK file using 512-byte reads and writes, and then close both files.

```

fcb1  db      0          ; drive = default
      db      'MYFILE '  ; 8 character filename
      db      'DAT'     ; 3 character extension
      db      25 dup (0) ; remainder of fcb1
fcb2  db      0          ; drive = default
      db      'MYFILE '  ; 8 character filename
      db      'BAK'     ; 3 character extension
      db      25 dup (0) ; remainder of fcb2
buff  db      512 dup (?) ; buffer for file I/O
      .
      .
      .
      ; open MYFILE.DAT...
mov    dx,seg fcb1      ; DS:DX = address of FCB
mov    ds,dx
mov    dx,offset fcb1
mov    ah,0fh          ; Function 0FH = open
int    21h            ; transfer to MS-DOS
or     al,al          ; did open succeed?
jnz    error          ; jump if open failed
      ; create MYFILE.BAK...
mov    dx,offset fcb2  ; DS:DX = address of FCB
mov    ah,16h         ; Function 16H = create
int    21h            ; transfer to MS-DOS
or     al,al          ; did create succeed?
jnz    error          ; jump if create failed
      ; set record length to 512
mov    word ptr fcb1+0eh,512
mov    word ptr fcb2+0eh,512
      ; set DTA to our buffer...
mov    dx,offset buff  ; DS:DX = buffer address
mov    ah,1ah         ; Function 1AH = set DTA
int    21h            ; transfer to MS-DOS
loop:  ; read MYFILE.DAT
mov    dx,offset fcb1  ; DS:DX = FCB address
mov    ah,14h         ; Function 14H = seq. read
int    21h            ; transfer to MS-DOS
or     al,al          ; was read successful?
jnz    done           ; no, quit
      ; write MYFILE.BAK...

```

(more)

```

mov     dx,offset fcb2 ; DS:DX = FCB address
mov     ah,15h         ; Function 15H = seq. write
int     21h           ; transfer to MS-DOS
or      al,al         ; was write successful?
jnz     error         ; jump if write failed
jmp     loop          ; continue to end of file
done:
mov     dx,offset fcb1 ; DS:DX = FCB for MYFILE.DAT
mov     ah,10h        ; Function 10H = close file
int     21h           ; transfer to MS-DOS
or      al,al         ; did close succeed?
jnz     error         ; jump if close failed
mov     dx,offset fcb2 ; DS:DX = FCB for MYFILE.BAK
mov     ah,10h        ; Function 10H = close file
int     21h           ; transfer to MS-DOS
or      al,al         ; did close succeed?
jnz     error         ; jump if close failed

```

Other FCB file operations

As it does with file handles, MS-DOS provides FCB-oriented functions to rename or delete a file. Unlike the other FCB functions and their handle counterparts, these two functions accept wildcard characters. An additional FCB function allows the size or existence of a file to be determined without actually opening the file.

Renaming a file

Function 17H (Rename File) renames a file (or files) in the current directory. The file to be renamed cannot have the hidden or system attribute. Before calling Function 17H, the program must create a special FCB that contains the drive code at offset 00H, the old filename at offset 01H, and the new filename at offset 11H. Both the current and the new filenames can contain the ? wildcard character.

When the function call is made, DS:DX must point to the special FCB structure. MS-DOS searches the current directory for the old filename. If it finds the old filename, MS-DOS then searches for the new filename and, if it finds no matching filename, changes the directory entry for the old filename to reflect the new filename. If the old filename field of the special FCB contains any wildcard characters, MS-DOS renames every matching file. Duplicate filenames are not permitted; the process will fail at the first duplicate name.

If the operation is successful, MS-DOS returns zero in AL; if the operation fails, it returns 0FFH. The error condition may indicate either that no files were renamed or that at least one file was renamed but the operation was then terminated because of a duplicate filename.

Example: Rename all the files with the extension .ASM in the current directory of the default disk drive to have the extension .COD.

```

renfcb db      0           ; default drive
       db      '????????' ; wildcard filename
       db      'ASM'      ; old extension
       db      5 dup (0)  ; reserved area
       db      '????????' ; wildcard filename
       db      'COD'      ; new extension
       db      15 dup (0) ; remainder of FCB
       .
       .
       .
mov     dx,seg renfcb    ; DS:DX = address of
mov     ds,dx           ; "special" FCB
mov     dx,offset renfcb
mov     ah,17h         ; Function 17H = rename
int     21h            ; transfer to MS-DOS
or      al,al          ; did function succeed?
jnz     error          ; jump if rename failed
       .
       .
       .

```

Deleting a file

Function 13H (Delete File) deletes a file from the current directory. The file should not be currently open by any process. If the file to be deleted has special attributes, such as read-only, the program must use an extended FCB to remove the file. Directories cannot be deleted with this function, even with an extended FCB.

Function 13H is called with DS:DX pointing to an unopened, valid FCB containing the name of the file to be deleted. The filename can contain the ? wildcard character; if it does, MS-DOS deletes all files matching the specified name. If at least one file matches the FCB and is deleted, MS-DOS returns 00H in AL; if no matching filename is found, it returns 0FFH.

Note: This function, if it succeeds, does not return any information about which and how many files were deleted. When multiple files must be deleted, closer control can be exercised by using the Find File functions (Functions 11H and 12H) to inspect candidate filenames. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Disk Directories and Volume Labels. The files can then be deleted individually.

Example: Delete all the files in the current directory of the current disk drive that have the extension .BAK and whose filenames have A as the first character.

```

delfcb db      0           ; default drive
       db      'A???????' ; wildcard filename
       db      'BAK'      ; extension
       db      25 dup (0)  ; remainder of FCB

```

(more)

```

mov     dx,seg delfcb    ; DS:DX = FCB address
mov     ds,dx
mov     dx,offset delfcb
mov     ah,13h          ; Function 13H = delete
int     21h            ; transfer to MS-DOS
or      al,al          ; did function succeed?
jnz     error          ; jump if delete failed

```

Finding file size and testing for existence

Function 23H (Get File Size) is used primarily to find the size of a disk file without opening it, but it may also be used instead of Function 11H (Find First File) to simply test for the existence of a file. Before calling Function 23H, the program must parse the filename into an unopened FCB, initialize the record size field of the FCB (offset 0EH), and set the DS:DX registers to point to the FCB.

When Function 23H returns, AL contains 00H if the file was found in the current directory of the specified drive and 0FFH if the file was not found.

If the file was found, the random record field at FCB offset 21H contains the number of records (rounded upward) in the target file, in terms of the value in the record size field (offset 0EH) of the FCB. If the record size is at least 64 bytes, only the first 3 bytes of the random record field are used; if the record size is less than 64 bytes, all 4 bytes are used. To obtain the size of the file in bytes, the program must set the record size field to 1 before the call. This method is not any faster than simply opening the file, but it does avoid the overhead of closing the file afterward (which is necessary in a networking environment).

Summary

MS-DOS supports two distinct but overlapping sets of file and record management services. The handle-oriented functions operate in terms of null-terminated (ASCIIZ) filenames and 16-bit file identifiers, called handles, that are returned by MS-DOS after a file is opened or created. The filenames can include a full path specifying the file's location in the hierarchical directory structure. The information associated with a file handle, such as the current read/write pointer for the file, the date and time of the last write to the file, and the file's read/write permissions, sharing mode, and attributes, is maintained in a table internal to MS-DOS.

In contrast, the FCB-oriented functions use a 37-byte structure called a file control block, located in the application program's memory space, to specify the name and location of the file. After a file is opened or created, the FCB is used by both MS-DOS and the application to hold other information about the file, such as the current read/write file pointer, while that file is in use. Because FCBs predate the hierarchical directory structure that was introduced in MS-DOS version 2.0 and do not have room to hold the path for a file, the FCB functions cannot be used to access files that are not in the current directory of the specified drive.

In addition to their lack of support for pathnames, the FCB functions have much poorer error reporting capabilities than handle functions and are nearly useless in networking environments because they do not support file sharing and locking. Consequently, it is strongly recommended that the handle-related file and record functions be used exclusively in all new applications.

*Robert Byers
Code by Ray Duncan*

Article 8

Disk Directories and Volume Labels

MS-DOS, being a disk operating system, provides facilities for cataloging disk files. The data structure used by MS-DOS for this purpose is the directory, a linear list of names in which each name is associated with a physical location on the disk. Directories are accessed and updated implicitly whenever files are manipulated, but both directories and their contents can also be manipulated explicitly using several of the MS-DOS Interrupt 21H service functions.

MS-DOS versions 1.x support only one directory on each disk. Versions 2.0 and later, however, support multiple directories linked in a two-way, hierarchical tree structure (Figure 8-1), and the complete specification of the name of a file or directory thus must describe the location in the directory hierarchy in which the name appears. This specification, or path, is created by concatenating a disk drive specifier (for example, A: or C:), the

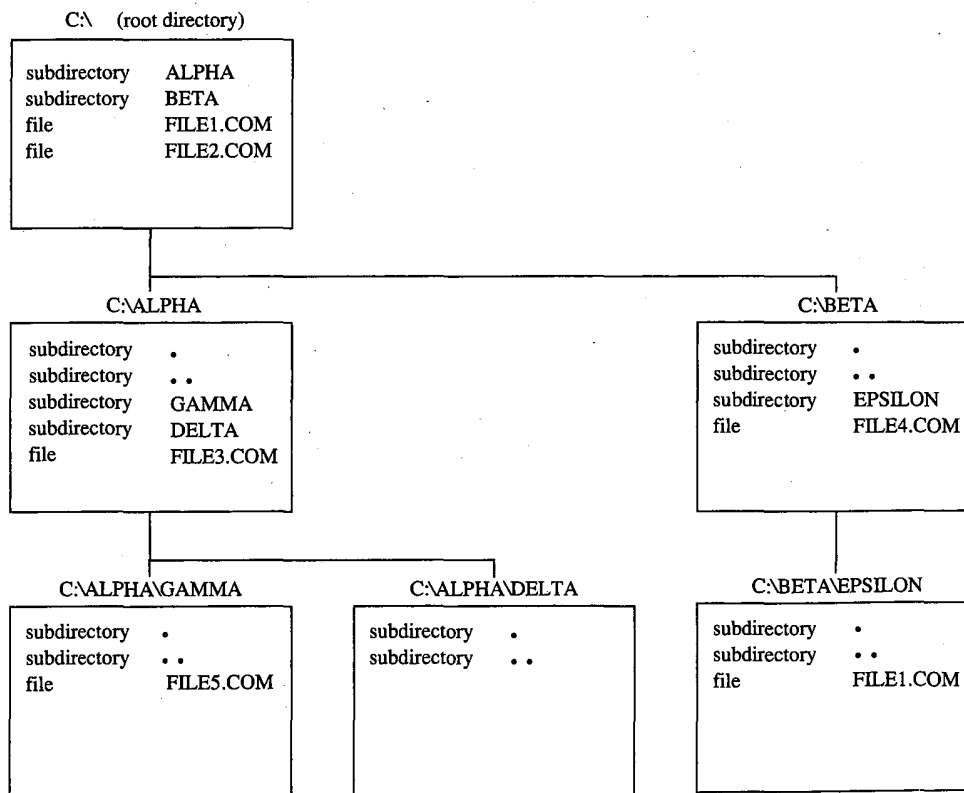


Figure 8-1. Typical hierarchical directory structure (MS-DOS versions 2.0 and later).

names of the directories in hierarchical order starting with the root directory, and finally the name of the file or directory. For example, in Figure 8-1, the complete pathname for FILE5.COM is C:\ALPHA\GAMMA\FILE5.COM. The two instances of FILE1.COM, in the root directory and in the directory EPSILON, are distinguished by their pathnames: C:\FILE1.COM in the first instance and C:\BETA\EPSILON\FILE1.COM in the second.

Note: If no drive is specified, the current drive is assumed. Also, if the first name in the specification is not preceded by a backslash, the specification is assumed to be relative to the current directory. For example, if the current directory is C:\BETA\EPSILON, the specification \FILE1.COM indicates the file FILE1.COM in the root directory and the specification FILE1.COM indicates the file FILE1.COM in the directory C:\BETA\EPSILON. See Figure 8-1.

Although the casual user of MS-DOS need not be concerned with how this hierarchical directory structure is implemented, MS-DOS programmers should be familiar with the internal structure of directories and with the Interrupt 21H functions available for manipulating directory contents and maintaining the links between directories. This article provides that information.

Logical Structure of MS-DOS Directories

An MS-DOS directory consists of a list of 32-byte directory entries, each of which contains a name and descriptive information. In MS-DOS versions 1.x, each name must be a filename; in versions 2.0 and later, volume labels and directory names can also appear in directory entries.

Directory searches

Directory entries are not sorted, nor are they maintained as a linked list. Thus, when MS-DOS searches a directory for a name, the search must proceed linearly from the first name in the directory. In MS-DOS versions 1.x, a directory search continues until the specified name is found or until every entry in the directory has been examined. In versions 2.0 and later, the search continues until the specified name is found or until a null directory entry (that is, one whose first byte is zero) is encountered. This null entry indicates the logical end of the directory.

Adding and deleting directory entries

MS-DOS deletes a directory entry by marking it with 0E5H in the first byte rather than by erasing it or excising it from the directory. New names are added to the directory by reusing the first deleted entry in the list. If no deleted entries are available, MS-DOS appends the new entry to the list.

The current directory

When more than one directory exists on a disk, MS-DOS keeps track of a default search directory known as the current directory. The current directory is the directory used for all implicit directory searches, such as those occasioned by a request to open a file, if no alternative path is specified. At startup, MS-DOS makes the root directory the current directory, but any other directory can be designated later, either interactively by using the CHDIR command or from within an application by using Interrupt 21H Function 3BH (Change Current Directory).

Directory Format

The root directory is created by the MS-DOS FORMAT program. See USER COMMANDS: FORMAT. The FORMAT program places the root directory immediately after the disk's file allocation tables (FATs). FORMAT also determines the size of the root directory. The size depends on the capacity of the storage medium: FORMAT places larger root directories on high-capacity fixed disks and smaller root directories on floppy disks. In contrast, the size of subdirectories is limited only by the storage capacity of the disk because disk space for subdirectories is allocated dynamically, as it is for any MS-DOS file. The size and physical location of the root directory can be derived from data in the BIOS parameter block (BPB) in the disk boot sector. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

Because space for the root directory is allocated only when the disk is formatted, the root directory cannot be deleted or moved. Subdirectories, whose disk space is allocated dynamically, can be added or deleted as needed.

Directory entry format

Each 32-byte directory entry consists of seven fields, including a name, an attribute byte, date and time stamps, and information that describes the file's size and physical location on the disk (Figure 8-2). The fields are formatted as described in the following paragraphs.

Byte	0	0BH	0CH	16H	18H	1AH	1CH	1FH
	Name	Attribute	(Reserved)	Time	Date	Starting cluster	File size	

Figure 8-2. Format of a directory entry.

The name field (bytes 0–0AH) contains an 11-byte name unless the first byte of the field indicates that the directory entry is deleted or null. The name can be an 11-byte filename (8-byte name followed by a 3-byte extension), an 11-byte subdirectory name (8-byte name

followed by a 3-byte extension), or an 11-byte volume label. Names less than 8 bytes and extensions less than 3 bytes are padded to the right with blanks so that the extension always appears in bytes 08-0AH of the name field. The first byte of the name field can contain certain reserved values that affect the way MS-DOS processes the directory entry:

Value	Meaning
0	Null directory entry (logical end of directory in MS-DOS versions 2.0 and later)
5	First character of name to be displayed as the character represented by 0E5H (MS-DOS version 3.2)
0E5H	Deleted directory entry

When MS-DOS creates a subdirectory, it always includes two aliases as the first two entries in the newly created directory. The name . (an ASCII period) is an alias for the name of the current directory; the name .. (two ASCII periods) is an alias for the directory's parent directory—that is, the directory in which the entry containing the name of the current directory is found.

The attribute field (byte 0BH) is an 8-bit field that describes the way MS-DOS processes the directory entry (Figure 8-3). Each bit in the attribute field designates a particular attribute of that directory entry; more than one of the bits can be set at a time.

Bit	7	6	5	4	3	2	1	0
	(Reserved)	(Reserved)	Archive	Sub-directory	Volume label	System file	Hidden file	Read-only file

Figure 8-3. Format of the attribute field in a directory entry.

The read-only bit (bit 0) is set to 1 to mark a file read-only. Interrupt 21H Function 3DH (Open File with Handle) will fail if it is used in an attempt to open this file for writing. The hidden bit (bit 1) is set to 1 to indicate that the entry is to be skipped in normal directory searches—that is, in directory searches that do not specifically request that hidden entries be included in the search. The system bit (bit 2) is set to 1 to indicate that the entry refers to a file used by the operating system. Like the hidden bit, the system bit excludes a directory entry from normal directory searches. The volume label bit (bit 3) is set to 1 to indicate that the directory entry represents a volume label. The subdirectory bit (bit 4) is set to 1 when the directory entry contains the name and location of another directory. This bit is always set for the directory entries that correspond to the current directory (.) and the parent directory (..). The archive bit (bit 5) is set to 1 by MS-DOS functions that close a file that has been written to. Simply opening and closing a file is not sufficient to update the archive bit in the file's directory entry.

The time and date fields (bytes 16–17H and 18–19H) are initialized by MS-DOS when the directory entry is created. These fields are updated whenever a file is written to. The formats of these fields are shown in Figures 8-4 and 8-5.

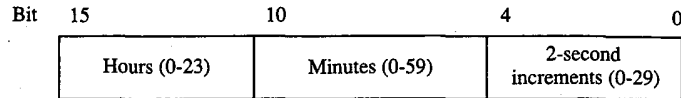


Figure 8-4. Format of the time field in a directory entry.

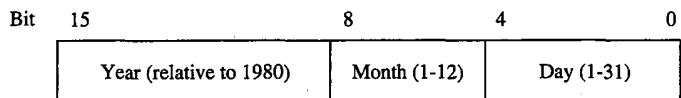


Figure 8-5. Format of the date field in a directory entry.

The starting cluster field (bytes 1A–1BH) indicates the disk location of the first cluster assigned to the file. This cluster number can be used as an entry point to the file allocation table (FAT) for the disk. (Cluster numbers can be converted to logical sector numbers with the aid of the information in the disk's BPB.)

For the . entry (the alias for the directory that contains the entry), the starting cluster field contains the starting cluster number of the directory itself. For the .. entry (the alias for the parent directory), the value in the starting cluster field refers to the parent directory unless the parent directory is the root directory, in which case the starting cluster number is zero.

The file size field (bytes 1C–1FH) is a 32-bit integer that indicates the file size in bytes.

Volume Labels

The generic term *volume* refers to a unit of auxiliary storage such as a floppy disk, a fixed disk, or a reel of magnetic tape. In computer environments where many different volumes might be used, the operating system can uniquely identify each volume by initializing it with a volume label.

Volume labels are implemented in MS-DOS versions 2.0 and later as a specific type of directory entry specified by setting bit 3 in the attribute field to 1. In a volume label directory entry, the name field contains an 11-byte string specifying a name for the disk volume. A volume label can appear only in the root directory of a disk, and only one volume label can be present on any given disk.

In MS-DOS versions 2.0 and later, the FORMAT command can be used with the /V switch to initialize a disk with a volume label. In versions 3.0 and later, the LABEL command can be used to create, update, or delete a volume label. Several commands can display a disk's volume label, including VOL, DIR, LABEL, TREE, and CHKDSK. See USER COMMANDS.

In MS-DOS versions 2.x, volume labels are simply a convenience for the user; no MS-DOS routine uses a volume label for any other purpose. In MS-DOS versions 3.x, however, the SHARE command examines a disk's volume label when it attempts to verify whether a disk volume has been inadvertently replaced in the midst of a file read or write operation. Removable disk volumes should therefore be assigned unique volume names if they are to contain shared files.

Functional Support for MS-DOS Directories

Several Interrupt 21H service routines can be useful to programmers who need to manipulate directories and their contents (Table 8-1). The routines can be broadly grouped into two categories: those that use a modified file control block (FCB) to pass filenames to and from the Interrupt 21H service routines (Functions 11H, 12H, 17H, and 23H) and those that use hierarchical path specifications (Functions 39H, 3AH, 3BH, 43H, 47H, 4EH, 4FH, 56H, and 57H). See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management; SYSTEM CALLS: INTERRUPT 21H.

The functions that use an FCB require that the calling program reserve enough memory for an extended FCB before the Interrupt 21H function is called. The calling program initializes the filename and extension fields of the FCB and passes the address of the FCB to the MS-DOS service routine in DS:DX. The functions that use pathnames expect all pathnames to be in ASCIIZ format—that is, the last character of the name must be followed by a zero byte.

Names in pathnames passed to Interrupt 21H functions can be separated by either a backslash (\) or a forward slash (/). (The forward slash is the separator character used in pathnames in UNIX/XENIX systems.) For example, the pathnames C:\MSP\SOURCE\ROSE.PAS and C:\MSP\SOURCE\ROSE.PAS are equivalent when passed to an Interrupt 21H function. The forward slash can thus be used in a pathname in a program that must run on both MS-DOS and UNIX/XENIX. However, the MS-DOS command processor (COMMAND.COM) recognizes only the backslash as a pathname separator character, so forward slashes cannot be used as separators in the command line.

Table 8-1. MS-DOS Functions for Accessing Directories.

Function	Call With	Returns	Comment
Find First File	AH = 11H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) DTA updated (if directory entry found)	If default not satisfactory, DTA must be set before using this function.
Find Next File	AH = 12H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) DTA updated (if directory entry found)	Use the same FCB for Function 11H and Function 12H.

(more)

Table 8-1. *Continued.*

Function	Call With	Returns	Comment
Rename File	AH = 17H DS:DX = pointer to modified FCB INT 21H	AL = 0 (file renamed) or 0FFH (no directory entry or duplicate filename)	
Get File Size	AH = 23H DS:DX = pointer to unopened FCB INT 21H	AL = 0 (directory entry found) or 0FFH (not found) FCB updated with number of records in file	
Create Directory	AH = 39H DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Remove Directory	AH = 3AH DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Change Current Directory	AH = 3BH DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Get/Set File Attributes	AH = 43H AL = 0 (get attributes) 1 (set attributes) CX = attributes (if AL = 1) DS:DX = pointer to ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error) CX = attribute field from directory entry (if called with AL = 0)	Cannot be used to modify the volume label or subdirectory bits.
Get Current Directory	AH = 47H DS:SI = pointer to 64-byte buffer DL = drive number INT 21H	Carry flag set (if error) AX = error code (if error) Buffer updated with pathname of current directory	
Find First File	AH = 4EH DS:DX = pointer to ASCIIZ pathname CX = file attributes to match INT 21H	Carry flag set (if error) AX = error code (if error) DTA updated	If default not satisfactory, DTA must be set before using this function.
Find Next File	AH = 4FH INT 21H	Carry flag set (if error) AX = error code (if error) DTA updated	

(more)

Table 8-1. *Continued.*

Function	Call With	Returns	Comment
Rename File	AH = 56H DS:DX = pointer to ASCIIZ pathname ES:DI = pointer to new ASCIIZ pathname INT 21H	Carry flag set (if error) AX = error code (if error)	
Get/Set Date/Time of File	AH = 57H AL = 0 (get date/time) 1 (set date/time) BX = handle CX = time (if AL = 1) DX = date (if AL = 1) INT 21H	Carry flag set (if error) AX = error code (if error) CX = time (if AL = 0) DX = date (if AL = 0)	

Searching a directory

Two pairs of Interrupt 21H functions are available for directory searches. Functions 11H and 12H use FCBs to transfer filenames to MS-DOS; these functions are available in all versions of MS-DOS, but they cannot be used with pathnames. Functions 4EH and 4FH support pathnames, but these functions are unavailable in MS-DOS versions 1.x. All four functions require the address of the disk transfer area (DTA) to be initialized appropriately before the function is invoked. When Function 12H or 4FH is used, the current DTA must be the same as the DTA for the preceding call to Function 11H or 4EH.

The Interrupt 21H directory search functions are designed to be used in pairs. The Find First File functions return the first matching directory entry in the current directory (Function 11H) or in the specified directory (Function 4EH). The Find Next File functions (Functions 12H and 4FH) can be called repeatedly after a successful call to the corresponding Find First File function. Each call to one of the Find Next File functions returns the next directory entry that matches the name originally specified to the Find First File function. A directory search can thus be summarized as follows:

```
call "find first file" function

while ( matching directory entry returned )
    call "find next file" function
```

Wildcard characters

This search strategy is used because name specifications can include the wildcard characters ?, which matches any single character, and * (*see* below). When one or more wildcard characters appear in the name specified to one of the Find First File functions, only the nonwildcard characters in the name participate in the directory search. Thus, for example, the specification FOO? matches the filenames FOO1, FOO2, and so on; the specification FOO?????.??? matches FOO4.COM, FOOBAR.EXE, and FOONEW.BAK, as well as FOO1 and FOO2; the specification ???????.TXT matches all files whose extension is .TXT; the specification ???????.??? matches all files in the directory.

Function 4EH also recognizes the wildcard character *, which matches any remaining characters in a filename or extension. MS-DOS expands the * wildcard character internally to question marks. Thus, for example, the specification FOO * is the same as FOO?????; the specification FOO *.* is the same as FOO?????.???; and, of course, the specification *.* is the same as ????????..???

Examining a directory entry

All four Interrupt 21H directory search functions return the name, attribute, file size, time, and date fields for each directory entry found during a directory search. The current DTA is used to return this data, although the format is different for the two pairs of functions: Functions 11H and 12H return a copy of the 32-byte directory entry—including the cluster number—in the DTA; Functions 4EH and 4FH return a 43-byte data structure that does not include the starting cluster number. See SYSTEM CALLS: INTERRUPT 21H: Function 4EH.

The attribute field of a directory entry can be examined using Function 43H (Get/Set File Attributes). Also, Function 57H (Get/Set Date/Time of File) can be used to examine a file's time or date. However, unlike the other functions discussed here, Function 57H is intended only for files that are being actively used within an application—that is, Function 57H can be called to examine the file's time or date stamp only after the file has been opened or created using an Interrupt 21H function that returns a handle (Function 3CH, 3DH, 5AH, or 5BH).

Modifying a directory entry

Four Interrupt 21H functions can modify the contents of a directory entry. Function 17H (Rename File) can be used to change the name field in any directory entry, including hidden or system files, subdirectories, and the volume label. Related Function 56H (Rename File) also changes the name field of a filename but cannot rename a volume label or a hidden or system file. However, it can be used to move a directory entry from one directory to another. (This capability is restricted to filenames only; subdirectory entries cannot be moved with Function 56H.)

Functions 43H (Get/Set File Attributes) and 57H (Get/Set Date/Time of File) can be used to modify specific fields in a directory entry. Function 43H can mark a directory entry as a hidden or system file, although it cannot modify the volume label or subdirectory bits. Function 57H, as noted above, can be used only with a previously opened file; it provides a way to read or update a file's time and date stamps without writing to the file itself.

Creating and deleting directories

Function 39H (Create Directory) exists only to create directories—that is, directory entries with the subdirectory bit set to 1. (Interrupt 21H functions that create files, such as Function 3CH, cannot assign the subdirectory attribute to a directory entry.) The converse function, 3AH (Remove Directory), deletes a subdirectory entry from a directory. (The subdirectory must be completely empty.) Again, Interrupt 21H functions that delete files from directories, such as Function 41H, cannot be used to delete subdirectories.

Specifying the current directory

A call to Interrupt 21H Function 47H (Get Current Directory) returns the pathname of the current directory in use by MS-DOS to a user-supplied buffer. The converse operation, in which a new current directory can be specified to MS-DOS, is performed by Function 3BH (Change Current Directory).

Programming examples: Searching for files

The subroutines in Figure 8-6 below illustrate Functions 4EH and 4FH, which use path specifications passed as ASCIIZ strings to search for files. Figure 8-7 applies these assembly-language subroutines in a simple C program that lists the attributes associated with each entry in the current directory. Note how the directory search is performed in the WHILE loop in Figure 8-7 by using a global wildcard file specification (*.*) and by repeatedly executing *FindNextFile()* until no further matching filenames are found. (See Programming Example: Updating a Volume Label for examples of the FCB-related search functions, 11H and 21H.)

```

                                TITLE   'DIRS.ASM'

;
; Subroutines for DIRDUMP.C
;

ARG1      EQU      [bp + 4]      ; stack frame addressing for C arguments
ARG2      EQU      [bp + 6]

_TEXT     SEGMENT byte public 'CODE'
          ASSUME  cs:_TEXT

;-----
;
; void SetDTA( DTA );
;         char *DTA;
;-----

_SetDTA   PUBLIC   _SetDTA
          PROC    near

          push   bp
          mov    bp,sp

          mov    dx,ARG1        ; DS:DX -> DTA
          mov    ah,1Ah         ; AH = INT 21H function number
          int    21h            ; pass DTA to MS-DOS

```

Figure 8-6. Subroutines illustrating Interrupt 21H Functions 4EH and 4FH.

(more)


```

        pop    bp
        ret

_SetDTA    ENDP

;-----
;
; int GetCurrentDir( *path );          /* returns error code */
;     char *path;                    /* pointer to buffer to contain path */
;-----

        PUBLIC _GetCurrentDir
_GetCurrentDir PROC    near

        push  bp
        mov   bp,sp
        push  si

        mov   si,ARG1                ; DS:SI -> buffer
        xor   dl,dl                    ; DL = 0 (default drive number)
        mov   ah,47h                  ; AH = INT 21H function number
        int   21h                      ; call MS-DOS; AX = error code
        jc    L01                      ; jump if error

        xor   ax,ax                    ; no error, return AX = 0

L01:     pop   si
        pop   bp
        ret

_GetCurrentDir ENDP

;-----
;
; int FindFirstFile( path, attribute ); /* returns error code */
;     char *path;
;     int attribute;
;-----

        PUBLIC _FindFirstFile
_FindFirstFile PROC    near

        push  bp
        mov   bp,sp

        mov   dx,ARG1                ; DS:DX -> path
        mov   cx,ARG2                ; CX = attribute
        mov   ah,4Eh                  ; AH = INT 21H function number
        int   21h                      ; call MS-DOS; AX = error code
        jc    L02                      ; jump if error

```

Figure 8-6. Continued.

(more)

```

                                xor     ax,ax           ; no error, return AX = 0
L02:                            pop     bp
                                ret
_FindFirstFile ENDP
;-----
;
; int FindNextFile();           /* returns error code */
;
;-----

_FINDNEXTFILE PUBLIC _FindNextFile
_FINDNEXTFILE PROC near

                                push   bp
                                mov    bp,sp

                                mov    ah,4Fh         ; AH = INT 21H function number
                                int    21h           ; call MS-DOS; AX = error code
                                jc     L03            ; jump if error

                                xor    ax,ax         ; if no error, set AX = 0

L03:                            pop     bp
                                ret
_FindNextFile ENDP

_TEXT ENDS

_DATA SEGMENT word public 'DATA'

CurrentDir DB 64 dup(?)
.DTA DB 64 dup(?)

_DATA ENDS

END
```

Figure 8-6. Continued.

```

/* DIRDUMP.C */

#define AllAttributes 0x3F          /* bits set for all attributes */

main()
{
    static char CurrentDir[64];
    int     ErrorCode;
    int     FileCount = 0;

    struct
    {
        char    reserved[21];
        char    attrib;
        int     time;
        int     date;
        long    size;
        char    name[13];
    }         DTA;

    /* display current directory name */

    ErrorCode = GetCurrentDir( CurrentDir );
    if( ErrorCode )
    {
        printf( "\nError %d: GetCurrentDir", ErrorCode );
        exit( 1 );
    }

    printf( "\nCurrent directory is \"%s\"", CurrentDir );

    /* display files and attributes */

    SetDTA( &DTA );                /* pass DTA to MS-DOS */

    ErrorCode = FindFirstFile( " *.*", AllAttributes );

    while( !ErrorCode )
    {
        printf( "\n%12s -- ", DTA.name );
        ShowAttributes( DTA.attrib );
        ++FileCount;

        ErrorCode = FindNextFile( );
    }

    /* display file count and exit */

    printf( "\nCurrent directory contains %d files\n", FileCount );
    return( 0 );
}

```

Figure 8-7. The complete DIRDUMP.C program.

(more)

```

ShowAttributes( a )
int    a;
{
    int    i;
    int    mask = 1;

    static char *AttribName[] =
    {
        "read-only ",
        "hidden ",
        "system ",
        "volume ",
        "subdirectory ",
        "archive "
    };

    for( i=0; i<6; i++ )          /* test each attribute bit */
    {
        if( a & mask )
            printf( AttribName[i] ); /* display a message if bit is set */
        mask = mask << 1;
    }
}

```

Figure 8-7. Continued.

Programming example: Updating a volume label

To create, modify, or delete a volume-label directory entry, the Interrupt 21H functions that work with FCBs should be used. Figure 8-8 contains four subroutines that show how to search for, rename, create, or delete a volume label in MS-DOS versions 2.0 and later.

```

        TITLE    'VOLS.ASM'

;-----
;
; C-callable routines for manipulating MS-DOS volume labels.
; Note: These routines modify the current DTA address.
;
;-----

ARG1            EQU    [bp + 4]          ; stack frame addressing

DGROUP         GROUP    _DATA

_TEXT          SEGMENT byte public 'CODE'
              ASSUME    cs:_TEXT,ds:DGROUP

```

Figure 8-8. Subroutines for manipulating volume labels.

(more)

```

;-----
;
; char *GetVolLabel();          /* returns pointer to volume label name */
;
;-----

_GetVolLabel PUBLIC _GetVolLabel
_GetVolLabel PROC near

    push    bp
    mov     bp,sp
    push    si
    push    di

    call    SetDTA          ; pass DTA address to MS-DOS
    mov     dx,offset DGROU:ExtendedFCB
    mov     ah,11h         ; AH = INT 21H function number
    int     21h           ; Search for First Entry
    test    al,al
    jnz     L01

    ; label found so make a copy
    mov     si,offset DGROU:DTA + 8
    mov     di,offset DGROU:VolLabel
    call    CopyName
    mov     ax,offset DGROU:VolLabel ; return the copy's address
    jmp     short L02

L01:      xor     ax,ax          ; no label, return 0 (null pointer)

L02:      pop     di
          pop     si
          pop     bp
          ret

_GetVolLabel ENDP

;-----
;
; int RenameVolLabel( label );  /* returns error code */
;     char *label;             /* pointer to new volume label name */
;
;-----

_RenameVolLabel PUBLIC _RenameVolLabel
_RenameVolLabel PROC near

    push    bp
    mov     bp,sp
    push    si
    push    di

```

Figure 8-8. Continued.

(more)

```

mov     si,offset DGROUP:VolLabel ; DS:SI -> old volume name
mov     di,offset DGROUP:Name1
call    CopyName ; copy old name to FCB

mov     si,ARG1
mov     di,offset DGROUP:Name2
call    CopyName ; copy new name into FCB

mov     dx,offset DGROUP:ExtendedFCB ; DS:DX -> FCB
mov     ah,17h ; AH = INT 21H function number
int     21h ; rename
xor     ah,ah ; AX = 00H (success) or 0FFH (failure)

pop     di ; restore registers and return
pop     si
pop     bp
ret

```

```
_RenameVolLabel ENDP
```

```

;-----
;
; int NewVolLabel( label ); ; returns error code */
; char *label; ; pointer to new volume label name */
;-----

```

```

PUBLIC _NewVolLabel
_NewVolLabel PROC near

push    bp
mov     bp,sp
push    si
push    di

mov     si,ARG1
mov     di,offset DGROUP:Name1
call    CopyName ; copy new name to FCB

mov     dx,offset DGROUP:ExtendedFCB
mov     ah,16h ; AH = INT 21H function number
int     21h ; create directory entry
xor     ah,ah ; AX = 00H (success) or 0FFH (failure)

pop     di ; restore registers and return
pop     si
pop     bp
ret

_NewVolLabel ENDP

```

Figure 8-8. Continued.

(more)

```

-----
;
; int DeleteVolLabel();          /* returns error code */
;
-----

        PUBLIC  _DeleteVolLabel
_DeleteVolLabel PROC  near

        push   bp
        mov    bp,sp
        push   si
        push   di

        mov    si,offset DGROUP:VolLabel
        mov    di,offset DGROUP:Name1
        call   CopyName          ; copy current volume name to FCB

        mov    dx,offset DGROUP:ExtendedFCB
        mov    ah,13h           ; AH = INT 21H function number
        int    21h             ; delete directory entry
        xor    ah,ah           ; AX = 00H (success) or 0FFH (failure)

        pop    di              ; restore registers and return
        pop    si
        pop    bp
        ret

_DeleteVolLabel ENDP

-----
;
; miscellaneous subroutines
;
-----

SetDTA      PROC      near

        push   ax              ; preserve registers used
        push   dx

        mov    dx,offset DGROUP:DTA ; DS:DX -> DTA
        mov    ah,1Ah          ; AH = INT 21H function number
        int    21h            ; set DTA

        pop    dx              ; restore registers and return
        pop    ax
        ret

SetDTA      ENDP

```

Figure 8-8. Continued.

(more)

```

CopyName      PROC      near          ; Caller: SI -> ASCIIZ source
                                           ;           DI -> destination
                push     ds
                pop      es           ; ES = DGROUP
                mov      cx,11       ; length of name field

L11:          lodsb
                test     al,al
                jz       L12         ; .. until null character is reached
                stosb
                loop    L11

L12:          mov      al,' '       ; pad new name with blanks
                rep     stosb
                ret

CopyName      ENDP

_TEXT        ENDS

_DATA       SEGMENT word public 'DATA'

VolLabel     DB        11 dup(0),0

ExtendedFCB  DB        0FFh        ; must be 0FFH for extended FCB
                DB        5 dup(0)  ; (reserved)
                DB        1000b     ; attribute byte (bit 3 = 1)
                DB        0        ; default drive ID
Name1        DB        11 dup('?') ; global wildcard name
                DB        5 dup(0)  ; (unused)
Name2        DB        11 dup(0)   ; second name (for renaming entry)
                DB        9 dup(0)  ; (unused)

DTA          DB        64 dup(0)

_DATA       ENDS

                END

```

Figure 8-8. Continued.

Richard Wilton

Article 9

Memory Management

Personal computers that are MS-DOS compatible can be outfitted with as many as three kinds of random-access memory (RAM): conventional memory, expanded memory, and extended memory.

All MS-DOS machines have at least some conventional memory, but the presence of expanded or extended memory depends on the installed hardware options and the model of microprocessor on which the computer is based. Each storage class has its own capabilities, characteristics, and limitations. Each also has its own management techniques, which are the subject of this chapter.

Conventional Memory

Conventional memory is the term for the up to 1 MB of memory that is directly addressable by an Intel 8086/8088 microprocessor or by an 80286 or 80386 microprocessor running in real mode (8086-emulation mode). Physical addresses for references to conventional memory are generated by a 16-bit segment register, which acts as a base register and holds a paragraph address, combined with a 16-bit offset contained in an index register or in the instruction being executed.

On IBM PCs and compatibles, MS-DOS and the programs that run under its control occupy the bottom 640 KB or less of the conventional memory space. The memory space above the 640 KB mark is partitioned among ROM (read-only memory) chips on the system board that contain various primitive device handlers and test programs and among RAM and ROM chips on expansion boards that are used for input and output buffers and for additional device-dependent routines.

The bottom 640 KB of memory administered by MS-DOS is divided into three zones (Figure 9-1):

- The interrupt vector table
- The operating system area
- The transient program area

The interrupt vector table occupies the lowest 1024 bytes of memory (locations 00000–003FFH); its address and length are hard-wired into the processor and cannot be changed. Each doubleword position in the table is called an interrupt vector and contains the segment and offset of an interrupt handler routine for the associated hardware or software interrupt number. Interrupt handler routines are usually built into the operating system,

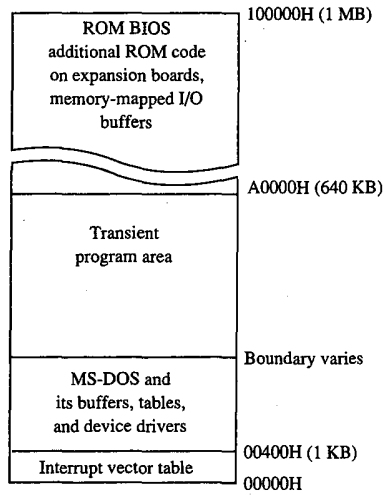


Figure 9-1. A diagram showing conventional memory in an IBM PC-compatible MS-DOS system. The bottom 1024 bytes of memory are used for the interrupt vector table. The memory above the vector table, up to the 640 KB boundary, is available for use by MS-DOS and the programs that run under its control. The top 384 KB are used for the ROM BIOS, other device-control and diagnostic routines, and memory-mapped input and output.

but in special cases application programs can contain handler routines of their own. Vectors for interrupt numbers that are not used for software linkages or by some hardware device are usually initialized by the operating system to point to a simple interrupt return (IRET) instruction or to a routine that displays an error message.

The operating-system area begins immediately above the interrupt vector table and holds the operating system proper, its tables and buffers, any additional installable device drivers specified in the CONFIG.SYS file, and the resident portion of the COMMAND.COM command interpreter. The amount of memory occupied by the operating-system area varies with the version of MS-DOS being used, the number of disk buffers, and the number and size of installed device drivers.

The transient program area (TPA) is the remainder of RAM above the operating-system area, extending to the 640 KB limit or to the end of installed RAM (whichever is smaller). External MS-DOS commands (such as CHKDSK) and other programs are loaded into the TPA for execution. The transient portion of COMMAND.COM also runs in this area.

The TPA is organized into a structure called the memory arena, which is divided into portions called *arena entries* (or memory blocks). These entries are allocated in paragraph (16-byte) multiples and can be as small as one paragraph or as large as the entire TPA. Each arena entry is preceded by a control structure called an arena entry header, which contains information indicating the size and status of the arena entry.

MS-DOS inspects the arena entry headers whenever a function requesting a memory-block allocation, modification, or release is issued; when a program is loaded and executed with the EXEC function (Interrupt 21H Function 4BH); or when a program is terminated. If any of the arena entry headers appear to be damaged, MS-DOS returns an error to the calling process. If that process is COMMAND.COM, COMMAND.COM then displays the message *Memory allocation error* and halts the system.

MS-DOS support for conventional memory management

The MS-DOS kernel supports three memory-management functions, invoked with Interrupt 21H, that operate on the TPA:

- Function 48H (Allocate Memory Block)
- Function 49H (Free Memory Block)
- Function 4AH (Resize Memory Block)

These three functions (Table 9-1) can be called by application programs, by the command processor, and by MS-DOS itself to dynamically allocate, resize, and release arena entries as they are needed. See SYSTEM CALLS: INTERRUPT 21H: Functions 48H; 49H; 4AH.

Table 9-1. MS-DOS Memory-Management Functions.

Function Name	Call With	Returns
Allocate Memory Block	AH = 48H BX = paragraphs needed	AX = segment of allocated block If failed: BX = size of largest available block in paragraphs
Free Memory Block	AH = 49H ES = segment of block to release	nothing
Resize (Allocated) Memory Block	AH = 4AH BX = new size of block in paragraphs ES = segment of block to resize	If failed: BX = maximum size for block in paragraphs
Get/Set Allocation Strategy*	AH = 58H AL = 00H (get strategy) 01H (set strategy) If setting: BX = strategy: 00H = first fit 01H = best fit 02H = last fit	If getting: AX = strategy code

*MS-DOS versions 3.x only.

When the MS-DOS kernel receives a memory-allocation request, it inspects the chain of arena entry headers to find a free arena entry that can satisfy the request. The memory manager can use any of three allocation strategies:

- First fit—the arena entry at the lowest address that is large enough to satisfy the request
- Best fit—the smallest available arena entry that satisfies the request, regardless of its position
- Last fit—the arena entry at the highest address that is large enough to satisfy the request

If the arena entry selected is larger than the size needed to fulfill the request, the arena entry is divided and the program is given an arena entry exactly the size it requires. A new arena entry header is then created for the remaining portion of the original arena entry; it is marked “unowned” and can be used to satisfy subsequent allocation calls.

Research on allocation strategies has demonstrated that the first-fit approach is most efficient, and this is the default strategy used by MS-DOS. However, in MS-DOS versions 3.0 and later, an application program can select a different strategy for the memory manager with Interrupt 21H Function 58H (Get/Set Allocation Strategy). See SYSTEM CALLS: INTERRUPT 21H: Function 58H.

Using the memory-management functions

When a program begins executing, it already owns two arena entries allocated on its behalf by the MS-DOS EXEC function (Interrupt 21H Function 4BH). The first entry holds the program’s environment and is just large enough to contain this information; the second entry (called the program block in this article) contains the program’s PSP, code, data, and stack.

The amount of memory MS-DOS allocates to the program block for a newly loaded transient program depends on its type (.COM or .EXE). Under typical conditions, a .COM program is allocated all of the first arena entry that is large enough to hold the contents of its file, plus 256 bytes for the PSP and at least 2 bytes for the stack. Because the TPA is seldom fragmented into more than one arena entry before a program is loaded, a .COM program usually ends up owning all the memory in the system that does not belong to the operating system itself—memory divided between a relatively small environment and a comparatively immense program block.

The amount of memory allocated to a .EXE program, on the other hand, is controlled by two fields called MINALLOC and MAXALLOC in the .EXE program file header. The MINALLOC field tells the MS-DOS loader how many paragraphs of memory, in addition to the memory required to hold the initialized code and the data present in the file, *must* be available for the program to execute at all. The MAXALLOC field contains the maximum number of excess paragraphs, *if available*, to allocate to the program.

The default value placed in MAXALLOC by the Microsoft Object Linker is FFFFH paragraphs, corresponding to 1 MB. Consequently, a .EXE program is typically allocated all of available memory when it is loaded, as is a .COM file. Although it is possible to set the MAXALLOC field to other, smaller values with the linker's /CPARMAXALLOC switch or with the EXEMOD utility supplied with Microsoft language compilers, few programmers bother to do so.

In short, when a program begins executing, it usually owns all of available memory — frequently much more memory than it needs. If the program wants to be well behaved in its use of memory and, possibly, load child programs as well, it should immediately release any extra memory. In assembly-language programs, the extra memory is released by calling Interrupt 21H Function 4AH (Resize Memory Block) with the segment of the program's PSP in the ES register and the number of paragraphs of memory to retain for the program's use in the BX register. (See Figures 9-2 and 9-3.) In most high-level languages, such as Microsoft C, excess memory is released by the run-time library's startup module.

```

    .
    .
    .
_TEXT  segment para public 'CODE'

    org    100h

    assume cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT

main   proc    near           ; entry point from MS-DOS
                                ; CS = DS = ES = SS = PSP

                                ; first move our stack
                                ; to a safe place...
    mov    sp,offset stk

                                ; now release extra memory...
    mov    bx,offset stk       ; calculate paragraphs to keep
    mov    cl,4                ; (divide offset of end of
    shr    bx,cl               ; program by 16 and round up)
    inc    bx
    mov    ah,4ah              ; Fxn 4AH = resize mem block
    int    21h                 ; transfer to MS-DOS
    jc    error                ; jump if resize failed
    .
    .                           ; otherwise go on with work...
    .
main   endp

```

(more)

Figure 9-2. An example of a .COM program releasing excess memory after it receives control from MS-DOS. Interrupt 21H Function 4AH is called with the segment address of the program's PSP in register ES and the number of paragraphs of memory to retain in register BX.

```

        dw      64 dup (?)
stk     equ     $           ; base of new stack area

_TEXT  ends

        end     main       ; defines program entry point

```

Figure 9-2. Continued.

```

_TEXT  segment word public 'CODE'      ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:STACK

main   proc    far                   ; entry point from MS-DOS
        ; CS = _TEXT segment,
        ; DS = ES = PSP

        mov     ax,_DATA              ; set DS = our data segment
        mov     ds,ax

        ; give back extra memory...
        mov     ax,es                 ; let AX = segment of PSP base
        mov     bx,ss                 ; and BX = segment of stack base
        sub     bx,ax                 ; reserve seg stack - seg psp
        add     bx,stksize/16        ; plus paragraphs of stack
        inc     bx                     ; round up
        mov     ah,4ah                ; Fxn 4AH = resize memory block
        int     21h                  ; transfer to MS-DOS
        jc     error                 ; jump if resize failed

        .
        .
        .

main   endp

_TEXT  ends

_DATA  segment word public 'DATA'     ; static & variable data

        .
        .
        .

_DATA  ends

```

(more)

Figure 9-3. An example of a .EXE program releasing excess memory after it receives control from MS-DOS. This particular code sequence depends on the segment order shown. When a .EXE program is linked from many different object modules, other techniques may be needed to determine the amount of memory occupied by the program at run time.

```

STACK  segment para stack 'STACK'

        db      stksize dup (?)

STACK  ends

        end      main          ; defines program entry point

```

Figure 9-3. Continued.

Later, if the transient program needs additional memory for a buffer, table, or other work area, it can call Interrupt 21H Function 48H (Allocate Memory Block) with the desired number of paragraphs. If a sufficiently large block of memory is available, MS-DOS creates a new arena entry of the requested size and returns a pointer to its base in the form of a segment address in the AX register. If an arena entry of the requested size cannot be created, MS-DOS returns an error code in the AX register and the size in paragraphs of the largest available block of memory in the BX register. The application program can inspect this value to determine whether it can continue in a degraded fashion with a smaller amount of memory.

When a program finishes using an allocated arena entry, it should promptly call Interrupt 21H Function 49H to release it. This allows MS-DOS to collect small blocks of freed memory into contiguous arena entries and reduces the chance that future allocation requests by the same program will fail because of memory fragmentation. In any case, all arena entries owned by a program are released when the program terminates with Interrupt 20H or with Interrupt 21H Function 00H or 4CH.

A program skeleton demonstrating the use of dynamic memory allocation services is shown in Figure 9-4.

```

.
.
.
mov     bx,800h          ; 800H paragraphs = 32 KB
mov     ah,48h          ; Fxn 48H = allocate block
int     21h             ; transfer to MS-DOS
jc      error           ; jump if allocation failed
mov     bufseg,ax       ; save segment of block

.
.
.
mov     dx,offset file1 ; DS:DX = filename address
mov     ax,3d00h        ; Fxn 3DH = open, read only
int     21h             ; transfer to MS-DOS
jc      error           ; jump if open failed
mov     handle1,ax     ; save handle for work file

```

(more)

Figure 9-4. A skeleton example of dynamic memory allocation. The program requests a 32 KB memory block, uses it to copy its working file to a backup file, and then releases the memory block. Note the use of ASSUME directives to force the assembler to generate proper segment overrides on references to variables containing file handles.

```

                                ; create backup file...
mov     dx,offset file2 ; DS:DX = filename address
mov     cx,0             ; CX = attribute (normal)
mov     ah,3ch           ; Fxn 3CH = create file
int     21h             ; transfer to MS-DOS
jc      error           ; jump if create failed
mov     handle2,ax      ; save handle for backup file

push    ds              ; set ES = our data segment
pop     es
mov     ds,bufseg      ; set DS:DX = allocated block
xor     dx,dx

assume  ds:NOTHING,es:_DATA ; tell assembler

next:
mov     bx,handle1     ; handle for work file
mov     cx,8000h       ; try to read 32 KB
mov     ah,3fh         ; Fxn 3FH = read
int     21h           ; transfer to MS-DOS
jc      error         ; jump if read failed
or     ax,ax          ; was end of file reached?
jz     done           ; yes, exit this loop

                                ; now write backup file...
mov     cx,ax          ; set write length = read length
mov     bx,handle2     ; handle for backup file
mov     ah,40h         ; Fxn 40H = write
int     21h           ; transfer to MS-DOS
jc      error         ; jump if write failed
cmp     ax,cx          ; was write complete?
jne     error         ; no, disk must be full
jmp     next          ; transfer another record

done:  push    es      ; restore DS = data segment
        pop     ds

        assume  ds:_DATA,es:NOTHING ; tell assembler

                                ; release allocated block...
mov     es,bufseg      ; segment base of block
mov     ah,49h         ; Fxn 49H = release block
int     21h           ; transfer to MS-DOS
jc      error         ; (should never fail)

                                ; now close backup file...
mov     bx,handle2     ; handle for backup file
mov     ah,3eh         ; Fxn 3EH = close
int     21h           ; transfer to MS-DOS
jc      error         ; jump if close failed

```

Figure 9-4. Continued.

(more)


```

file1  db      'MYFILE.DAT',0 ; name of working file
file2  db      'MYFILE.BAK',0 ; name of backup file

handle1 dw    ?                ; handle for working file
handle2 dw    ?                ; handle for backup file
bufseg dw    ?                ; segment of allocated block

```

Figure 9-4. Continued.

Expanded Memory

The original Expanded Memory Specification (EMS) version 3.0 was developed as a joint effort of Lotus Development Corporation and Intel Corporation and was announced at the Spring COMDEX in 1985. The EMS was designed to provide a uniform means for applications running on 8086/8088-based personal computers, or on 80286/80386-based computers in real mode, to circumvent the 1 MB limit on conventional memory, thus providing such programs with much larger amounts of fast random-access storage. The EMS version 3.2, modified from 3.0 to add support for multitasking operating systems, was released shortly afterward as a joint effort of Lotus, Intel, and Microsoft.

The EMS is a functional definition of a bank-switched memory subsystem; it consists of user-installable boards that plug into the IBM PC's expansion bus and a resident driver program called the Expanded Memory Manager (EMM) that is provided by the board manufacturer. As much as 8 MB of expanded memory can be installed in a single machine. Expanded memory is made available to application software in 16 KB pages, which are mapped by the EMM into a contiguous 64 KB area called the page frame somewhere above the conventional memory area used by MS-DOS (0-640 KB). An application program can thus access as many as four 16 KB expanded memory pages simultaneously. The location of the page frame is user configurable so that it will not conflict with other hardware options (Figure 9-5).

The Expanded Memory Manager

The Expanded Memory Manager provides a hardware-independent interface between application programs and the expanded memory board(s). The EMM is supplied by the board manufacturer in the form of an installable character-device driver and is linked into MS-DOS by a DEVICE directive added to the CONFIG.SYS file on the system startup disk.

Internally, the EMM is divided into two distinct components that can be referred to as the driver and the manager. The driver portion mimics some of the actions of a genuine installable device driver, in that it includes Initialization and Output Status subfunctions and a valid device header. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

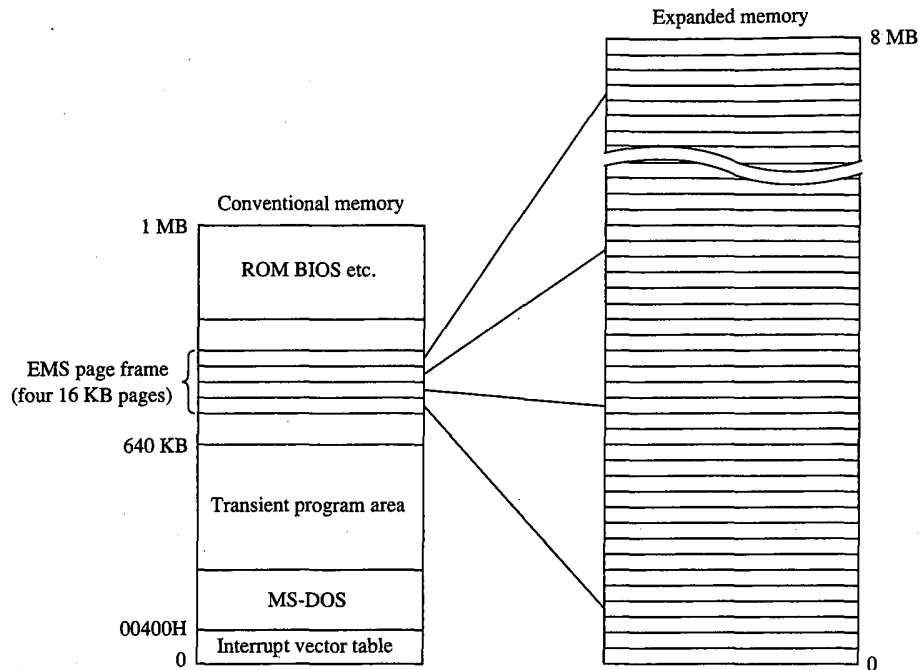


Figure 9-5. A sketch of the relationship of expanded memory to conventional memory; 16 KB pages of expanded memory are mapped into a 64 KB area, called the page frame, above the 640 KB boundary. The location of the page frame can be configured by the user to eliminate conflicts with ROMs or I/O buffers on expansion boards.

The second, and major, element of the EMM is the true interface between application software and the expanded memory hardware. Several classes of services provide

- Status of the expanded memory subsystem
- Allocation of expanded memory pages
- Mapping of logical pages into physical memory
- Deallocation of expanded memory pages
- Support for multitasking operating systems
- Diagnostic routines

Application programs communicate with the EMM directly by means of a software interrupt (Interrupt 67H). The MS-DOS kernel does not take part in expanded memory manipulations and does not use expanded memory for its own purposes.

Checking for expanded memory

Before it attempts to use expanded memory for storage, an application program must establish that the EMM is present and functional, and then it must use the manager portion of the EMM to check the status of the memory boards themselves. There are two methods a program can use to test for the existence of the EMM.

The first method is to issue an Open File or Device request (Interrupt 21H Function 3DH) using the guaranteed device name of the EMM driver: EMMXXXXX0. If the open operation succeeds, one of two conditions is indicated — either the driver is present or a file with the same name exists in the current directory of the default disk drive. To rule out the latter possibility, the application can issue IOCTL Get Device Information (Interrupt 21H Function 44H Subfunction 00H) and Check Output Status (Interrupt 21H Function 44H Subfunction 07H) requests to determine whether the handle returned by the open operation is associated with a file or with a device. In either case, the handle that was obtained from the open function should then be closed (Interrupt 21H Function 3EH) so that it can be reused for another file or device.

The second method of testing for the driver is to use the address that is found in the vector for Interrupt 67H to inspect the device header of the presumed EMM. (The contents of the vector can be obtained conveniently with Interrupt 21H Function 35H.) If the EMM is present, the name field at offset 0AH of the device header contains the string EMMXXXXX0. This method is nearly foolproof, and it avoids the relatively high overhead of an MS-DOS open function. However, it is somewhat less well behaved because it involves inspection of memory that does not belong to the application.

The two methods of testing for the existence of the EMM are illustrated in Figures 9-6 and 9-7.

```

; attempt to "open" EMM...
mov     dx,seg emm_name ; DS:DX = address of name
mov     ds,dx          ; of EMM
mov     dx,offset emm_name
mov     ax,3d00h       ; Fxn 3DH, Mode = 00H
; = open, read-only
int     21h           ; transfer to MS-DOS
jc      error         ; jump if open failed

; open succeeded, make sure
; it was not a file...

```

(more)

Figure 9-6. Testing for the presence of the Expanded Memory Manager with the MS-DOS Open File or Device (Interrupt 21H Function 3DH) and IOCTL (Interrupt 21H Function 44H) functions.

```

mov     bx,ax           ; BX = handle from open
mov     ax,4400h        ; Fxn 44H Subfxn 00H
                        ; = IOCTL Get Device Information
int     21h            ; transfer to MS-DOS
jc      error          ; jump if IOCTL call failed
and     dx,80h         ; Bit 7 = 1 if character device
jz      error          ; jump if it was a file

                        ; EMM is present, make sure
                        ; it is available...
                        ; (BX still contains handle)
mov     ax,4407h        ; Fxn 44H Subfxn 07H
                        ; = IOCTL Get Output Status
int     21h            ; transfer to MS-DOS
jc      error          ; jump if IOCTL call failed
or      al,al          ; test device status
jz      error          ; if AL = 0 EMM is not available

                        ; now close handle ...
                        ; (BX still contains handle)
mov     ah,3eh         ; Fxn 3EH = Close
int     21h            ; transfer to MS-DOS
jc      error          ; jump if close failed
.
.
.
emm_name db 'EMMXXXX0',0 ; guaranteed device name for EMM

```

Figure 9-6. Continued.

```

emm_int equ 67h        ; EMM software interrupt
.
.
.
                        ; first fetch contents of
                        ; EMM interrupt vector...
mov     al,emm_int     ; AL = EMM int number
mov     ah,35h         ; Fxn 35H = get vector
int     21h            ; transfer to MS-DOS
                        ; now ES:BX = handler address

                        ; assume ES:0000 points
                        ; to base of the EMM...

```

(more)

Figure 9-7. Testing for the presence of the Expanded Memory Manager by inspecting the name field in the device driver header.

```

mov     di,10           ; ES:DI = address of name
                        ; field in device header
mov     si,seg emm_name ; DS:SI = address of
mov     ds,si          ; expected EMM driver name
mov     si,offset emm_name
mov     cx,8           ; length of name field
cld
repz   cmpsb          ; compare names...
jnz    error          ; jump if driver absent
.
.
.

emm_name db 'EMMXXX0' ; guaranteed device name for EMM

```

Figure 9-7. Continued.

Using expanded memory

After establishing that the EMM is present, the application program can bypass MS-DOS and communicate with the EMM directly by means of software Interrupt 67H. The calling sequence is as follows:

```

mov     ah,function    ; AH selects EMM function
.
.
.                    ; Load other registers with
                    ; values specific to the
                    ; requested service

int     67h           ; Transfer to EMM

```

In general, the ES:DI registers are used to pass the address of a buffer or an array, and the DX register is used to hold an expanded memory "handle." Some EMM functions also use other registers (chiefly AL and BX) to pass such information as logical and physical page numbers. Table 9-2 summarizes the services available from the EMM.

Upon return from an EMM function call, the AH register contains zero if the function was successful; otherwise, AH contains an error code with the most significant bit set (Table 9-3). Other values are typically returned in the AL and BX registers or in a user-specified buffer.

Table 9-2. Summary of the Software Interface to Application Programs Provided by the EMM.*

Function Name	Action	Call With	Returns	Comments
Get Manager Status	Test whether the expanded memory software and hardware are functional.	AH = 40H	AH = status	This call is used after the program has established, with one of the techniques presented in Figures 9-6 and 9-7, that the EMM is present.
Get Page Frame Segment	Obtain the segment address of the EMM page frame.	AH = 41H	AH = status BX = segment of page frame, if AH = 00H	The page frame is divided into four 16 KB pages that are used to map logical expanded memory pages into the physical memory space of the 8086/8088 processor.
Get Expanded Memory Pages	Obtain the number of logical expanded memory pages present in the system and the number of pages that are not already allocated.	AH = 42H	AH = status BX = unallocated EMM pages, if AH = 00H DX = total EMM pages in system	The application need not have already acquired an EMM handle to use this function.
Allocate Expanded Memory	Obtain an EMM handle and allocate logical pages to be controlled by that handle.	AH = 43H BX = logical pages to allocate	AH = status DX = handle, if AH = 00H	This function is equivalent to a file-open function for the EMM. The handle returned is analogous to a file handle and owns a certain number of EMM pages. The handle must be used with every subsequent request to map memory and must be released by a close operation when the application is finished.
Map Memory	Map one of the logical pages of expanded memory assigned to a handle onto one of the four physical pages within the EMM's page frame.	AH = 44H AL = physical page (0-3) BX = logical page (0...n-1) DX = EMM handle	AH = status	This function can fail because either the available EMM handles or the EMM pages have been exhausted. Function 42H can be called by the application to determine the actual number of pages available. The logical page number must be in the range 0-n-1, where n is the number of logical pages previously allocated to the EMM handle with Function 43H. To access the memory after it has been mapped to a physical page, the application also needs the segment of the EMM's page frame, which can be obtained with Function 41H.

Release Handle and Memory	Deallocate the logical pages of expanded memory currently assigned to a handle and then release the handle itself for reuse.	AH = 45H DX = EMM handle	AH = status	This function is the equivalent of a close operation on a file. It notifies the EMM that the application will not be making further use of the data it may have stored within expanded memory pages.
Get EMM Version	Return the version number of the EMM software.	AH = 46H	AH = status AL = EMM version, if AH = 00H	The returned value is the version of the EMM with which the driver complies. The version number is encoded as BCD, with the integer part in the upper 4 bits and the fractional part in the lower 4 bits.
Save Mapping Context	Save the contents of the expanded memory page-mapping registers on the expanded memory boards, associating those contents with a specific EMM handle.	AH = 47H DX = EMM handle	AH = status	This function is designed for use by interrupt handlers and resident drivers or utilities that must access expanded memory. The handle supplied to the function is the handle that was assigned to the interrupt handler during its initialization sequence, not to the program that was interrupted.
Restore Mapping Context	Restore the contents of all expanded memory hardware page-mapping registers to the values associated with the given handle.	AH = 48H DX = EMM handle	AH = status	Use of this function must be balanced by a previous call to EMM Function 47H. It allows an interrupt handler or a resident driver that used expanded memory to restore the mapping context to its state at the point of interruption.
Get Number of EMM Handles	Return the number of active EMM handles.	AH = 4BH	AH = status BX = number of EMM handles, if AH = 00H	If the number of handles returned is zero, none of the expanded memory is in use. The number of active EMM handles never exceeds 255. A single program can make several allocation requests and therefore own several EMM handles.
Get Pages Owned by Handle	Return the number of logical expanded memory pages allocated to a specific handle.	AH = 4CH DX = EMM handle	AH = status BX = logical pages, if AH = 00H	The number of pages returned if the function is successful is always in the range 1–512. An EMM handle never has zero pages of memory allocated to it.

*EMM Functions 49H and 4AH (not listed) were defined in EMS version 3.0 and are "reserved" in later EMS versions.

(more)

Table 9-2. Continued.

Function Name	Action	Call With	Returns	Comments
Get Pages for All Handles	Return an array that contains all the active handles and the number of logical expanded memory pages associated with each handle.	AH = 4DH DI = offset of array to receive information ES = array segment	AH = status BX = number of active EMM handles If AH = 00H, array is filled in as described in comments column	The array is filled in with doubleword entries. The first word of each entry contains a handle; the second word contains the number of pages associated with that handle. The value returned in BX gives the number of valid doubleword entries in the array. Because 255 is the maximum number of EMM handles, the array need not be larger than 1020 bytes.
Get/Set Page Map	Save or set the contents of the EMM page-mapping registers on the expanded memory boards.	AH = 4EH AL = subfunction number DS:SI = array holding mapping information (Subfunctions 01H, 02H) ES:DI = array to receive information (Subfunctions 00H, 02H)	AH = status AL = bytes in page-mapping array (Subfunction 03H) Array pointed to by ES:DI receives mapping information for Subfunctions 00H and 02H	Subfunctions: 00H = get mapping registers into array 01H = set mapping registers from array 02H = get and set mapping registers in one operation 03H = return needed size of page-mapping array This function was added in EMM version 3.2 and is designed to support multitasking. It should not ordinarily be used by application programs. The content of the array is hardware and EMM software dependent. In addition to the contents of the page-mapping registers, it may contain other information that is necessary to restore the expanded memory subsystem to its previous state.

Table 9-3. The Expanded Memory Manager (EMM) Error Codes.

Error Code	Significance
00H	Function was successful.
80H	Internal error in the EMM software. Possible causes include an error in the driver itself or damage to its memory image.
81H	Malfunction in the expanded memory hardware.
82H	EMM is busy.
83H	Invalid expanded memory handle.
84H	Function requested by the application is not supported by the EMM.
85H	No more expanded memory handles available.
86H	Error in save or restore of mapping context.
87H	Allocation request specified more logical pages than are available in the system; no pages were allocated.
88H	Allocation request specified more logical pages than are currently available in the system (the request does not exceed the physical pages that exist, but some are already allocated to other handles); no pages were allocated.
89H	Zero pages cannot be allocated.
8AH	Logical page requested for mapping is outside the range of pages assigned to the handle.
8BH	Illegal physical page number in mapping request (not in the range 0-3).
8CH	Save area for mapping contexts is full.
8DH	Save of mapping context failed because save area already contains a context associated with the requested handle.
8EH	Restore of mapping context failed because save area does not contain a context for the requested handle.
8FH	Subfunction parameter not defined.

An application program that uses expanded memory should regard that memory as a system resource, such as a file or a device, and use only the documented EMM services to allocate, access, and release expanded memory pages. Here is the general strategy that can be used by such a program:

1. Establish the presence of the EMM by one of the two methods demonstrated in Figures 9-6 and 9-7.
2. After the driver is known to be present, check its operational status with EMM Function 40H.
3. Check the version number of the EMM with EMM Function 46H to ensure that all services the application will request are available.
4. Obtain the segment of the page frame used by the EMM with EMM Function 41H.
5. Allocate the desired number of expanded memory pages with EMM Function 43H. If the allocation is successful, the EMM returns a handle in DX that is used by the application to refer to the expanded memory pages it owns. This step is exactly analogous

- to opening a file and using the handle obtained from the open function for subsequent read/write operations on the file.
6. If the requested number of pages is not available, query the EMM for the actual number of pages available (EMM Function 42H) and determine whether the program can continue.
 7. After successfully allocating the number of expanded memory pages needed, use EMM Function 44H to map logical pages in and out of the physical page frame, to store and retrieve data in expanded memory.
 8. When finished using the expanded memory pages, release them by calling EMM Function 45H. Otherwise, the pages will not be available for use by other programs until the system is restarted.

A program skeleton that illustrates this general approach to the use of expanded memory is shown in Figure 9-8.

```

mov     ah,40h           ; test EMM status
int     67h
or      ah,ah
jnz     error           ; jump if bad status from EMM

mov     ah,46h           ; check EMM version
int     67h
or      ah,ah
jnz     error           ; jump if couldn't get version
cmp     al,30h          ; make sure at least ver. 3.0
jb      error           ; jump if wrong EMM version

mov     ah,41h           ; get page frame segment
int     67h
or      ah,ah
jnz     error           ; jump if failed to get frame
mov     page_frame,bx   ; save segment of page frame

mov     ah,42h           ; get no. of available pages
int     67h
or      ah,ah
jnz     error           ; jump if get pages error
mov     total_pages,dx  ; save total EMM pages
mov     avail_pages,bx  ; save available EMM pages
or      bx,bx
jz      error           ; abort if no pages available

mov     ah,43h           ; try to allocate EMM pages

```

(more)

Figure 9-8. A program skeleton for the use of expanded memory. This code assumes that the presence of the Expanded Memory Manager has already been verified with one of the techniques shown in Figures 9-6 and 9-7.

```

mov     bx,needed_pages
int     67h           ; if allocation is successful
or      ah,ah
jnz     error        ; jump if allocation failed

mov     emm_handle,dx ; save handle for allocated pages
.
.           ; now we are ready for other
.           ; processing using EMM pages
.
.           ; map in EMM memory page...
mov     bx,log_page  ; BX <- EMM logical page number
mov     al,phys_page ; AL <- EMM physical page (0-3)
mov     dx,emm_handle ; EMM handle for our pages
mov     ah,44h       ; Fxn 44H = map EMM page
int     67h
or      ah,ah
jnz     error        ; jump if mapping error

.
.
.           ; program ready to terminate,
.           ; give up allocated EMM pages...
mov     dx,emm_handle ; handle for our pages
mov     ah,45h       ; EMM Fxn 45H = release pages
int     67h
or      ah,ah
jnz     error        ; jump if release failed
.
.

```

Figure 9-8. Continued.

An interrupt handler or resident driver that uses the EMM follows the same general procedure outlined in steps 1 through 8, with a few minor variations. It may need to acquire an EMM handle and allocate pages before the operating system is fully functional; in particular, the MS-DOS services Open File or Device (Interrupt 21H Function 3DH), IOCTL (Interrupt 21H Function 44H), and Get Interrupt Vector (Interrupt 21H Function 35H) cannot be assumed to be available. Thus, such a handler or driver must use a modified version of the "get interrupt vector" technique to test for the existence of the EMM, fetching the contents of the Interrupt 67H vector directly instead of using MS-DOS Interrupt 21H Function 35H.

A device driver or interrupt handler typically owns its expanded memory pages on a permanent basis (until the system is restarted) and never deallocates them. Such a program must also take care to save (EMM Function 47H) and restore (EMM Function 48H) the EMM's page-mapping context (the EMM pages mapped into the page frame at the time the device driver or interrupt handler takes control of the system) so that use of the expanded memory by a foreground program will not be disturbed.

The EMM relies heavily on the good behavior of application software to avoid the corruption of expanded memory. If several applications that use expanded memory are running under a multitasking manager, such as Microsoft Windows, and one or more of those applications does not abide strictly by the EMM's conventions, the data stored in expanded memory can be corrupted.

Extended Memory

Extended memory is that storage at addresses above 1 MB (100000H) that can be accessed by an 80286 or 80386 microprocessor running in protected mode. IBM PC/AT-compatible machines can (theoretically) have as much as 15 MB of extended memory installed, in addition to the usual 1 MB of conventional memory address space. Unlike expanded memory, extended memory is linearly addressable: The address of each memory cell is fixed, so no special manager program is required.

Protected-mode operating systems, such as Microsoft XENIX and MS OS/2, can use extended memory for execution of programs. MS-DOS, on the other hand, runs in real mode on an 80286 or 80386, and programs running under its control cannot ordinarily execute from extended memory or even address that memory for storage of data.

To provide some access to extended memory for real-mode programs, IBM PC/AT-compatible machines contain two routines in their ROM BIOS (Tables 9-4 and 9-5) that allow the amount of extended memory present to be determined (Interrupt 15H Function 88H) and that transfer blocks of data between conventional memory and extended

Table 9-4. IBM PC/AT ROM BIOS Interrupt 15H Functions for Access to Extended Memory.

Interrupt 15H Function	Call With	Returns
Move Extended Memory Block	AH = 87H* CX = length (words) ES:SI = address of block move descriptor table	Carry flag = 0 if successful 1 if error AH = status: 00H no error 01H RAM parity error 02H exception inter- rupt error 03H gate address line 20 failed
Obtain Size of Extended Memory	AH = 88H	AX = kilobytes of memory installed above 1 MB

*Table 9-5 shows the descriptor table format used by Function 87H.

memory (Interrupt 15H Function 87H). These routines can be used by electronic disks (RAMdisks) and by other programs that wish to use extended memory for fast storage and retrieval of information that would otherwise have to be written to a slower physical disk drive.

Table 9-5. Block Move Descriptor Table Format for IBM PC/AT ROM BIOS Interrupt 15H Function 87H (Move Extended Memory Block).

Bytes	Contents
00-0FH	Zero
10-11H	Segment length in bytes ($2 \cdot CX - 1$ or greater)
12-14H	24-bit source address
15H	Access rights byte (93H)
16-17H	Zero
18-19H	Segment length in bytes ($2 \cdot CX - 1$ or greater)
1A-1CH	24-bit destination address
1DH	Access rights byte (93H)
1E-1FH	Zero
20-2FH	Zero

Note: This data structure actually constitutes a global descriptor table (GDT) to be used by the CPU while it is running in protected mode; the zero bytes at offsets 0-0FH and 20-2FH are filled in by the ROM BIOS code before the mode transition. The supplied 24-bit address is a linear address in the range 000000-FFFFFFH (not a segment and offset), with the least significant byte first and the most significant byte last.

Programmers should use these ROM BIOS routines with caution. Data stored in extended memory is volatile; it is lost if the machine is turned off. The transfer of data to or from extended memory involves a switch from real mode to protected mode and back again. This is a relatively slow process on 80286-based machines; in some cases it is only marginally faster than actually reading the data from a fixed disk. In addition, programs that use the ROM BIOS extended memory functions are not compatible with the MS-DOS 3.x Compatibility Box of MS OS/2, nor are they reliable if used for communications or networking.

Finally, a major deficit in these ROM BIOS functions is that they do not make any attempt to arbitrate between two or more programs or device drivers that are using extended memory for temporary storage. For example, if an application program and an installed RAMdisk driver attempt to put data in the same area of extended memory, no error is returned to either program, but the data belonging to one or both may be destroyed.

Figure 9-9 demonstrates the use of the ROM BIOS routines to transfer a block of data from extended memory to conventional memory.

```

                                ; block move descriptor table
bmdt  db      8 dup (0)          ; dummy descriptor
      db      8 dup (0)          ; GDT descriptor
      db      8 dup (0)          ; source segment descriptor
      db      8 dup (0)          ; destination segment descriptor
      db      8 dup (0)          ; BIOS CS segment descriptor
      db      8 dup (0)          ; BIOS SS segment descriptor

buff  db      80h dup (0)        ; buffer to receive data
.
.
.
mov   dx,10h                    ; DX:AX = source extended memory
mov   ax,0                      ; address 100000H (1 MB)
mov   bx,seg buff               ; DS:BX = destination conventional
mov   ds,bx                     ; memory address
mov   bx,offset buff
mov   cx,80h                    ; CX = length to move (bytes)
mov   si,seg bmdt              ; ES:SI = block move descriptor table
mov   es,si
mov   si,offset bmdt
call  getblk                    ; get block from extended memory
or    ah,ah                     ; test status
jnz   error                    ; jump if block move failed
.
.
.

getblk proc  near               ; transfer block from extended
                                ; memory to real memory
                                ; call with
                                ; DX:AX = extended memory address
                                ; DS:BX = destination buffer
                                ;   CX = length (bytes)
                                ; ES:SI = block move descriptor table
                                ; returns
                                ;   AH = 0 if transfer OK
mov   es:[si+10h],cx           ; store length in descriptors
mov   es:[si+18h],cx
                                ; store access rights bytes
mov   byte ptr es:[si+15h],93h
mov   byte ptr es:[si+1dh],93h

```

(more)

Figure 9-9. Demonstration of a block move from extended memory to conventional memory using the ROM BIOS routine. The procedure getblk accepts a source address in extended memory, a destination address in conventional memory, a length in bytes, and the segment and offset of a block move descriptor table. The extended-memory address is a linear 32-bit address, of which only the lower 24 bits are significant; the conventional-memory address is a segment and offset. The getblk routine converts the destination segment and offset to a linear address, builds the appropriate fields in the block move descriptor table, invokes the ROM BIOS routine to perform the transfer, and returns the status in the AH register.

```

; source (extended memory) address
mov     es:[si+12h],ax
mov     es:[si+14h],dl
; destination (conv memory) address
mov     ax,ds
mov     dx,16
mul     dx
add     ax,bx
adc     dx,0
mov     es:[si+1ah],ax
mov     es:[si+1ch],dl

shr     cx,1
mov     ah,87h
int     15h

ret
; back to caller

```

Figure 9-9. Continued.

Summary

Personal computers that run MS-DOS can support as many as three different types of fast, random-access memory (RAM). Each type has specific characteristics and requires different techniques for its management.

Conventional memory is the term used for the 1 MB of linear address space that can be accessed by an 8086 or 8088 microprocessor or by an 80286 or 80386 microprocessor running in real mode. MS-DOS and the programs that execute under its control run in this address space. MS-DOS provides application programs with services to dynamically allocate and release blocks of conventional memory.

As much as 8 MB of expanded memory can be installed in a PC and used for electronic disks, disk caching, and storage of application program data. The memory is made available in 16 KB pages and is administered by a driver program called the Expanded Memory Manager, which provides allocation, mapping, deallocation, and multitasking support.

Extended memory refers to the memory at addresses above 1 MB that can be accessed by an 80286-based or 80386-based microprocessor running in protected mode; it is not available in PCs based on the 8086 or 8088 microprocessors. As much as 15 MB of extended memory can be installed; however, the ROM BIOS services to access the memory are primitive and slow, and no manager is provided to arbitrate between multiple programs that attempt to use the same extended memory addresses for storage.

Ray Duncan

Article 10

The MS-DOS EXEC Function

The MS-DOS system loader, which brings .COM or .EXE files from disk into memory and executes them, can be invoked by any program with the MS-DOS EXEC function (Interrupt 21H Function 4BH). The default MS-DOS command interpreter, COMMAND.COM, uses the EXEC function to load and run its external commands, such as CHKDSK, as well as other application programs. Many popular commercial programs, such as databases and word processors, use EXEC to load and run subsidiary programs (spelling checkers, for example) or to load and run a second copy of COMMAND.COM. This allows a user to run subsidiary programs or enter MS-DOS commands without losing his or her current working context.

When EXEC is used by one program (called the parent) to load and run another (called the child), the parent can pass certain information to the child in the form of a set of strings called the environment, a command line, and two file control blocks. The child program also inherits the parent program's handles for the MS-DOS standard devices and for any other files or character devices the parent has opened (unless the open operation was performed with the "noninheritance" option). Any operations performed by the child on inherited handles, such as seeks or file I/O, also affect the file pointers associated with the parent's handles. A child program can, in turn, load another program, and the cycle can be repeated until the system's memory area is exhausted.

Because MS-DOS is not a multitasking operating system, a child program has complete control of the system until it has finished its work; the parent program is suspended. This type of processing is sometimes called synchronous execution. When the child terminates, the parent regains control and can use another system function call (Interrupt 21H Function 4DH) to obtain the child's return code and determine whether the program terminated normally, because of a critical hardware error, or because the user entered a Control-C.

In addition to loading a child program, EXEC can also be used to load subprograms and overlays for application programs written in assembly language or in a high-level language that does not include an overlay manager in its run-time library. Such overlays typically cannot be run as self-contained programs; most require "helper" routines or data in the application's root segment.

The EXEC function is available only with MS-DOS versions 2.0 and later. With MS-DOS versions 1.x, a parent program can use Interrupt 21H Function 26H to create a program segment prefix for a child but must carry out the loading, relocation, and execution of the child's code and data itself, without any assistance from the operating system.

How EXEC Works

When the EXEC function receives a request to execute a program, it first attempts to locate and open the specified program file. If the file cannot be found, EXEC fails immediately and returns an error code to the caller.

If the file exists, EXEC opens the file, determines its size, and inspects the first block of the file. If the first 2 bytes of the block are the ASCII characters *MZ*, the file is assumed to contain a .EXE load module, and the sizes of the program's code, data, and stack segments are obtained from the .EXE file header. Otherwise, the entire file is assumed to be an absolute load image (a .COM program). The actual filename extension (.COM or .EXE) is ignored in this determination.

At this point, the amount of memory needed to load the program is known, so EXEC attempts to allocate two blocks of memory: one to hold the new program's environment and one to contain the program's code, data, and stack segments. Assuming that enough memory is available to hold the program itself, the amount actually allocated to the program varies with its type. Programs of the .COM type are usually given all the free memory in the system (unless the memory area has previously become fragmented), whereas the amount assigned to a .EXE program is controlled by two fields in the file header, MINALLOC and MAXALLOC, that are set by the Microsoft Object Linker (LINK). See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program; PROGRAMMING TOOLS: The Microsoft Object Linker; PROGRAMMING UTILITIES: LINK.

EXEC then copies the environment from the parent into the memory allocated for child's environment, builds a program segment prefix (PSP) at the base of the child's program memory block, and copies into the child's PSP the command tail and the two default file control blocks passed by the parent. The previous contents of the terminate (Interrupt 22H), Control-C (Interrupt 23H), and critical error (Interrupt 24H) vectors are saved in the new PSP, and the terminate vector is updated so that control will return to the parent program when the child terminates or is aborted.

The actual code and data portions of the child program are then read from the disk file into the program memory block above the newly constructed PSP. If the child is a .EXE program, a relocation table in the file header is used to fix up segment references within the program to reflect its actual load address.

Finally, the EXEC function sets up the CPU registers and stack according to the program type and transfers control to the program. The entry point for a .COM file is always offset 100H within the program memory block (the first byte following the PSP). The entry point for a .EXE file is specified in the file header and can be anywhere within the program. See also PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

When EXEC is used to load and execute an overlay rather than a child program, its operation is much simpler than described above. For an overlay, EXEC does not attempt to allocate memory or build a PSP or environment. It simply loads the contents of the file at the

address specified by the calling program and performs any necessary relocations (if the overlay file has a .EXE header), using a segment value that is also supplied by the caller. EXEC then returns to the program that invoked it, rather than transferring control to the code in the newly loaded file. The requesting program is responsible for calling the overlay at the appropriate location.

Using EXEC to Load a Program

When one program loads and executes another, it must follow these steps:

1. Ensure that enough free memory is available to hold the code, data, and stack of the child program.
2. Set up the information to be passed to EXEC and the child program.
3. Call the MS-DOS EXEC function to run the child program.
4. Recover and examine the child program's termination and return codes.

Making memory available

MS-DOS typically allocates all available memory to a .COM or .EXE program when it is loaded. (The infrequent exceptions to this rule occur when the transient program area is fragmented by the presence of resident data or programs or when a .EXE program is loaded that was linked with the /CPARMAXALLOC switch or modified with EXEMOD.) Therefore, before a program can load another program, it must free any memory it does not need for its own code, data, and stack.

The extra memory is released with a call to the MS-DOS Resize Memory Block function (Interrupt 21H Function 4AH). In this case, the segment address of the parent's PSP is passed in the ES register, and the BX register holds the number of paragraphs of memory the program must retain for its own use. If the prospective parent is a .COM program, it must be certain to move its stack to a safe area if it is reducing its memory allocation to less than 64 KB.

Preparing parameters for EXEC

When used to load and execute a program, the EXEC function must be supplied with two principal parameters:

- The address of the child program's pathname
- The address of a parameter block

The parameter block, in turn, contains the addresses of information to be passed to the child program.

The program name

The pathname for the child program must be an unambiguous, null-terminated (ASCIIIZ) file specification (no wildcard characters). If a path is not included, the current directory is searched for the program; if a drive-specifier is not present, the default drive is used.

The parameter block

The parameter block contains the addresses of four data items (Figure 10-1):

- The environment block
- The command tail
- The two default file control blocks (FCBs)

The position reserved in the parameter block for the pointer to an environment is only 2 bytes and contains a segment address, because an environment is always paragraph aligned (its address is always evenly divisible by 16); a value of 0000H indicates the parent program's environment should be inherited unchanged. The remaining three addresses are all doubleword addresses in the standard Intel format, with an offset value in the lower word and a segment value in the upper word.

To Call

```
AH      = 4BH
AL      = 00H   load and execute child process
         = 03H   load overlay
DS:DX   = segment:offset of ASCIIZ pathname for an executable program file
ES:BX   = segment:offset of parameter block
```

Returns

If function is successful:
Carry flag is clear.
Other registers are preserved if MS-DOS version 3.0 or later, destroyed if MS-DOS versions 2.x.

If function is not successful:
Carry flag is set.

AX = error code

Parameter Block Format

Offset	Contents
If AL = 00H (load and execute program):	
00H	Segment pointer of the environment to be passed
02H	Offset of command-line tail for the new PSP
04H	Segment of command-line tail for the new PSP
06H	Offset of first file control block, to be copied into new PSP at offset 5CH
08H	Segment of first file control block
0AH	Offset of second file control block, to be copied into new PSP at offset 6CH
0CH	Segment of second file control block
If AL = 03H (load overlay):	
00H	Segment address where overlay is to be loaded
02H	Relocation factor to apply to loaded image

Figure 10-1. Synopsis of calling conventions for the MS-DOS EXEC function (Interrupt 21H Function 4BH), which can be used to load and execute child processes or overlays.

The environment

An environment always begins on a paragraph boundary and is composed of a series of null-terminated (ASCIIZ) strings of the form:

name=variable

The end of the entire set of strings is indicated by an additional null byte.

If the environment pointer in the parameter block supplied to an EXEC call contains zero, the child simply acquires a copy of the parent's environment. The parent can, however, provide a segment pointer to a different or expanded set of strings. In either case, under MS-DOS versions 3.0 and later, EXEC appends the child program's fully qualified path-name to its environment block. The maximum size of an environment is 32 KB, so very large amounts of information can be passed between programs by this mechanism.

The original, or master, environment for the system is owned by the command processor that is loaded when the system is turned on or restarted (usually COMMAND.COM). Strings are placed in the system's master environment by COMMAND.COM as a result of PATH, SHELL, PROMPT, and SET commands, with default values always present for the first two. For example, if an MS-DOS version 3.2 system is started from drive C and a PATH command is not present in the AUTOEXEC.BAT file nor a SHELL command in the CONFIG.SYS file, the master environment will contain the two strings:

```
PATH=
COMSPEC=C:\COMMAND.COM
```

These specifications are used by COMMAND.COM to search for executable "external" commands and to find its own executable file on the disk so that it can reload its transient portion when necessary. When the PROMPT string is present (as a result of a previous PROMPT or SET PROMPT command), COMMAND.COM uses it to tailor the prompt displayed to the user.

```

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0000 43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
0010 4E 44 2E 43 4F 4D 00 50 52 4F 4D 50 54 3D 24 70 ND.COM.PROMPT=$p
0020 24 5F 24 64 20 20 20 24 74 24 68 24 68 24 68 24 $_$d $t$h$h$h$h$
0030 68 24 68 24 68 20 24 71 24 71 24 67 00 50 41 54 h$h$h $q$q$g.PAT
0040 48 3D 43 3A 5C 53 59 53 54 45 4D 3B 43 3A 5C 41 H=C:\SYSTEM;C:\A
0050 53 4D 3B 43 3A 5C 57 53 3B 43 3A 5C 45 54 48 45 SM;C:\WS;C:\ETHE
0060 52 4E 45 54 3B 43 3A 5C 46 4F 52 54 48 5C 50 43 RNET;C:\FORTH\PC
0070 33 31 3B 00 00 01 00 43 3A 5C 46 4F 52 54 48 5C 31;...C:\FORTH\
0080 50 43 33 31 5C 46 4F 52 54 48 2E 43 4F 4D 00 PC31\FORTH.COM.
```

Figure 10-2. Dump of a typical environment under MS-DOS version 3.2. This particular example contains the default COMSPEC parameter and two relatively complex PATH and PROMPT control strings that were set up by entries in the user's AUTOEXEC file. Note the two null bytes at offset 73H, which indicate the end of the environment. These bytes are followed by the pathname of the program that owns the environment.

Other strings in the environment are used only for informational purposes by transient programs and do not affect the operation of the operating system proper. For example, the Microsoft C Compiler and the Microsoft Object Linker look in the environment for INCLUDE, LIB, and TMP strings that specify the location of *include* files, library files, and temporary working files. Figure 10-2 contains a hex dump of a typical environment block.

The command tail

The command tail to be passed to the child program takes the form of a byte indicating the length of the remainder of the command tail, followed by a string of ASCII characters terminated with an ASCII carriage return (0DH); the carriage return is not included in the length byte. The command tail can include switches, filenames, and other parameters that can be inspected by the child program and used to influence its operation. It is copied into the child program's PSP at offset 80H.

When COMMAND.COM uses EXEC to run a program, it passes a command tail that includes everything the user typed in the command line except the name of the program and any redirection parameters. I/O redirection is processed within COMMAND.COM itself and is manifest in the behavior of the standard device handles that are inherited by the child program. Any other program that uses EXEC to run a child program must try to perform any necessary redirection on its own and must supply an appropriate command tail so that the child program will behave as though it had been loaded by COMMAND.COM.

The default file control blocks

The two default FCBs pointed to by the EXEC parameter block are copied into the child program's PSP at offsets 5CH and 6CH. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

Few of the currently popular application programs use FCBs for file and record I/O because FCBs do not support the hierarchical directory structure. But some programs do inspect the default FCBs as a quick way to isolate the first two switches or other parameters from the command tail. Therefore, to make its own identity transparent to the child program, the parent should emulate the action of COMMAND.COM by parsing the first two parameters of the command tail into the default FCBs. This can be conveniently accomplished with the MS-DOS function Parse Filename (Interrupt 21H Function 29H).

If the child program does not require one or both of the default FCBs, the corresponding address in the parameter block can be initialized to point to two dummy FCBs in the application's memory space. These dummy FCBs should consist of 1 zero byte followed by 11 bytes containing ASCII blank characters (20H).

Running the child program

After the parent program has constructed the necessary parameters, it can invoke the EXEC function by issuing Interrupt 21H with the registers set as follows:

```
AH      = 4BH
AL      = 00H (EXEC subfunction to load and execute program)
DS:DX   = segment:offset of program pathname
ES:BX   = segment:offset of parameter block
```

Upon return from the software interrupt, the parent must test the carry flag to determine whether the child program did, in fact, run. If the carry flag is clear, the child program was successfully loaded and given control. If the carry flag is set, the EXEC function failed, and the error code returned in AX can be examined to determine why. The usual reasons are

- The specified file could not be found.
- The file was found, but not enough memory was free to load it.

Other causes are uncommon and can be symptoms of more severe problems in the system as a whole (such as damage to disk files or to the memory image of MS-DOS). With MS-DOS versions 3.0 and later, additional details about the cause of an EXEC failure can be obtained by subsequently calling Interrupt 21H Function 59H (Get Extended Error Information).

In general, supplying either an invalid address for an EXEC parameter block or invalid addresses within the parameter block itself does *not* cause a failure of the EXEC function, but may result in the child program behaving in unexpected ways.

Special considerations

With MS-DOS versions 2.x, the previous contents of all the parent registers except for CS:IP can be destroyed after an EXEC call, including the stack pointer in SS:SP. Consequently, before issuing the EXEC call, the parent must push onto the stack the contents of any registers that it needs to preserve, and then it must save the stack segment and offset in a location that is addressable with the CS segment register. Upon return, the stack segment and offset can be loaded into SS:SP with code segment overrides, and then the other registers can be restored by popping them off the stack. With MS-DOS versions 3.0 and later, registers are preserved across an EXEC call in the usual fashion.

Note: The code segments of Windows applications that use this technique should be given the IMPURE attribute.

In addition, a bug in MS-DOS version 2.0 and in PC-DOS versions 2.0 and 2.1 causes an arbitrary doubleword in the parent's stack segment to be destroyed during an EXEC call. When the parent is a .COM program and SS = PSP, the damaged location falls within the PSP and does no harm; however, in the case of a .EXE parent where DS = SS, the affected location may overlap the data segment and cause aberrant behavior or even a crash after the return from EXEC. This bug was fixed in MS-DOS versions 2.11 and later and in PC-DOS versions 3.0 and later.

Examining the child program's return codes

If the EXEC function succeeds, the parent program can call Interrupt 21H Function 4DH (Get Return Code of Child Process) to learn whether the child executed normally to completion and passed back a return code or was terminated by the operating system because of an external event. Function 4DH returns:

AH = termination type:

- 00H Child terminated normally (that is, exited via Interrupt 20H or Interrupt 21H Function 00H or Function 4CH).
- 01H Child was terminated by user's entry of a Ctrl-C.
- 02H Child was terminated by critical error handler (either the user responded with *A* to the *Abort, Retry, Ignore* prompt from the system's default Interrupt 24H handler, or a custom Interrupt 24H handler returned to MS-DOS with action code = 02H in register AL).
- 03H Child terminated normally and stayed resident (that is, exited via Interrupt 21H Function 31H or Interrupt 27H).

AL = return code:

Value passed by the child program in register AL when it terminated with Interrupt 21H Function 4CH or 31H.

00H if the child terminated using Interrupt 20H, Interrupt 27H, or Interrupt 21H Function 00H.

These values are only guaranteed to be returned once by Function 4DH. Thus, a subsequent call to Function 4DH, without an intervening EXEC call, does not necessarily return any useful information. Additionally, if Function 4DH is called without a preceding successful EXEC call, the returned values are meaningless.

Using COMMAND.COM with EXEC

An application program can "shell" to MS-DOS — that is, provide the user with an MS-DOS prompt without terminating — by using EXEC to load and execute a secondary copy of COMMAND.COM with an empty command tail. The application can obtain the location of the COMMAND.COM disk file by inspecting its own environment for the COMSPEC string. The user returns to the application from the secondary command processor by typing *exit* at the COMMAND.COM prompt.

Batch-file interpretation is carried out by COMMAND.COM, and a batch (.BAT) file cannot be called using the EXEC function directly. Similarly, the sequential search for .COM, .EXE, and .BAT files in all the locations specified in the environment's PATH variable is a function of COMMAND.COM, rather than of EXEC. To execute a batch file or search the system path for a program, an application program can use EXEC to load and execute a secondary copy of COMMAND.COM to use as an intermediary. The application finds the location of COMMAND.COM as described in the preceding paragraph, but it passes a command tail in the form:

```
/C program parameter1 parameter2 ...
```


where *program* is the .EXE, .COM, or .BAT file to be executed. When *program* terminates, the secondary copy of COMMAND.COM exits and returns control to the parent.

A parent and child example

The source programs PARENT.ASM in Figure 10-3 and CHILD.ASM in Figure 10-4 illustrate how one program uses EXEC to load another.

```

        name      parent
        title     'PARENT --- demonstrate EXEC call'
;
; PARENT.EXE --- demonstration of EXEC to run process
;
; Uses MS-DOS EXEC (Int 21H Function 4BH Subfunction 00H)
; to load and execute a child process named CHILD.EXE,
; then displays CHILD's return code.
;
; Ray Duncan, June 1987
;

stdin  equ      0           ; standard input
stdout equ      1           ; standard output
stderr equ      2           ; standard error

stksize equ     128         ; size of stack

cr     equ      0dh         ; ASCII carriage return
lf     equ      0ah         ; ASCII linefeed

DGROUP group     _DATA, _ENVIR, _STACK

_TEXT  segment byte public 'CODE'      ; executable code segment

        assume  cs:_TEXT, ds:_DATA, ss:_STACK

stk_seg dw       ?           ; original SS contents
stk_ptr dw       ?           ; original SP contents

main   proc      far         ; entry point from MS-DOS

        mov     ax, _DATA     ; set DS = our data segment
        mov     ds, ax

; now give back extra memory
; so child has somewhere to run...

```

Figure 10-3. PARENT.ASM, source code for PARENT.EXE.

(more)

```

mov     ax,es                ; let AX = segment of PSP base
mov     bx,ss                ; and BX = segment of stack base
sub     bx,ax                ; reserve seg stack - seg psp
add     bx,stksize/16       ; plus paragraphs of stack
mov     ah,4ah              ; fxn 4AH = modify memory block
int     21h
jc      main1

                                ; display parent message ...
mov     dx,offset DGROUP:msg1 ; DS:DX = address of message
mov     cx,msg1_len         ; CX = length of message
call    pmsg

push    ds                  ; save parent's data segment
mov     stk_seg,ss          ; save parent's stack pointer
mov     stk_ptr,sp

                                ; now EXEC the child process...
mov     ax,ds                ; set ES = DS
mov     es,ax
mov     dx,offset DGROUP:cname ; DS:DX = child pathname
mov     bx,offset DGROUP:parm ; ES:BX = parameter block
mov     ax,4b00h            ; function 4BH subfunction 00H
int     21h                 ; transfer to MS-DOS

cli                                           ; (for bug in some early 8088s)
mov     ss,stk_seg           ; restore parent's stack pointer
mov     sp,stk_ptr
sti                                           ; (for bug in some early 8088s)
pop     ds                   ; restore DS = our data segment

jc      main2                ; jump if EXEC failed

                                ; otherwise EXEC succeeded,
                                ; convert and display child's
                                ; termination and return codes...
mov     ah,4dh              ; fxn 4DH = get return code
int     21h                 ; transfer to MS-DOS
xchg    al,ah               ; convert termination code
mov     bx,offset DGROUP:msg4a
call    b2hex
mov     al,ah                ; get back return code
mov     bx,offset DGROUP:msg4b ; and convert it
call    b2hex
mov     dx,offset DGROUP:msg4 ; DS:DX = address of message
mov     cx,msg4_len         ; CX = length of message
call    pmsg                ; display it

mov     ax,4c00h            ; no error, terminate program
int     21h                 ; with return code = 0

```

Figure 10-3. Continued.

(more)

```

main1:  mov     bx,offset DGROUP:msg2a ; convert error code
        call   b2hex
        mov     dx,offset DGROUP:msg2 ; display message 'Memory
        mov     cx,msg2_len           ; resize failed...'
        call   pmsg
        jmp    main3

main2:  mov     bx,offset DGROUP:msg3a ; convert error code
        call   b2hex
        mov     dx,offset DGROUP:msg3 ; display message 'EXEC
        mov     cx,msg3_len           ; call failed...'
        call   pmsg

main3:  mov     ax,4c01h                ; error, terminate program
        int    21h                    ; with return code = 1

main    endp                          ; end of main procedure

b2hex  proc   near                    ; convert byte to hex ASCII
                                           ; call with AL = binary value
                                           ;         BX = addr to store string
        push  ax
        shr   al,1
        shr   al,1
        shr   al,1
        shr   al,1
        call  ascii                    ; become first ASCII character
        mov   [bx],al                  ; store it
        pop  ax
        and  al,0fh                    ; isolate lower 4 bits, which
        call  ascii                    ; become the second ASCII character
        mov   [bx+1],al                ; store it
        ret
b2hex  endp

ascii  proc   near                    ; convert value 00-0FH in AL
                                           ; into a "hex ASCII" character
        add  al,'0'
        cmp  al,'9'
        jle  ascii2                    ; jump if in range 00-09H,
        add  al,'A'-'9'-1              ; offset it to range 0A-0FH,

ascii2: ret                             ; return ASCII char. in AL
ascii  endp

pmsg   proc   near                    ; displays message on standard output
                                           ; call with DS:DX = address,
                                           ;         CX = length

```

Figure 10-3. Continued.

(more)

```

        mov     bx,stdout           ; BX = standard output handle
        mov     ah,40h             ; function 40H = write file/device
        int     21h               ; transfer to MS-DOS
        ret                          ; back to caller

pmsg    endp

_TEXT   ends

_DATA   segment para public 'DATA' ; static & variable data segment

cname   db     'CHILD.EXE',0      ; pathname of child process

pars    dw     _ENVIR             ; segment of environment block
        dd     tail               ; long address, command tail
        dd     fcb1               ; long address, default FCB #1
        dd     fcb2               ; long address, default FCB #2

tail    db     fcb1-tail-2        ; command tail for child
        db     'dummy command tail',cr

fcb1    db     0                  ; copied into default FCB #1 in
        db     11 dup (' ')       ; child's program segment prefix
        db     25 dup (0)

fcb2    db     0                  ; copied into default FCB #2 in
        db     11 dup (' ')       ; child's program segment prefix
        db     25 dup (0)

msg1    db     cr,lf,'Parent executing!',cr,lf
msg1_len equ  $-msg1

msg2    db     cr,lf,'Memory resize failed, error code='
msg2a   db     'xxh.',cr,lf
msg2_len equ  $-msg2

msg3    db     cr,lf,'EXEC call failed, error code='
msg3a   db     'xxh.',cr,lf
msg3_len equ  $-msg3

msg4    db     cr,lf,'Parent regained control!'
        db     cr,lf,'Child termination type='
msg4a   db     'xxh, return code='
msg4b   db     'xxh.',cr,lf
msg4_len equ  $-msg4

_DATA   ends

_ENVIR  segment para public 'DATA' ; example environment block
        ; to be passed to child

```

Figure 10-3. Continued.

(more)

```

        db      'PATH=',0          ; basic PATH, PROMPT,
        db      'PROMPT=$p$_$n$g',0 ; and COMSPEC strings
        db      'COMSPEC=C:\COMMAND.COM',0
        db      0                  ; extra null terminates block

_ENVIR ends

_STACK segment para stack 'STACK'

        db      stksize dup (?)

_STACK ends

        end      main              ; defines program entry point

```

Figure 10-3. Continued.

```

        name     child
        title    'CHILD process'
;
; CHILD.EXE --- a simple process loaded by PARENT.EXE
; to demonstrate the MS-DOS EXEC call, Subfunction 00H.
;
; Ray Duncan, June 1987
;

stdin  equ      0          ; standard input
stdout equ      1          ; standard output
stderr equ      2          ; standard error

cr     equ      0dh        ; ASCII carriage return
lf     equ      0ah        ; ASCII linefeed

DGROUP group  _DATA,STACK

_TEXT  segment byte public 'CODE' ; executable code segment

        assume  cs:_TEXT,ds:_DATA,ss:STACK

main   proc     far        ; entry point from MS-DOS

        mov     ax,_DATA    ; set DS = our data segment
        mov     ds,ax

; display child message ...

```

Figure 10-4. CHILD.ASM, source code for CHILD.EXE.

(more)

```

        mov     dx,offset msg           ; DS:DX = address of message
        mov     cx,msg_len             ; CX = length of message
        mov     bx,stdout              ; BX = standard output handle
        mov     ah,40h                 ; AH = fxn 40H, write file/device
        int     21h                    ; transfer to MS-DOS
        jc     .main2                  ; jump if any error

        mov     ax,4c00h               ; no error, terminate child
        int     21h                    ; with return code = 0

main2:   mov     ax,4c01h               ; error, terminate child
        int     21h                    ; with return code = 1

main     endp                          ; end of main procedure

_TEXT   ends

_DATA   segment para public 'DATA'     ; static & variable data segment

msg     db     cr,lf,'Child executing!',cr,lf
msg_len equ    $-msg

_DATA   ends

STACK   segment para stack 'STACK'

        dw     64 dup (?)

STACK   ends

        end     main                   ; defines program entry point

```

Figure 10-4. Continued.

PARENT.ASM can be assembled and linked into the executable program PARENT.EXE with the following commands:

```

C>MASM PARENT; <Enter>
C>LINK PARENT; <Enter>

```

Similarly, CHILD.ASM can be assembled and linked into the file CHILD.EXE as follows:

```

C>MASM CHILD; <Enter>
C>LINK CHILD; <Enter>

```

When PARENT.EXE is executed with the command

```

C>PARENT <Enter>

```

PARENT reduces the size of its main memory block with a call to Interrupt 21H Function 4AH, to maximize the amount of free memory in the system, and then calls the EXEC function to load and execute CHILD.EXE.

CHILD.EXE runs exactly as though it had been loaded directly by COMMAND.COM. CHILD resets the DS segment register to point to its own data segment, uses Interrupt 21H Function 40H to display a message on standard output, and then terminates using Interrupt 21H Function 4CH, passing a return code of zero.

When PARENT.EXE regains control, it first checks the carry flag to determine whether the EXEC call succeeded. If the EXEC call failed, PARENT displays an error message and terminates with Interrupt 21H Function 4CH, itself passing a nonzero return code to COMMAND.COM to indicate an error.

Otherwise, PARENT uses Interrupt 21H Function 4DH to obtain CHILD.EXE's termination type and return code, which it converts to ASCII and displays. PARENT then terminates using Interrupt 21H Function 4CH and passes a return code of zero to COMMAND.COM to indicate success. COMMAND.COM in turn receives control and displays a new user prompt.

Using EXEC to Load Overlays

Loading overlays with the EXEC function is much less complex than using EXEC to run another program. The main program, called the root segment, must carry out the following steps to load and execute an overlay:

1. Make a memory block available to receive the overlay.
2. Set up the overlay parameter block to be passed to the EXEC function.
3. Call the EXEC function to load the overlay.
4. Execute the code within the overlay by transferring to it with a far call.

The overlay itself can be constructed as either a memory image (.COM) or a relocatable (.EXE) file and need not be the same type as the root program. In either case, the overlay should be designed so that the entry point (or a pointer to the entry point) is at the beginning of the module after it is loaded. This allows the root and overlay modules to be maintained separately and avoids a need for the root to have "magical" knowledge of addresses within the overlay.

To prevent users from inadvertently running an overlay directly from the command line, overlay files should be assigned an extension other than .COM or .EXE. The most convenient method relates overlays to their root segment by assigning them the same filename but an extension such as .OVL or .OV1, .OV2, and so on.

Making memory available

If EXEC is to load a child program successfully, the parent must release memory. In contrast, EXEC loads an overlay into memory that *belongs* to the calling program. If the

root segment is a .COM program and has not explicitly released extra memory, the root segment program need only ensure that the system contains enough memory to load the overlay and that the overlay load address does not conflict with its own code, data, or stack areas.

If the root segment program was loaded from a .EXE file, no straightforward way exists for it to determine unequivocally how much memory it already owns. The simplest course is for the program to release all extra memory, as discussed earlier in the section on loading a child program, and then use the MS-DOS memory allocation function (Interrupt 21H Function 48H) to obtain a new block of memory that is large enough to hold the overlay.

Preparing overlay parameters

When it is used to load an overlay, the EXEC function requires two major parameters:

- The address of the pathname for the overlay file
- The address of an overlay parameter block

As for a child program, the pathname for the overlay file must be an unambiguous ASCIIZ file specification (again, no wildcard characters), and it must include an explicit extension. As before, if a path and/or drive are not included in the pathname, the current directory and default drive are used.

The overlay parameter block contains the segment address at which the overlay should be loaded and a fixup value to be applied to any relocatable items within the overlay file. If the overlay file is in .EXE format, the fixup value is typically the same as the load address; if the overlay is in memory-image (.COM) format, the fixup value should be zero. The EXEC function does not attempt to validate the load address or the fixup value or to ensure that the load address actually belongs to the calling program.

Loading and executing the overlay

After the root segment program has prepared the filename of the overlay file and the overlay parameter block, it can invoke the EXEC function to load the overlay by issuing an Interrupt 21H with the registers set as follows:

AH	= 4BH
AL	= 03H (EXEC subfunction to load overlay)
DS:DX	= segment:offset of overlay file pathname
ES:BX	= segment:offset of overlay parameter block

Upon return from Interrupt 21H, the root segment must test the carry flag to determine whether the overlay was loaded. If the carry flag is clear, the overlay file was located and brought into memory at the requested address. The overlay can then be entered by a far call and should exit back to the root segment with a far return.

If the carry flag is set, the overlay file was not found or some other (probably severe) system problem was encountered, and the AX register contains an error code. With MS-DOS

versions 3.0 and later, Interrupt 21H Function 59H can be used to get more information about the EXEC failure. An invalid load address supplied in the overlay parameter block does not (usually) cause the EXEC function itself to fail but may result in the disconcerting message *Memory Allocation Error, System Halted* when the root program terminates.

An overlay example

The source programs ROOT.ASM in Figure 10-5 and OVERLAY.ASM in Figure 10-6 demonstrate the use of EXEC to load a program overlay. The program ROOT.EXE is executable from the MS-DOS prompt; it represents the root segment of an application. OVERLAY is constructed as a .EXE file (although it is named OVERLAY.OVL because it cannot be run alone) and represents a subprogram that can be loaded by the root segment when and if it is needed.

```

        name      root
        title     'ROOT --- demonstrate EXEC overlay'
;
; ROOT.EXE --- demonstration of EXEC for overlays
;
; Uses MS-DOS EXEC (Int 21H Function 4BH Subfunction 03H)
; to load an overlay named OVERLAY.OVL, calls a routine
; within the OVERLAY, then recovers control and terminates.
;
; Ray Duncan, June 1987.
;

stdin  equ      0                ; standard input
stdout equ      1                ; standard output
stderr equ      2                ; standard error

stksize equ     128              ; size of stack

cr      equ     0dh              ; ASCII carriage return
lf      equ     0ah              ; ASCII linefeed

DGROUP group    _DATA,_STACK

_TEXT   segment byte public 'CODE' ; executable code segment
        assume  cs:_TEXT,ds:_DATA,ss:_STACK

stk_seg dw      ?                ; original SS contents
stk_ptr dw      ?                ; original SP contents

```

Figure 10-5. ROOT.ASM, source code for ROOT.EXE.

(more)

```

main    proc    far                ; entry point from MS-DOS

        mov     ax,_DATA           ; set DS = our data segment
        mov     ds,ax

                                           ; now give back extra memory
        mov     ax,es              ; AX = segment of PSP base
        mov     bx,ss              ; BX = segment of stack base
        sub     bx,ax              ; reserve seg stack - seg psp
        add     bx,stksize/16      ; plus paragraphs of stack
        mov     ah,4ah             ; fxn 4AH = modify memory block
        int     21h                ; transfer to MS-DOS
        jc     main1               ; jump if resize failed

                                           ; display message 'Root
                                           ; segment executing...'
        mov     dx,offset DGROUP:msg1 ; DS:DX = address of message
        mov     cx,msg1_len        ; CX = length of message
        call    pmsg

                                           ; allocate memory for overlay
        mov     bx,1000h           ; get 64 KB (4096 paragraphs)
        mov     ah,48h            ; fxn 48H, allocate mem block
        int     21h                ; transfer to MS-DOS
        jc     main2               ; jump if allocation failed

        mov     pars,ax            ; set load address for overlay
        mov     pars+2,ax          ; set relocation segment for overlay
        mov     word ptr entry+2,ax ; set segment of entry point

        push    ds                 ; save root's data segment
        mov     stk_seg,ss         ; save root's stack pointer
        mov     stk_ptr,sp

                                           ; now use EXEC to load overlay
        mov     ax,ds              ; set ES = DS
        mov     es,ax
        mov     dx,offset DGROUP:oname ; DS:DX = overlay pathname
        mov     bx,offset DGROUP:pars ; ES:BX = parameter block
        mov     ax,4b03h           ; function 4BH, subfunction 03H
        int     21h                ; transfer to MS-DOS

        cli                       ; (for bug in some early 8088s)
        mov     ss,stk_seg         ; restore root's stack pointer
        mov     sp,stk_ptr
        sti                       ; (for bug in some early 8088s)
        pop     ds                 ; restore DS = our data segment

        jc     main3               ; jump if EXEC failed

                                           ; otherwise EXEC succeeded...

```

Figure 10-5. Continued.

(more)

```

    push    ds                ; save our data segment
    call   dword ptr entry   ; now call the overlay
    pop    ds                ; restore our data segment

                                ; display message that root
                                ; segment regained control...
    mov    dx,offset DGROUP:msg5 ; DS:DX = address of message
    mov    cx,msg5_len        ; CX = length of message
    call   pmsg               ; display it

    mov    ax,4c00h          ; no error, terminate program
    int   21h                ; with return code = 0

main1:  mov    bx,offset DGROUP:msg2a ; convert error code
        call   b2hex
        mov    dx,offset DGROUP:msg2 ; display message 'Memory
        mov    cx,msg2_len          ; resize failed...'
        call   pmsg
        jmp    main4

main2:  mov    bx,offset DGROUP:msg3a ; convert error code
        call   b2hex
        mov    dx,offset DGROUP:msg3 ; display message 'Memory
        mov    cx,msg3_len          ; allocation failed...'
        call   pmsg
        jmp    main4

main3:  mov    bx,offset DGROUP:msg4a ; convert error code
        call   b2hex
        mov    dx,offset DGROUP:msg4 ; display message 'EXEC
        mov    cx,msg4_len          ; call failed...'
        call   pmsg

main4:  mov    ax,4c01h          ; error, terminate program
        int   21h                ; with return code = 1

main    endp                ; end of main procedure

b2hex  proc    near          ; convert byte to hex ASCII
                                ; call with AL = binary value
                                ; BX = addr to store string
        push   ax
        shr   al,1
        shr   al,1
        shr   al,1
        shr   al,1
        call  ascii          ; become first ASCII character
        mov   [bx],al        ; store it
        pop   ax

```

Figure 10-5. Continued.

(more)

```

        and    al,0fh                ; isolate lower 4 bits, which
        call   ascii                ; become the second ASCII character
        mov    [bx+1],al            ; store it
        ret
b2hex  endp

ascii  proc    near                ; convert value 00-0FH in AL
        add    al,'0'                ; into a "hex ASCII" character
        cmp    al,'9'
        jle    ascii2              ; jump if in range 00-09H,
        add    al,'A'-'9'-1         ; offset it to range 0A-0FH,
ascii2: ret                        ; return ASCII char. in AL.
ascii  endp

pmsg   proc    near                ; displays message on standard output
                                           ; call with DS:DX = address,
                                           ;           CX = length

        mov    bx,stdout            ; BX = standard output handle
        mov    ah,40h              ; function 40H = write file/device
        int    21h                 ; transfer to MS-DOS
        ret                        ; back to caller

pmsg   endp

_TEXT  ends

_DATA  segment para public 'DATA'    ; static & variable data segment

oname  db      'OVERLAY.OVL',0      ; pathname of overlay file

pars   dw      0                    ; load address (segment) for file
        dw      0                    ; relocation (segment) for file

entry  dd      0                    ; entry point for overlay

msg1   db      cr,lf,'Root segment executing!',cr,lf
msg1_len equ  $-msg1

msg2   db      cr,lf,'Memory resize failed, error code='
msg2a  db      'xxh.',cr,lf
msg2_len equ  $-msg2

msg3   db      cr,lf,'Memory allocation failed, error code='
msg3a  db      'xxh.',cr,lf
msg3_len equ  $-msg3

```

Figure 10-5. Continued.

(more)

```

msg4    db    cr,lf,'EXEC call failed, error code='
msg4a   db    'xxh.',cr,lf
msg4_len equ  $-msg4

msg5    db    cr,lf,'Root segment regained control!',cr,lf
msg5_len equ  $-msg5

_DATA   ends

_STACK  segment para stack 'STACK'

        db    stksize dup (?)

_STACK  ends

        end    main                ; defines program entry point

```

Figure 10-5. Continued.

```

        name    overlay
        title   'OVERLAY segment'
;
; OVERLAY.OVL --- a simple overlay segment
; loaded by ROOT.EXE to demonstrate use of
; the MS-DOS EXEC call Subfunction 03H.
;
; The overlay does not contain a STACK segment
; because it uses the ROOT segment's stack.
;
; Ray Duncan, June 1987
;

stdin   equ    0                ; standard input
stdout  equ    1                ; standard output
stderr  equ    2                ; standard error

cr      equ    0dh              ; ASCII carriage return
lf      equ    0ah              ; ASCII linefeed

_TEXT   segment byte public 'CODE' ; executable code segment

        assume cs:_TEXT,ds:_DATA
ovlay   proc    far                ; entry point from root segment

        mov    ax,_DATA          ; set DS = local data segment
        mov    ds,ax

```

Figure 10-6. OVERLAY.ASM, source code for OVERLAY.OVL.

(more)

```

                                ; display overlay message ...
    mov     dx,offset msg         ; DS:DX = address of message
    mov     cx,msg_len           ; CX = length of message
    mov     bx,stdout            ; BX = standard output handle
    mov     ah,40h               ; AH = fxn 40H, write file/device
    int     21h                 ; transfer to MS-DOS

    ret                           ; return to root segment

ovlay     endp                   ; end of ovlay procedure

_TEXT    ends

_DATA    segment para public 'DATA' ; static & variable data segment

msg      db      cr,lf,'Overlay executing!',cr,lf
msg_len  equ     $-msg

_DATA    ends

        end

```

Figure 10-6. Continued.

ROOT.ASM can be assembled and linked into the executable program ROOT.EXE with the following commands:

```

C>MASM ROOT; <Enter>
C>LINK ROOT; <Enter>

```

OVERLAY.ASM can be assembled and linked into the file OVERLAY.OVL by typing

```

C>MASM OVERLAY; <Enter>
C>LINK OVERLAY,OVERLAY.OVL; <Enter>

```

The Microsoft Object Linker will display the message

```
Warning: no stack segment
```

but this message can be ignored.

When ROOT.EXE is executed with the command

```
C>ROOT <Enter>
```

it first shrinks its main memory block with a call to Interrupt 21H Function 4AH and then allocates a separate block for the overlay with Interrupt 21H Function 48H. Next, ROOT calls the EXEC function to load the file OVERLAY.OVL into the newly allocated memory block. If the EXEC function fails, ROOT displays an error message and terminates with Interrupt 21H Function 4CH, passing a nonzero return code to COMMAND.COM to indicate an error. If the EXEC function succeeds, ROOT saves the contents of its DS segment register and then enters the overlay through an indirect far call.

The overlay resets the DS segment register to point to its own data segment, displays a message using Interrupt 21H Function 40H, and then returns. Note that the main procedure of the overlay is declared with the far attribute to force the assembler to generate the opcode for a far return.

When ROOT regains control, it restores the DS segment register to point to its own data segment again and displays an additional message, also using Interrupt 21H Function 40H, to indicate that the overlay executed successfully. ROOT then terminates using Interrupt 21H Function 4CH, passing a return code of zero to indicate success, and control returns to COMMAND.COM.

Ray Duncan



Part C
Customizing MS-DOS

Article 11

Terminate-and-Stay-Resident Utilities

The MS-DOS Terminate and Stay Resident system calls (Interrupt 21H Function 31H and Interrupt 27H) allow the programmer to install executable code or program data in a reserved block of RAM, where it resides while other programs execute. Global data, interrupt handlers, and entire applications can be made RAM-resident in this way. Programs that use the MS-DOS terminate-and-stay-resident capability are commonly known as TSR programs or TSRs.

This article describes how to install a TSR in RAM, how to communicate with the resident program, and how the resident program can interact with MS-DOS. The discussion proceeds from a general description of the MS-DOS functions useful to TSR programmers to specific details about certain MS-DOS structural elements necessary to proper functioning of a TSR utility and concludes with two programming examples.

Note: Microsoft cannot guarantee that the information in this article will be valid for future versions of MS-DOS.

Structure of a Terminate-and-Stay-Resident Utility

The executable code and data in TSRs can be separated into RAM-resident and transient portions (Figure 11-1). The RAM-resident portion of a TSR contains executable code and data for an application that performs some useful function on demand. The transient portion installs the TSR; that is, it initializes data and interrupt handlers contained in the RAM-resident portion of the program and executes an MS-DOS Terminate and Stay Resident function call that leaves the RAM-resident portion in memory and frees the memory used by the transient portion. The code in the transient portion of a TSR runs when the .EXE or .COM file containing the program is executed; the code in the RAM-resident portion runs only when it is explicitly invoked by a foreground program or by execution of a hardware or software interrupt.

TSRs can be broadly classified as passive or active, depending on the method by which control is transferred to the RAM-resident program. A passive TSR executes only when another program explicitly transfers control to it, either through a software interrupt or by means of a long JMP or CALL. The calling program is not interrupted by the TSR, so the status of MS-DOS, the system BIOS, and the hardware is well defined when the TSR program starts to execute.

In contrast, an active TSR is invoked by the occurrence of some event external to the currently running (foreground) program, such as a sequence of user keystrokes or a pre-defined hardware interrupt. Therefore, when it is invoked, an active TSR almost always

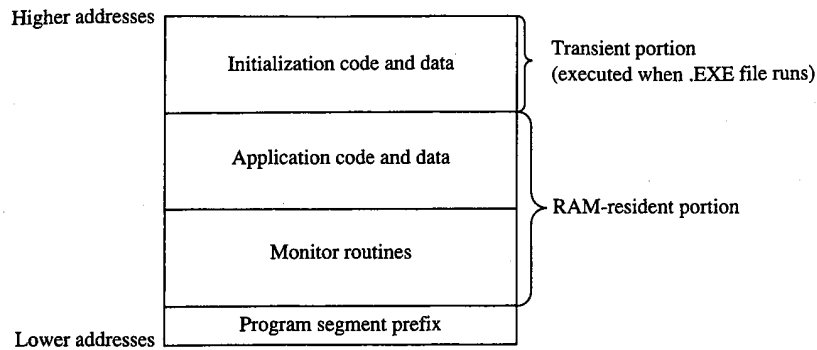


Figure 11-1. Organization of a TSR program in memory.

interrupts some other program and suspends its execution. To avoid disrupting the interrupted program, an active TSR must monitor the status of MS-DOS, the ROM BIOS, and the hardware and take control of the system only when it is safe to do so.

Passive TSRs are generally simpler in their construction than active TSRs because a passive TSR runs in the context of the calling program; that is, when the TSR executes, it assumes that it can use the calling program's program segment prefix (PSP), open files, current directory, and so on. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. It is the calling program's responsibility to ensure that the hardware and MS-DOS are in a stable state before it transfers control to a passive TSR.

An active TSR, on the other hand, is invoked asynchronously; that is, the status of the hardware, MS-DOS, and the executing foreground program is indeterminate when the event that invokes the TSR occurs. Therefore, active TSRs require more complex code. The RAM-resident portion of an active TSR must contain modules that monitor the operating system to determine when control can safely be transferred to the application portion of the TSR. The monitor routines typically test the status of keyboard input, ROM BIOS interrupt processing, hardware interrupt processing, and MS-DOS function processing. The TSR activates the application (the part of the RAM-resident portion that performs the TSR's main task) only when it detects the appropriate keyboard input and determines that the application will not interfere with interrupt and MS-DOS function processing.

Keyboard input

An active TSR usually contains a RAM-resident module that examines keyboard input for a predetermined keystroke sequence called a "hot-key" sequence. A user executes the RAM-resident application by entering this hot-key sequence at the keyboard.

The technique used in the TSR to monitor keyboard input depends on the keyboard hardware implementation. On computers in the IBM PC and PS/2 families, the keyboard coprocessor generates an Interrupt 09H for each keypress. Therefore, a TSR can monitor user keystrokes by installing an interrupt handler (interrupt service routine, or ISR) for Interrupt 09H. This handler can thus detect a specified hot-key sequence.

ROM BIOS interrupt processing

The ROM BIOS routines in IBM PCs and PS/2s are not reentrant. An active TSR that calls the ROM BIOS must ensure that its code does not attempt to execute a ROM BIOS function that was already being executed by the foreground process when the TSR program took control of the system.

The IBM ROM BIOS routines are invoked through software interrupts, so an active TSR can monitor the status of the ROM BIOS by replacing the default interrupt handlers with custom interrupt handlers that intercept the appropriate BIOS interrupts. Each of these interrupt handlers can maintain a status flag, which it increments before transferring control to the corresponding ROM BIOS routine and decrements when the ROM BIOS routine has finished executing. Thus, the TSR monitor routines can test these flags to determine when non-reentrant BIOS routines are executing.

Hardware interrupt processing

The monitor routines of an active TSR, which may themselves be executed as the result of a hardware interrupt, should not activate the application portion of the TSR if any other hardware interrupt is being processed. On IBM PCs, for example, hardware interrupts are processed in a prioritized sequence determined by an Intel 8259A Programmable Interrupt Controller. The 8259A does not allow a hardware interrupt to execute if a previous interrupt with the same or higher priority is being serviced. All hardware interrupt handlers include code that signals the 8259A when interrupt processing is completed. (The programming interface to the 8259A is described in IBM's *Technical Reference* manuals and in Intel's technical literature.)

If a TSR were to interrupt the execution of another hardware interrupt handler before the handler signaled the 8259A that it had completed its interrupt servicing, subsequent hardware interrupts could be inhibited indefinitely. Inhibition of high-priority hardware interrupts such as the timer tick (Interrupt 08H) or keyboard interrupt (Interrupt 09H) could cause a system crash. For this reason, an active TSR must monitor the status of all hardware interrupt processing by interrogating the 8259A to ensure that control is transferred to the RAM-resident application only when no other hardware interrupts are being serviced.

MS-DOS function processing

Unlike the IBM ROM BIOS routines, MS-DOS is reentrant to a limited extent. That is, there are certain times when MS-DOS's servicing of an Interrupt 21H function call invoked by a foreground process can be suspended so that the RAM-resident application can make an Interrupt 21H function call of its own. For this reason, an active TSR must monitor operating system activity to determine when it is safe for the TSR application to make its calls to MS-DOS.

MS-DOS Support for Terminate-and-Stay-Resident Programs

Several MS-DOS system calls are useful for supporting terminate-and-stay-resident utilities. These are listed in Table 11-1. See SYSTEM CALLS.

Table 11-1. MS-DOS Functions Useful in TSR Programs.

Function Name	Call With	Returns	Comment
Terminate and Stay Resident	AH = 31H AL = return code DX = size of resident program (in 16-byte paragraphs) INT 21H	Nothing	Preferred over Interrupt 27H with MS-DOS versions 2.x and later
Terminate and Stay Resident	CS = PSP DX = size of resident program (bytes) INT 27H	Nothing	Provided for compatibility with MS-DOS versions 1.x
Set Interrupt Vector	AH = 25H AL = interrupt number DS:DX = address of interrupt handler INT 21H	Nothing	
Get Interrupt Vector	AH = 35H AL = interrupt number INT 21H	ES:BX = address of interrupt handler	
Set PSP Address	AH = 50H BX = PSP segment INT 21H	Nothing	
Get PSP Address	AH = 51H INT 21H	BX = PSP segment	
Set Extended Error Information	AX = 5D0AH DS:DX = address of 11-word data structure: word 0: register AX as returned by Function 59H word 1: register BX word 2: register CX word 3: register DX word 4: register SI word 5: register DI word 6: register DS word 7: register ES words 8-0AH: reserved; should be 0 INT 21H	Nothing	MS-DOS versions 3.1 and later

(more)

Table 11-1. *Continued.*

Function Name	Call With	Returns	Comment
Get Extended Error Information	AH = 59H BX = 0 INT 21H	AX = extended error code BH = error class BL = suggested action CH = error locus	
Set Disk Transfer Area Address	AH = 1AH DS:DX = address of DTA INT 21H	Nothing	
Get Disk Transfer Area Address	AH = 2FH INT 21H	ES:BX = address of current DTA	
Get InDOS Flag Address	AH = 34H INT 21H	ES:BX = address of InDOS flag	

Terminate-and-stay-resident functions

MS-DOS provides two mechanisms for terminating the execution of a program while leaving a portion of it resident in RAM. The preferred method is to execute Interrupt 21H Function 31H.

Interrupt 21H Function 31H

When this Interrupt 21H function is called, the value in DX specifies the amount of RAM (in paragraphs) that is to remain allocated after the program terminates, starting at the program segment prefix (PSP). The function is similar to Function 4CH (Terminate Process with Return Code) in that it passes a return code in AL, but it differs in that open files are not automatically closed by Function 31H.

Interrupt 27H

When Interrupt 27H is executed, the value passed in DX specifies the number of bytes of memory required for the RAM-resident program. MS-DOS converts the value passed in DX from bytes to paragraphs, sets AL to zero, and jumps to the same code that would be executed for Interrupt 21H Function 31H. Interrupt 27H is less flexible than Interrupt 21H Function 31H because it limits the size of the program that can remain resident in RAM to 64 KB, it requires that CS point to the base of the PSP, and it does not pass a return code. Later versions of MS-DOS support Interrupt 27H primarily for compatibility with versions 1.x.

TSR RAM management

In addition to the RAM explicitly allocated to the TSR by means of the value in DX, the RAM allocated to the TSR's environment remains resident when the installation portion of the TSR program terminates. (The paragraph address of the environment is found at

offset 2CH in the TSR's PSP.) Moreover, if the installation portion of a TSR program has used Interrupt 21H Function 48H (Allocate Memory Block) to allocate additional RAM, this memory also remains allocated when the program terminates. If the RAM-resident program does not need this additional RAM, the installation portion of the TSR program should free it explicitly by using Interrupt 21H Function 49H (Free Memory Block) before executing Interrupt 21H Function 31H.

Set and Get Interrupt Vector functions

Two Interrupt 21H function calls are available to inspect or update the contents of a specified 8086-family interrupt vector. Function 25H (Set Interrupt Vector) updates the vector of the interrupt number specified in the AL register with the segment and offset values specified in DS:DX. Function 35H (Get Interrupt Vector) performs the inverse operation: It copies the current vector of the interrupt number specified in AL into the ES:BX register pair.

Although it is possible to manipulate interrupt vectors directly, the use of Interrupt 21H Functions 25H and 35H is generally more convenient and allows for upward compatibility with future versions of MS-DOS.

Set and Get PSP Address functions

MS-DOS uses a program's PSP to keep track of certain data unique to the program, including command-line parameters and the segment address of the program's environment. *See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.* To access this information, MS-DOS maintains an internal variable that always contains the location of the PSP associated with the foreground process. When a RAM-resident application is activated, it should use Interrupt 21H Functions 50H (Set Program Segment Prefix Address) and 51H (Get Program Segment Prefix Address) to preserve the current contents of this variable and to update the variable with the location of its own PSP. Function 50H (Set Program Segment Prefix Address) updates an internal MS-DOS variable that locates the PSP currently in use by the foreground process. Function 51H (Get Program Segment Prefix Address) returns the contents of the internal MS-DOS variable to the caller.

Set and Get Extended Error Information functions

In MS-DOS versions 3.1 and later, the RAM-resident program should preserve the foreground process's extended error information so that, if the RAM-resident application encounters an MS-DOS error, the extended error information pertaining to the foreground process will still be available and can be restored. Interrupt 21H Functions 59H and 5D0AH provide a mechanism for the RAM-resident program to save and restore this information during execution of a TSR application.

Function 59H (Get Extended Error Information), which became available in version 3.0, returns detailed information on the most recently detected MS-DOS error. The inverse operation is performed by Function 5D0AH (Set Extended Error Information), which can be used only in MS-DOS versions 3.1 and later. This function copies extended error information to MS-DOS from a data structure defined in the calling program.

Set and Get Disk Transfer Area Address functions

Several MS-DOS data transfer functions, notably Interrupt 21H Functions 21H, 22H, 27H, and 28H (the Random Read and Write functions) and Interrupt 21H Functions 14H and 15H (the Sequential Read and Write functions), require a program to specify a disk transfer area (DTA). By default, a program's DTA is located at offset 80H in its program segment prefix. If a RAM-resident application calls an MS-DOS function that uses a DTA, the TSR should save the DTA address belonging to the interrupted program by using Interrupt 21H Function 2FH (Get Disk Transfer Area Address), supply its own DTA address to MS-DOS using Interrupt 21H Function 1AH (Set Disk Transfer Area Address), and then, before terminating, restore the interrupted program's DTA.

The MS-DOS idle interrupt (Interrupt 28H)

Several of the first 12 MS-DOS functions (01H through 0CH) must wait for the occurrence of an expected event such as a user keypress. These functions contain an "idle loop" in which looping continues until the event occurs. To provide a mechanism for other system activity to take place while the idle loop is executing, these MS-DOS functions execute an Interrupt 28H from within the loop.

The default MS-DOS handler for Interrupt 28H is only an IRET instruction. By supplying its own handler for Interrupt 28H, a TSR can perform some useful action at times when MS-DOS is otherwise idle. Specifically, a custom Interrupt 28H handler can be used to examine the current status of the system to determine whether or not it is safe to activate the RAM-resident application.

Determining MS-DOS Status

A TSR can infer the current status of MS-DOS from knowledge of its internal use of stacks and from a pair of internal status flags. This status information is essential to the proper execution of an active TSR because a RAM-resident application can make calls to MS-DOS only when those calls will not disrupt an earlier call made by the foreground process.

MS-DOS internal stacks

MS-DOS versions 2.0 and later may use any of three internal stacks: the I/O stack (*IOStack*), the disk stack (*DiskStack*), and the auxiliary stack (*AuxStack*). In general, *IOStack* is used for Interrupt 21H Functions 01H through 0CH and *DiskStack* is used for the remaining Interrupt 21H functions; *AuxStack* is normally used only when MS-DOS has detected a critical error and subsequently executed an Interrupt 24H. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers. Specifically, MS-DOS's internal stack use depends on which MS-DOS function is being executed and on the value of the critical error flag.

The critical error flag

The critical error flag (*ErrorMode*) is a 1-byte flag that MS-DOS uses to indicate whether or not a critical error has occurred. During normal, errorless execution, the value of the

critical error flag is zero. Whenever MS-DOS detects a critical error, it sets this flag to a nonzero value before it executes Interrupt 24H. If an Interrupt 24H handler subsequently invokes an MS-DOS function by using Interrupt 21H, the nonzero value of the critical error flag tells MS-DOS to use its auxiliary stack for Interrupt 21H Functions 01H through 0CH instead of using the I/O stack as it normally would.

In other words, when control is transferred to MS-DOS through Interrupt 21H, the function number and the critical error flag together determine MS-DOS stack use for the function. Figure 11-2 outlines the internal logic used on entry to an MS-DOS function to select which stack is to be used during processing of the function. As stated above, for Functions 01H through 0CH, MS-DOS uses *IOStack* if the critical error flag is zero and *AuxStack* if the flag is nonzero. For function numbers greater than 0CH, MS-DOS usually uses *DiskStack*, but Functions 50H, 51H, and 59H are important exceptions. Functions 50H and 51H use either *IOStack* (in versions 2.x) or the stack supplied by the calling program (in versions 3.x). In version 3.0, Function 59H uses either *IOStack* or *AuxStack*, depending on the value of the critical error flag, but in versions 3.1 and later, Function 59H always uses *AuxStack*.

MS-DOS versions 2.x

```

if (FunctionNumber >= 01H and FunctionNumber <= 0CH)
  or
  FunctionNumber = 50H
  or
  FunctionNumber = 51H

then if ErrorMode = 0
  then use IOStack
  else use AuxStack

else ErrorMode = 0
  use DiskStack

```

MS-DOS version 3.0

```

if FunctionNumber = 50H
  or
  FunctionNumber = 51H
  or
  FunctionNumber = 62H

then use caller's stack

else if (FunctionNumber >= 01H and FunctionNumber <= 0CH)
  or
  Function Number = 59H

  then if ErrorMode = 0
    then use IOStack
    else use AuxStack

  else ErrorMode = 0
    use DiskStack

```

Figure 11-2. Strategy for use of MS-DOS internal stacks.

(more)

MS-DOS versions 3.1 and later

```

if  FunctionNumber = 33H
  or
  FunctionNumber = 50H
  or
  FunctionNumber = 51H
  or
  FunctionNumber = 62H

then use caller's stack

else if      (FunctionNumber >= 01H and FunctionNumber <= 0CH)

  then if    ErrorMode = 0
    then use IOStack
    else use AuxStack

  else if FunctionNumber = 59H
    then use AuxStack
    else ErrorMode = 0
      use DiskStack

```

Figure 11-2. Continued.

This scheme makes Functions 01H through 0CH reentrant in a limited sense, in that a substitute critical error (Interrupt 24H) handler invoked while the critical error flag is nonzero can still use these Interrupt 21H functions. In this situation, because the flag is nonzero, *AuxStack* is used for Functions 01H through 0CH instead of *IOStack*. Thus, if *IOStack* is in use when the critical error is detected, its contents are preserved during the handler's subsequent calls to these functions.

The stack-selection logic differs slightly between MS-DOS versions 2 and 3. In versions 3.x, a few functions — notably 50H and 51H — avoid using any of the MS-DOS stacks. These functions perform uncomplicated tasks that make minimal demands for stack space, so the calling program's stack is assumed to be adequate for them.

The InDOS flag

InDOS is a 1-byte flag that is incremented each time an Interrupt 21H function is invoked and decremented when the function terminates. The flag's value remains nonzero as long as code within MS-DOS is being executed. The value of InDOS does not indicate which internal stack MS-DOS is using.

Whenever MS-DOS detects a critical error, it zeros InDOS before it executes Interrupt 24H. This action is taken to accommodate substitute Interrupt 24H handlers that do not return control to MS-DOS. If InDOS were not zeroed before such a handler gained control, its value would never be decremented and would therefore be incorrect during subsequent calls to MS-DOS.

The address of the 1-byte InDOS flag can be obtained from MS-DOS by using Interrupt 21H Function 34H (Return Address of InDOS Flag). In versions 3.1 and later, the 1-byte critical error flag is located in the byte preceding InDOS, so, in effect, the address of both

flags can be found using Function 34H. Unfortunately, there is no easy way to find the critical error flag in other versions. The recommended technique is to scan the MS-DOS segment, which is returned in the ES register by Function 34H, for one of the following sequences of instructions:

```
test    ss:[CriticalErrorFlag],0FFH    ;(versions 3.1 and later)
jne     NearLabel
push    ss:[NearWord]
int     28H
```

or

```
cmp     ss:[CriticalErrorFlag],00      ;(versions earlier than 3.1)
jne     NearLabel
int     28H
```

When the TEST or CMP instruction has been identified, the offset of the critical error flag can be obtained from the instruction's operand field.

The Multiplex Interrupt

The MS-DOS multiplex interrupt (Interrupt 2FH) provides a general mechanism for a program to verify the presence of a TSR and communicate with it. A program communicates with a TSR by placing an identification value in AH and a function number in AL and issuing an Interrupt 2FH. The TSR's Interrupt 2FH handler compares the value in AH to its own predetermined ID value. If they match, the TSR's handler keeps control and performs the function specified in the AL register. If they do not match, the TSR's handler relinquishes control to the previously installed Interrupt 2FH handler. (Multiplex ID values 00H through 7FH are reserved for use by MS-DOS; therefore, user multiplex numbers should be in the range 80H through 0FFH.)

The handler in the following example recognizes only one function, corresponding to AL = 00H. In this case, the handler returns the value 0FFH in AL, signifying that the handler is indeed resident in RAM. Thus, a program can detect the presence of the handler by executing Interrupt 2FH with the handler's ID value in AH and 00H in AL.

```
mov     ah,MultiplexID
mov     al,00H
int     2FH
cmp     al,0FFH
je     AlreadyInstalled
```

To ensure that the identification byte is unique, its value should be determined at the time the TSR is installed. One way to do this is to pass the value to the TSR program as a command-line parameter when the TSR program is installed. Another approach is to place the identification value in an environment variable. In this way, the value can be found in the environment of both the TSR and any other program that calls Interrupt 2FH to verify the TSR's presence.

In practice, the multiplex interrupt can also be used to pass information to and from a RAM-resident program in the CPU registers, thus providing a mechanism for a program to share control or status information with a TSR.

TSR Programming Examples

One effective way to become familiar with TSRs is to examine functional programs. Therefore, the subsequent pages present two examples: a simple passive TSR and a more complex active TSR.

HELLO.ASM

The "bare-bones" TSR in Figure 11-3 is a passive TSR. The RAM-resident application, which simply displays the message *Hello, World*, is invoked by executing a software interrupt. This example illustrates the fundamental interactions among a RAM-resident program, MS-DOS, and programs that execute after the installation of the RAM-resident utility.

```

;
; Name:          hello
;
; Description:   This RAM-resident (terminate-and-stay-resident) utility
;               displays the message "Hello, World" in response to a
;               software interrupt.
;
; Comments:     Assemble and link to create HELLO.EXE.
;
;               Execute HELLO.EXE to make resident.
;
;               Execute INT 64h to display the message.
;

TSRInt      EQU      64h
STDOUT      EQU      1

RESIDENT_TEXT SEGMENT byte public 'CODE'
              ASSUME cs:RESIDENT_TEXT,ds:RESIDENT_DATA

TSRAction   PROC     far

              sti                    ; enable interrupts

              push   ds              ; preserve registers
              push   ax
              push   bx
              push   cx
              push   dx

```

Figure 11-3. HELLO.ASM, a passive TSR.

(more)

```

        mov     dx,seg RESIDENT_DATA
        mov     ds,dx
        mov     dx,offset Message      ; DS:DX -> message
        mov     cx,16                  ; CX = length
        mov     bx,STDOUT              ; BX = file handle
        mov     ah,40h                 ; AH = INT 21H function 40H
                                         ; (Write File)
        int     21h                    ; display the message

        pop     dx                      ; restore registers and exit
        pop     cx
        pop     bx
        pop     ax
        pop     ds
        iret

TSRAction ENDP

RESIDENT_TEXT ENDS

RESIDENT_DATA SEGMENT word public 'DATA'

Message DB 0Dh,0Ah,'Hello, World',0Dh,0Ah

RESIDENT_DATA ENDS

TRANSIENT_TEXT SEGMENT para public 'TCODE'
ASSUME cs:TRANSIENT_TEXT,ss:TRANSIENT_STACK

HelloTSR PROC far                      ; At entry:   CS:IP -> SnapTSR
                                         ;           SS:SP -> stack
                                         ;           DS,ES -> PSP

; Install this TSR's interrupt handler

        mov     ax,seg RESIDENT_TEXT
        mov     ds,ax
        mov     dx,offset RESIDENT_TEXT:TSRAction
        mov     al,TSRInt
        mov     ah,25h
        int     21h

; Terminate and stay resident

        mov     dx,cs                  ; DX = paragraph address of start of
                                         ; transient portion (end of resident
                                         ; portion)
        mov     ax,es                  ; ES = PSP segment
        sub     dx,ax                  ; DX = size of resident portion

```

Figure 11-3. Continued.

(more)

```

                mov     ax,3100h        ; AH = INT 21H function number (TSR)
                ; AL = 00H (return code)
                int     21h

HelloTSR       ENDP

TRANSIENT_TEXT ENDS

TRANSIENT_STACK SEGMENT word stack 'TSTACK'

                DB     80h dup(?)

TRANSIENT_STACK ENDS

                END     HelloTSR

```

Figure 11-3. Continued.

The transient portion of the program (in the segments *TRANSIENT_TEXT* and *TRANSIENT_STACK*) runs only when the file HELLO.EXE is executed. This installation code updates an interrupt vector to point to the resident application (the procedure *TSRAction*) and then calls Interrupt 21H Function 31H to terminate execution, leaving the segments *RESIDENT_TEXT* and *RESIDENT_DATA* in RAM.

The order in which the code and data segments appear in the listing is important. It ensures that when the program is executed as a .EXE file, the resident code and data are placed in memory at lower addresses than the transient code and data. Thus, when Interrupt 21H Function 31H is called, the memory occupied by the transient portion of the program is freed without disrupting the code and data in the resident portion.

The RAM containing the resident portion of the utility is left intact by MS-DOS during subsequent execution of other programs. Thus, after the TSR has been installed, any program that issues the software interrupt recognized by the TSR (in this example, Interrupt 64H) will transfer control to the routine *TSRAction*, which uses Interrupt 21H Function 40H to display a simple message on standard output.

Part of the reason this example is so short is that it performs no error checking. A truly reliable version of the program would check the version of MS-DOS in use, verify that the program was not already installed in memory, and chain to any previously installed interrupt handlers that use the same interrupt vector. (The next program, SNAP.ASM, illustrates these techniques.) However, the primary reason the program is small is that it makes the basic assumption that MS-DOS, the ROM BIOS, and the hardware interrupts are all stable at the time the resident utility is executed.

SNAP.ASM

The preceding assumption is a reliable one in the case of the passive TSR in Figure 11-3, which executes only when it is explicitly invoked by a software interrupt. However, the situation is much more complicated in the case of the active TSR in Figure 11-4. This

program is relatively long because it makes no assumptions about the stability of the operating environment. Instead, it monitors the status of MS-DOS, the ROM BIOS, and the hardware interrupts to decide when the RAM-resident application can safely execute.

```

;
; Name:          snap
;
; Description:   This RAM-resident (terminate-and-stay-resident) utility
;               produces a video "snapshot" by copying the contents of the
;               video regeneration buffer to a disk file.  It may be used
;               in 80-column alphanumeric video modes on IBM PCs and PS/2s.
;
; Comments:     Assemble and link to create SNAP.EXE.
;
;               Execute SNAP.EXE to make resident.
;
;               Press Alt-Enter to dump current contents of video buffer
;               to a disk file.
;
MultiplexID     EQU     0CAh           ; unique INT 2FH ID value
TSRStackSize    EQU     100h          ; resident stack size in bytes
KB_FLAG        EQU     17h           ; offset of shift-key status flag in
;                                     ; ROM BIOS keyboard data area
KBIns          EQU     80h           ; bit masks for KB_FLAG
KBCaps         EQU     40h
KBNum          EQU     20h
KBScroll       EQU     10h
KBAlt          EQU     8
KBctl          EQU     4
KBLeft         EQU     2
KBRight        EQU     1
SCEnter        EQU     1Ch
CR             EQU     0Dh
LF             EQU     0Ah
TRUE           EQU     -1
FALSE          EQU     0
PAGE
;-----
;
; RAM-resident routines
;
;-----
RESIDENT_GROUP GROUP RESIDENT_TEXT, RESIDENT_DATA, RESIDENT_STACK

```

Figure 11-4. SNAP.ASM, a video snapshot TSR.

(more)


```

RESIDENT_TEXT SEGMENT byte public 'CODE'
               ASSUME cs:RESIDENT_GROUP,ds:RESIDENT_GROUP

;-----
; System verification routines
;-----

VerifyDOSState PROC     near           ; Returns:   carry flag set if MS-DOS
                               ;               is busy
                               ; preserve these registers
               push     ds
               push     bx
               push     ax

               lds     bx,cs:ErrorModeAddr
               mov     ah,[bx]         ; AH = ErrorMode flag

               lds     bx,cs:InDOSAddr
               mov     al,[bx]        ; AL = InDOS flag

               xor     bx,bx          ; BH = 00H, BL = 00H
               cmp     bl,cs:InISR28 ; carry flag set if INT 28H handler
                               ; is running
               rcl     bl,01h         ; BL = 01H if INT 28H handler is running

               cmp     bx,ax          ; carry flag zero if AH = 00H
                               ; and AL <= BL
               pop     ax             ; restore registers
               pop     bx
               pop     ds
               ret

VerifyDOSState ENDP

VerifyIntState PROC     near           ; Returns:   carry flag set if hardware
                               ;               or ROM BIOS unstable

               push     ax            ; preserve AX

; Verify hardware interrupt status by interrogating Intel 8259A Programmable
; Interrupt Controller

               mov     ax,00001011b ; AH = 0
                               ; AL = 0CW3 for Intel 8259A (RR = 1,
                               ; RIS = 1)
               out     20h,al        ; request 8259A's in-service register
               jmp     short L10      ; wait a few cycles
L10:           in      al,20h         ; AL = hardware interrupts currently
                               ; being serviced (bit = 1 if in-service)

```

Figure 11-4. Continued.

(more)

```

        cmp     ah,al
        jc     L11           ; exit if any hardware interrupts still
                           ; being serviced

; Verify status of ROM BIOS interrupt handlers

        xor     al,al       ; AL = 00H

        cmp     al,cs:InISR5
        jc     L11         ; exit if currently in INT 05H handler

        cmp     al,cs:InISR9
        jc     L11         ; exit if currently in INT 09H handler

        cmp     al,cs:InISR10
        jc     L11         ; exit if currently in INT 10H handler

        cmp     al,cs:InISR13 ; set carry flag if currently in
                           ; INT 13H handler
L11:    pop     ax         ; restore AX and return
        ret

VerifyIntState ENDP

VerifyTSRState PROC near ; Returns: carry flag set if TSR
                           ; inactive
        rol     cs:HotFlag,1 ; carry flag set if (HotFlag = TRUE)
        cmc     ; carry flag set if (HotFlag = FALSE)
        jc     L20         ; exit if no hot key

        ror     cs:ActiveTSR,1 ; carry flag set if (ActiveTSR = TRUE)
        jc     L20         ; exit if already active

        call    VerifyDOSState
        jc     L20         ; exit if MS-DOS unstable

        call    VerifyIntState ; set carry flag if hardware or BIOS
                           ; unstable
L20:    ret

VerifyTSRState ENDP

PAGE
-----
; System monitor routines
-----

ISR5    PROC far           ; INT 05H handler
                           ; (ROM BIOS print screen)
        inc     cs:InISR5 ; increment status flag

```

Figure 11-4. Continued.

(more)

```

        pushf
        cli
        call    cs:PrevISR5    ; chain to previous INT 05H handler

        dec    cs:InISR5      ; decrement status flag
        iret

ISR5:   ENDP

ISR8:   PROC    far           ; INT 08H handler (timer tick, IRQ0)

        pushf
        cli
        call    cs:PrevISR8    ; chain to previous handler

        cmp    cs:InISR8,0
        jne    L31             ; exit if already in this handler

        inc    cs:InISR8      ; increment status flag

        sti                    ; interrupts are ok
        call   VerifyTSRState
        jc     L30             ; jump if TSR is inactive

        mov    byte ptr cs:ActiveTSR,TRUE
        call   TSRapp
        mov    byte ptr cs:ActiveTSR,FALSE

L30:    dec    cs:InISR8

L31:    iret

ISR8:   ENDP

ISR9:   PROC    far           ; INT 09H handler
        ; (keyboard interrupt IRQ1)

        push   ds              ; preserve these registers
        push   ax
        push   bx

        push   cs
        pop    ds              ; DS -> RESIDENT_GROUP

        in    al,60h          ; AL = current scan code

        pushf                    ; simulate an INT
        cli
        call   ds:PrevISR9     ; let previous handler execute

```

Figure 11-4. Continued.

(more)

```
        mov     ah,ds:InISR9    ; if already in this handler ..
        or      ah,ds:HotFlag   ; .. or currently processing hot key ..
        jnz    L43              ; .. jump to exit

        inc    ds:InISR9       ; increment status flag
        sti                                ; now interrupts are ok

; Check scan code sequence

        cmp    ds:HotSeqLen,0
        je     L40              ; jump if no hot sequence to match

        mov    bx,ds:HotIndex
        cmp    al,[bx+HotSequence] ; test scan code sequence
        jne    L41              ; jump if no match

        inc    bx
        cmp    bx,ds:HotSeqLen
        jb     L42              ; jump if not last scan code to match

; Check shift-key state

L40:     push   ds
        mov    ax,40h
        mov    ds,ax           ; DS -> ROM BIOS data area
        mov    al,ds:[KB_FLAG] ; AH = ROM BIOS shift-key flags
        pop    ds

        and    al,ds:HotKBMask ; AL = flags AND "don't care" mask
        cmp    al,ds:HotKBFlag
        jne    L42              ; jump if shift state does not match

; Set flag when hot key is found

        mov    byte ptr ds:HotFlag,TRUE

L41:     xor    bx,bx           ; reinitialize index

L42:     mov    ds:HotIndex,bx ; update index into sequence
        dec    ds:InISR9       ; decrement status flag

L43:     pop    bx              ; restore registers and exit
        pop    ax
        pop    ds
        iret

ISR9     ENDP
```

Figure 11-4. Continued.

(more)

```

ISR10      PROC   far           ; INT 10H handler (ROM BIOS video I/O)

           inc     cs:InISR10   ; increment status flag

           pushf
           cli
           call    cs:PrevISR10 ; chain to previous INT 10H handler

           dec     cs:InISR10   ; decrement status flag
           ired

ISR10      ENDP

ISR13      PROC   far           ; INT 13H handler
           ; (ROM BIOS fixed disk I/O)
           inc     cs:InISR13   ; increment status flag

           pushf
           cli
           call    cs:PrevISR13 ; chain to previous INT 13H handler

           pushf                ; preserve returned flags
           dec     cs:InISR13   ; decrement status flag
           popf                 ; restore flags register

           sti                 ; enable interrupts
           ret     2            ; simulate IRET without popping flags

ISR13      ENDP

ISR1B      PROC   far           ; INT 1BH trap (ROM BIOS Ctrl-Break)

           mov     byte ptr cs:Trap1B,TRUE

           ired

ISR1B      ENDP

ISR23      PROC   far           ; INT 23H trap (MS-DOS Ctrl-C)

           mov     byte ptr cs:Trap23,TRUE

           ired

ISR23      ENDP

ISR24      PROC   far           ; INT 24H trap (MS-DOS critical error)

           mov     byte ptr cs:Trap24,TRUE

```

Figure 11-4. Continued.

(more)

```

xor    al,al          ; AL = 00H (MS-DOS 2.x):
cmp    cs:MajorVersion,2 ; ignore the error
je     L50

mov    al,3           ; AL = 03H (MS-DOS 3.x):
                        ; fail the MS-DOS call in which
                        ; the critical error occurred

L50:    iret

ISR24   ENDP

ISR28   PROC    far          ; INT 28H handler
                        ; (MS-DOS idle interrupt)
pushf
cli
call   cs:PrevISR28      ; chain to previous INT 28H handler

cmp    cs:InISR28,0
jne    L61              ; exit if already inside this handler

inc    cs:InISR28       ; increment status flag

call   VerifyTSRState
jc     L60              ; jump if TSR is inactive

mov    byte ptr cs:ActiveTSR,TRUE
call   TSRapp
mov    byte ptr cs:ActiveTSR,FALSE

L60:    dec    cs:InISR28   ; decrement status flag

L61:    iret

ISR28   ENDP

ISR2F   PROC    far          ; INT 2FH handler
                        ; (MS-DOS multiplex interrupt)
                        ; Caller: AH = handler ID
                        ; AL = function number
                        ; Returns for function 0: AL = 0FFH
                        ; for all other functions: nothing

cmp    ah,MultiplexID
je     L70              ; jump if this handler is requested

jmp    cs:PrevISR2F     ; chain to previous INT 2FH handler

```

Figure 11-4. Continued.

(more)

```

L70:      test   al,al
          jnz   MultiplexIRET ; jump if reserved or undefined function

; Function 0: get installed state

          mov   al,0FFh      ; AL = 0FFH (this handler is installed)

MultiplexIRET:  iret        ; return from interrupt

ISR2F      ENDP

          PAGE

;
;
; AuxInt21--sets ErrorMode while executing INT 21H to force use of the
; AuxStack instead of the IOSTack.
;
;
AuxInt21    PROC    near          ; Caller:   registers for INT 21H
                                     ; Returns:  registers from INT 21H

          push  ds
          push  bx
          lds  bx,ErrorModeAddr
          inc  byte ptr [bx] ; ErrorMode is now nonzero
          pop  bx
          pop  ds

          int  21h              ; perform MS-DOS function

          push  ds
          push  bx
          lds  bx,ErrorModeAddr
          dec  byte ptr [bx] ; restore ErrorMode
          pop  bx
          pop  ds
          ret

AuxInt21    ENDP

Int21v     PROC    near          ; perform INT 21H or AuxInt21,
                                     ; depending on MS-DOS version

          cmp  DOSVersion,30Ah
          jb  L80                ; jump if earlier than 3.1

          int  21h              ; versions 3.1 and later
          ret

```

Figure 11-4. Continued.

(more)

```

L80:      call   AuxInt21      ; versions earlier than 3.1
         ret

Int21v    ENDP

         PAGE
-----
; RAM-resident application
-----

TSRapp    PROC    near

; Set up a safe stack

         push   ds            ; save previous DS on previous stack

         push   cs
         pop    ds            ; DS -> RESIDENT_GROUP

         mov    PrevSP,sp     ; save previous SS:SP
         mov    PrevSS,ss

         mov    ss,TSRSS     ; SS:SP -> RESIDENT_STACK
         mov    sp,TSRSP

         push   es            ; preserve remaining registers
         push   ax
         push   bx
         push   cx
         push   dx
         push   si
         push   di
         push   bp

         cld                  ; clear direction flag

; Set break and critical error traps

         mov    cx,NTrap
         mov    si,offset RESIDENT_GROUP:StartTrapList

L90:      lodsb                ; AL = interrupt number
         ; DS:SI -> byte past interrupt number

         mov    byte ptr [si],FALSE ; zero the trap flag

         push   ax            ; preserve AX
         mov    ah,35h        ; INT 21H function 35H
         ; (get interrupt vector)
         int    21h          ; ES:BX = previous interrupt vector
         mov    [si+1],bx     ; save offset and segment ..
         mov    [si+3],es     ; .. of previous handler

```

Figure 11-4. Continued.

(more)


```

        pop     ax             ; AL = interrupt number
        mov     dx,[si+5]     ; DS:DX -> this TSR's trap
        mov     ah,25h       ; INT 21H function 25H
        int     21h          ; (set interrupt vector)
        add     si,7          ; DS:SI -> next in list

        loop   L90

; Disable MS-DOS break checking during disk I/O

        mov     ax,3300h     ; AH = INT 21H function number
                           ; AL = 00H (request current break state)
        int     21h          ; DL = current break state
        mov     PrevBreak,dl ; preserve current state

        xor     dl,dl        ; DL = 00H (disable disk I/O break
                           ; checking)
        mov     ax,3301h     ; AL = 01H (set break state)
        int     21h

; Preserve previous extended error information

        cmp     DOSVersion,30Ah
        jb     L91           ; jump if MS-DOS version earlier
                           ; than 3.1
        push    ds           ; preserve DS
        xor     bx,bx        ; BX = 00H (required for function 59H)
        mov     ah,59h       ; INT 21H function 59H
        call   Int21v        ; (get extended error info)

        mov     cs:PrevExtErrDS,ds
        pop     ds
        mov     PrevExtErrAX,ax ; preserve error information
        mov     PrevExtErrBX,bx ; in data structure
        mov     PrevExtErrCX,cx
        mov     PrevExtErrDX,dx
        mov     PrevExtErrSI,si
        mov     PrevExtErrDI,di
        mov     PrevExtErrES,es

; Inform MS-DOS about current PSP

L91:     mov     ah,51h       ; INT 21H function 51H (get PSP address)
        call   Int21v        ; BX = foreground PSP

        mov     PrevPSP,bx   ; preserve previous PSP

        mov     bx,TSRPSP    ; BX = resident PSP
        mov     ah,50h       ; INT 21H function 50H (set PSP address)
        call   Int21v

```

Figure 11-4. Continued.

(more)

```

; Inform MS-DOS about current DTA (not really necessary in this application
; because DTA is not used)

mov     ah,2Fh           ; INT 21H function 2FH
int     21h             ; (get DTA address) into ES:BX
mov     PrevDTAoffs,bx
mov     PrevDTAseg,es

push    ds              ; preserve DS
mov     ds,TSRPSP
mov     dx,80h          ; DS:DX -> default DTA at PSP:0080H
mov     ah,1Ah          ; INT 21H function 1AH
int     21h            ; (set DTA address)
pop     ds              ; restore DS

; Open a file, write to it, and close it

mov     ax,0E07h        ; AH = INT 10H function number
                        ; (write teletype)
                        ; AL = 07H (bell character)
int     10h            ; emit a beep

mov     dx,offset RESIDENT_GROUP:SnapFile
mov     ah,3Ch          ; INT 21H function 3CH
                        ; (create file handle)
mov     cx,0            ; file attribute
int     21h
jc     L94              ; jump if file not opened

push    ax              ; push file handle
mov     ah,0Fh          ; INT 10H function 0FH (get video status)
int     10h            ; AL = video mode number
                        ; AH = number of character columns
pop     bx              ; BX = file handle

cmp     ah,80
jne     L93             ; jump if not 80-column mode

mov     dx,0B800h       ; DX = color video buffer segment
cmp     al,3
jbe     L92             ; jump if color alphanumeric mode

cmp     al,7
jne     L93             ; jump if not monochrome mode

mov     dx,0B000h       ; DX = monochrome video buffer segment

L92:    push    ds
        mov     ds,dx
        xor     dx,dx    ; DS:DX -> start of video buffer
        mov     cx,80*25*2 ; CX = number of bytes to write
        mov     ah,40h   ; INT 21H function 40H (write file)

```

Figure 11-4. Continued.

(more)

```

        int     21h
        pop     ds

L93:    mov     ah,3Eh      ; INT 21H function 3EH (close file)
        int     21h

        mov     ax,0E07h   ; emit another beep
        int     10h

; Restore previous DTA

L94:    push    ds         ; preserve DS
        lds    dx,PrevDTA ; DS:DX -> previous DTA
        mov     ah,1Ah     ; INT 21H function 1AH (set DTA address)
        int     21h
        pop     ds

; Restore previous PSP

        mov     bx,PrevPSP ; BX = previous PSP
        mov     ah,50h     ; INT 21H function 50H
        call    Int21v     ; (set PSP address)

; Restore previous extended error information

        mov     ax,DOSVersion
        cmp     ax,30Ah
        jb     L95         ; jump if MS-DOS version earlier than 3.1
        cmp     ax,0A00h
        jae    L95         ; jump if MS OS/2-DOS 3.x box

        mov     dx,offset RESIDENT_GROUP:PrevExtErrInfo
        mov     ax,5D0Ah
        int     21h       ; (restore extended error information)

; Restore previous MS-DOS break checking

L95:    mov     dl,PrevBreak ; DL = previous state
        mov     ax,3301h
        int     21h

; Restore previous break and critical error traps

        mov     cx,NTrap
        mov     si,offset RESIDENT_GROUP:StartTrapList
        push    ds         ; preserve DS

L96:    lods    byte ptr cs:[si] ; AL = interrupt number
        ; ES:SI -> byte past interrupt number

        lds    dx,cs:[si+1] ; DS:DX -> previous handler
        mov     ah,25h     ; INT 21H function 25H
        int     21h       ; (set interrupt vector)

```

Figure 11-4. Continued.

(more)

```

        add     si,7           ; DS:SI -> next in list
        loop   L96
        pop    ds             ; restore DS

; Restore all registers

        pop    bp
        pop    di
        pop    si
        pop    dx
        pop    cx
        pop    bx
        pop    ax
        pop    es

        mov    ss,PrevSS     ; SS:SP -> previous stack
        mov    sp,PrevSP
        pop    ds             ; restore previous DS

; Finally, reset status flag and return

        mov    byte ptr cs:HotFlag,FALSE
        ret

TSRapp      ENDP

RESIDENT_TEXT ENDS

RESIDENT_DATA SEGMENT word public 'DATA'

ErrorModeAddr DD ?           ; address of MS-DOS ErrorMode flag
InDOSAddr     DD ?           ; address of MS-DOS InDOS flag

NISR         DW (EndISRList-StartISRList)/8 ; number of installed ISRs

StartISRList DB 05h          ; INT number
InISR5       DB FALSE        ; flag
PrevISR5     DD ?            ; address of previous handler
             DW offset RESIDENT_GROUP:ISR5

             DB 08h
InISR8       DB FALSE
PrevISR8     DD ?
             DW offset RESIDENT_GROUP:ISR8

             DB 09h
InISR9       DB FALSE
PrevISR9     DD ?
             DW offset RESIDENT_GROUP:ISR9

             DB 10h
InISR10      DB FALSE

```

Figure 11-4. Continued.

(more)

```

PrevISR10      DD      ?
               DW      offset RESIDENT_GROUP:ISR10

               DB      13h
InISR13        DB      FALSE
PrevISR13      DD      ?
               DW      offset RESIDENT_GROUP:ISR13

               DB      28h
InISR28        DB      FALSE
PrevISR28      DD      ?
               DW      offset RESIDENT_GROUP:ISR28

               DB      2Fh
InISR2F        DB      FALSE
PrevISR2F      DD      ?
               DW      offset RESIDENT_GROUP:ISR2F

EndISRList     LABEL   BYTE

TSRPSP         DW      ?           ; resident PSP
TSRSP          DW      TSRStackSize ; resident SS:SP
TSRSS          DW      seg RESIDENT_STACK
PrevPSP        DW      ?           ; previous PSP
PrevSP         DW      ?           ; previous SS:SP
PrevSS         DW      ?

HotIndex       DW      0           ; index of next scan code in sequence
HotSeqLen      DW      EndHotSeq-HotSequence ; length of hot-key sequence

HotSequence    DB      SCEnter     ; hot sequence of scan codes
EndHotSeq      LABEL   BYTE

HotKBFlag      DB      KBAlt       ; hot value of ROM BIOS KB_FLAG
HotKBMask      DB      (KBIns OR KBCaps OR KNum OR KScroll) XOR 0FFh
HotFlag        DB      FALSE

ActiveTSR      DB      FALSE

DOSVersion     LABEL   WORD
               DB      ?           ; minor version number
MajorVersion   DB      ?           ; major version number

; The following data is used by the TSR application:

NTrap          DW      (EndTrapList-StartTrapList)/8 ; number of traps

StartTrapList  DB      1Bh
Trap1B         DB      FALSE
PrevISR1B      DD      ?
               DW      offset RESIDENT_GROUP:ISR1B

               DB      23h

```

Figure 11-4. Continued.

(more)

```

Trap23      DB      FALSE
PrevISR23   DD      ?
            DW      offset RESIDENT_GROUP:ISR23

            DB      24h
Trap24      DB      FALSE
PrevISR24   DD      ?
            DW      offset RESIDENT_GROUP:ISR24

EndTrapList LABEL  BYTE

PrevBreak   DB      ?           ; previous break-checking flag

PrevDTA     LABEL  DWORD       ; previous DTA address
PrevDTAoffs DW      ?
PrevDTAseg  DW      ?

PrevExtErrInfo LABEL  BYTE     ; previous extended error information
PrevExtErrAX DW      ?
PrevExtErrBX DW      ?
PrevExtErrCX DW      ?
PrevExtErrDX DW      ?
PrevExtErrSI DW      ?
PrevExtErrDI DW      ?
PrevExtErrDS DW      ?
PrevExtErrES DW      ?
            DW      3 dup(0)

SnapFile    DB      '\snap.img' ; output filename in root directory

RESIDENT_DATA ENDS

RESIDENT_STACK SEGMENT word stack 'STACK'
                DB      TSRStackSize dup(?)
RESIDENT_STACK ENDS

                PAGE
;-----
;
; Transient installation routines
;
;-----

TRANSIENT_TEXT SEGMENT para public 'TCODE'
                ASSUME  cs:TRANSIENT_TEXT,ds:RESIDENT_DATA,ss:RESIDENT_STACK

InstallSnapTSR PROC  far           ; At entry:  CS:IP -> InstallSnapTSR
                ;                SS:SP -> stack
                ;                DS,ES -> PSP

```

Figure 11-4. Continued.

(more)

```

; Save PSP segment

        mov     ax,seg RESIDENT_DATA
        mov     ds,ax           ; DS -> RESIDENT_DATA

        mov     TSRPSP,es      ; save PSP segment

; Check the MS-DOS version

        call    GetDOSVersion  ; AH = major version number
                                ; AL = minor version number

; Verify that this TSR is not already installed
;
; Before executing INT 2FH in MS-DOS versions 2.x, test whether INT 2FH
; vector is in use. If so, abort if PRINT.COM is using it.
;
; (Thus, in MS-DOS 2.x, if both this program and PRINT.COM are used,
; this program should be made resident before PRINT.COM.)

        cmp     ah,2
        ja     L101           ; jump if version 3.0 or later

        mov     ax,352Fh      ; AH = INT 21H function number
                                ; AL = interrupt number
        int     21h          ; ES:BX = INT 2FH vector

        mov     ax,es
        or     ax,bx          ; jump if current INT 2FH vector ..
        jnz    L100           ; .. is nonzero

        push   ds
        mov     ax,252Fh      ; AH = INT 21H function number
                                ; AL = interrupt number
        mov     dx,seg RESIDENT_GROUP
        mov     ds,dx
        mov     dx,offset RESIDENT_GROUP:MultiplexIRET

        int     21h          ; point INT 2FH vector to IRET
        pop    ds
        jmp    short L103     ; jump to install this TSR

L100:   mov     ax,0FF00h      ; look for PRINT.COM:
        int     2Fh          ; if resident, AH = print queue length;
                                ; otherwise, AH is unchanged

        cmp     ah,0FFh      ; if PRINT.COM is not resident ..
        je     L101           ; .. use multiplex interrupt

        mov     al,1
        call   FatalError    ; abort if PRINT.COM already installed

```

Figure 11-4. Continued.

(more)

```

L101:      mov     ah,MultiplexID ; AH = multiplex interrupt ID value
           xor     al,al         ; AL = 00H
           int     2Fh          ; multiplex interrupt

           test    al,al
           jz     L103          ; jump if ok to install

           cmp     al,0FFh
           jne    L102          ; jump if not already installed

           mov     al,2
           call   FatalError    ; already installed

L102:      mov     al,3
           call   FatalError    ; can't install

; Get addresses of INDOS and ErrorMode flags

L103:      call   GetDOSFlags

; Install this TSR's interrupt handlers

           push   es            ; preserve PSP segment

           mov     cx,NISR
           mov     si,offset StartISRList

L104:      lodsb                ; AL = interrupt number
           ; DS:SI -> byte past interrupt number
           push   ax            ; preserve AX
           mov     ah,35h       ; INT 21H function 35H
           int     21h          ; ES:BX = previous interrupt vector
           mov     [si+1],bx    ; save offset and segment ..
           mov     [si+3],es    ; .. of previous handler

           pop    ax            ; AL = interrupt number
           push   ds            ; preserve DS
           mov     dx,[si+5]
           mov     bx,seg RESIDENT_GROUP
           mov     ds,bx        ; DS:DX -> this TSR's handler
           mov     ah,25h       ; INT 21H function 25H
           int     21h          ; (set interrupt vector)
           pop    ds            ; restore DS
           add     si,7          ; DS:SI -> next in list
           loop   L104

; Free the environment

           pop    es            ; ES = PSP segment
           push   es            ; preserve PSP segment
           mov     es,es:[2Ch]  ; ES = segment of environment

```

Figure 11-4. Continued.

(more)


```

        mov     ah,49h           ; INT 21H function 49H
        int     21h             ; (free memory block)

; Terminate and stay resident

        pop     ax              ; AX = PSP segment
        mov     dx,cs           ; DX = paragraph address of start of
                                ; transient portion (end of resident
                                ; portion)
        sub     dx,ax           ; DX = size of resident portion

        mov     ax,3100h        ; AH = INT 21H function number
                                ; AL = 00H (return code)
        int     21h

InstallSnapTSR ENDP

GetDOSVersion PROC near        ; Caller: DS = seg RESIDENT_DATA
                                ; ES = PSP
                                ; Returns: AH = major version
                                ; AL = minor version
        ASSUME ds:RESIDENT_DATA

        mov     ah,30h          ; INT 21H function 30H:
                                ; (get MS-DOS version)
        int     21h
        cmp     al,2
        jnb    L110             ; jump if versions 1.x

        xchg    ah,al           ; AH = major version
                                ; AL = minor version
        mov     DOSVersion,ax    ; save with major version in
                                ; high-order byte
        ret

L110:    mov     al,00h
        call   FatalError       ; abort if versions 1.x

GetDOSVersion ENDP
GetDOSFlags PROC near          ; Caller: DS = seg RESIDENT_DATA
                                ; Returns: InDOSAddr -> InDOS
                                ; ErrorModeAddr -> ErrorMode
                                ; Destroys: AX,BX,CX,DI
        ASSUME ds:RESIDENT_DATA

; Get InDOS address from MS-DOS

        push   es

        mov     ah,34h          ; INT 21H function number
        int     21h             ; ES:BX -> InDOS

```

Figure 11-4. Continued.

(more)

```

        mov     word ptr InDOSAddr,bx
        mov     word ptr InDOSAddr+2,es

; Determine ErrorMode address

        mov     word ptr ErrorModeAddr+2,es      ; assume ErrorMode is
                                                ; in the same segment
                                                ; as INDOS

        mov     ax,DOSVersion
        cmp     ax,30Ah
        jb     L120          ; jump if MS-DOS version earlier
                        ; than 3.1 ..

        cmp     ax,0A00h
        jae     L120          ; .. or MS OS/2-DOS 3.x box

        dec     bx          ; in MS-DOS 3.1 and later, ErrorMode
        mov     word ptr ErrorModeAddr,bx      ; is just before INDOS
        jmp     short L125

L120:                                     ; scan MS-DOS segment for ErrorMode

        mov     cx,0FFFFh      ; CX = maximum number of bytes to scan
        xor     di,di          ; ES:DI -> start of MS-DOS segment

L121:        mov     ax,word ptr cs:LF2 ; AX = opcode for INT 28H

L122:        repne scasb      ; scan for first byte of fragment
        jne     L126          ; jump if not found

        cmp     ah,es:[di]      ; inspect second byte of opcode
        jne     L122          ; jump if not INT 28H

        mov     ax,word ptr cs:LF1 + 1 ; AX = opcode for CMP
        cmp     ax,es:[di][LF1-LF2]
        jne     L123          ; jump if opcode not CMP

        mov     ax,es:[di][ (LF1-LF2)+2] ; AX = offset of ErrorMode
        jmp     short L124      ; in DOS segment

L123:        mov     ax,word ptr cs:LF3 + 1 ; AX = opcode for TEST
        cmp     ax,es:[di][LF3-LF4]
        jne     L121          ; jump if opcode not TEST

        mov     ax,es:[di][ (LF3-LF4)+2] ; AX = offset of ErrorMode

L124:        mov     word ptr ErrorModeAddr,ax

L125:        pop     es
        ret

```

Figure 11-4. Continued.

(more)

```

; Come here if address of ErrorMode not found

L126:      mov     al,04h
          call    FatalError

; Code fragments for scanning for ErrorMode flag

LFnear    LABEL   near      ; dummy labels for addressing
LFbyte    LABEL   byte
LFword    LABEL   word

LF1:      cmp     ss:LFbyte,0 ; MS-DOS versions earlier than 3.1
          jne     LFnear      ; CMP ErrorMode,0
LF2:      int     28h
          ; MS-DOS versions 3.1 and later

LF3:      test    ss:LFbyte,0FFh ; TEST ErrorMode,0FFh
          jne     LFnear
          push   ss:LFword
LF4:      int     28h

GetDOSFlags ENDP

FatalError PROC   near      ; Caller:  AL = message number
          ;                               ES = PSP
          ASSUME ds:TRANSIENT_DATA

          push   ax          ; save message number on stack

          mov    bx,seg TRANSIENT_DATA
          mov    ds,bx

; Display the requested message

          mov    bx,offset MessageTable
          xor    ah,ah       ; AX = message number
          shl   ax,1        ; AX = offset into MessageTable
          add   bx,ax       ; DS:BX -> address of message
          mov   dx,[bx]     ; DS:DX -> message
          mov   ah,09h      ; INT 21H function 09H (display string)
          int   21h        ; display error message

          pop   ax         ; AL = message number
          or    al,al
          jz    L130       ; jump if message number is zero
          ; (MS-DOS versions 1.x)

; Terminate (MS-DOS 2.x and later)

          mov    ah,4Ch      ; INT 21H function 4CH
          int   21h        ; (terminate process with return code)

```

Figure 11-4. Continued.

(more)

```

; Terminate (MS-DOS 1.x)

L130          PROC    far

                push   es            ; push PSP:0000H
                xor    ax,ax
                push   ax
                ret                ; far return (jump to PSP:0000H)

L130          ENDP

FatalError    ENDP

TRANSIENT_TEXT ENDS

                PAGE

;
;
; Transient data segment
;
;

TRANSIENT_DATA SEGMENT word public 'DATA'

MessageTable  DW    Message0        ; MS-DOS version error
                DW    Message1        ; PRINT.COM found in MS-DOS 2.x
                DW    Message2        ; already installed
                DW    Message3        ; can't install
                DW    Message4        ; can't find flag

Message0      DB    CR,LF,'TSR requires MS-DOS 2.0 or later version',CR,LF,'$'
Message1      DB    CR,LF,'Can't install TSR: PRINT.COM active',CR,LF,'$'
Message2      DB    CR,LF,'This TSR is already installed',CR,LF,'$'
Message3      DB    CR,LF,'Can't install this TSR',CR,LF,'$'
Message4      DB    CR,LF,'Unable to locate MS-DOS ErrorMode flag',CR,LF,'$'

TRANSIENT_DATA ENDS

                END        InstallSnapTSR

```

Figure 11-4. Continued.

When installed, the SNAP program monitors keyboard input until the user types the hot-key sequence Alt-Enter. When the hot-key sequence is detected, the monitoring routine waits until the operating environment is stable and then activates the RAM-resident application, which dumps the current contents of the computer's video buffer into the file SNAP.IMG. Figure 11-5 is a block diagram of the RAM-resident and transient components of this TSR.

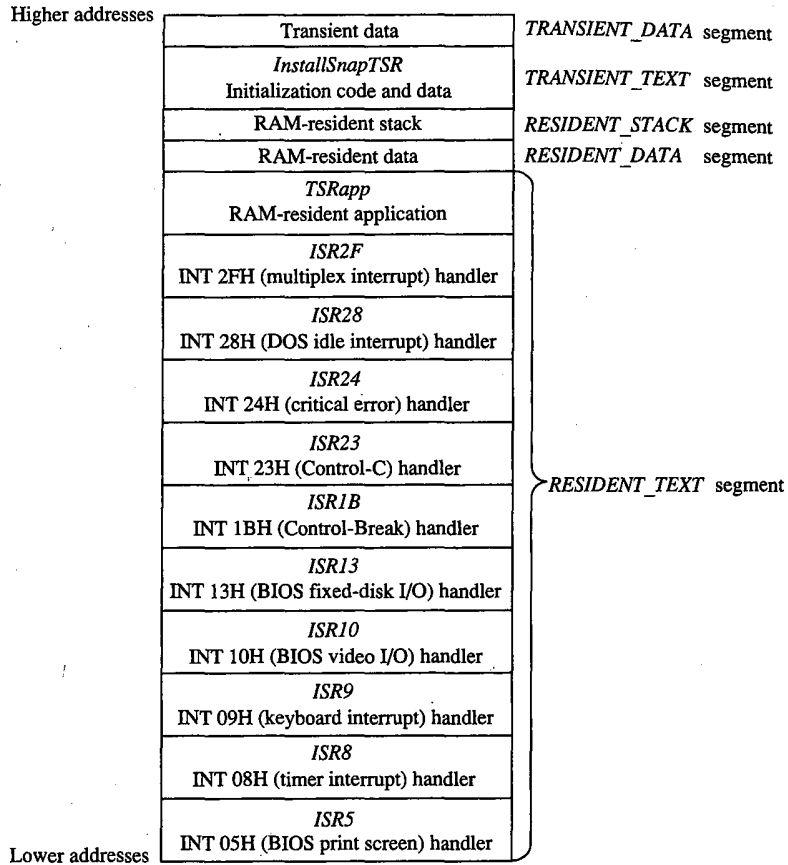


Figure 11-5. Block structure of the TSR program SNAP.EXE when loaded into memory. (Compare with Figure 11-1.)

Installing the program

When SNAP.EXE is run, only the code in the transient portion of the program is executed. The transient code performs several operations before it finally executes Interrupt 21H Function 31H (Terminate and Stay Resident). First it determines which MS-DOS version is in use. Then it executes the multiplex interrupt (Interrupt 2FH) to discover whether the resident portion has already been installed. If an MS-DOS version earlier than 2.0 is in use or if the resident portion has already been installed, the program aborts with an error message.

Otherwise, installation continues. The addresses of the InDOS and critical error flags are saved in the resident data segment. The interrupt service routines in the RAM-resident portion of the program are installed by updating all relevant interrupt vectors. The transient code then frees the RAM occupied by the program's environment, because the resident

portion of this program never uses the information contained there. Finally, the transient portion of the program, which includes the *TRANSIENT_TEXT* and *TRANSIENT_DATA* segments, is discarded and the program is terminated using Interrupt 21H Function 31H.

Detecting a hot key

The SNAP program detects the hot-key sequence (Alt-Enter) by monitoring each keypress. On IBM PCs and PS/2s, each keystroke generates a hardware interrupt on IRQ1 (Interrupt 09H). The TSR's Interrupt 09H handler compares the keyboard scan code corresponding to each keypress with a predefined sequence. The TSR's handler also inspects the shift-key status flags maintained by the ROM BIOS Interrupt 09H handler. When the predetermined sequence of keypresses is detected at the same time as the proper shift keys are pressed, the handler sets a global status flag (*HotFlag*).

Note how the TSR's handler transfers control to the previous Interrupt 09H ISR before it performs its own work. If the TSR's Interrupt 09H handler did not chain to the previous handler(s), essential system processing of keystrokes (particularly in the ROM BIOS Interrupt 09H handler) might not be performed.

Activating the application

The TSR monitors the status of *HotFlag* by regularly testing its value within a timer-tick handler. On IBM PCs and PS/2s, the timer-tick interrupt occurs on IRQ0 (Interrupt 08H) roughly 18.2 times per second. This hardware interrupt occurs regardless of what else the system is doing, so an Interrupt 08H ISR a convenient place to check whether *HotFlag* has been set.

As in the case of the Interrupt 09H handler, the TSR's Interrupt 08H handler passes control to previous Interrupt 08H handlers before it proceeds with its own work. This procedure is particularly important with Interrupt 08H because the ROM BIOS Interrupt 08H handler, which maintains the system's time-of-day clock and resets the system's Intel 8259A Programmable Interrupt Controller, must execute before the next timer tick can occur. The TSR's handler therefore defers its own work until control has returned after previous Interrupt 08H handlers have executed.

The only function of the TSR's Interrupt 08H handler is to attempt to transfer control to the RAM-resident application. The routine *VerifyTSRState* performs this task. It first examines the contents of *HotFlag* to determine whether a hot-key sequence has been detected. If so, it examines the state of the MS-DOS InDOS and critical error flags, the current status of hardware interrupts, and the current status of any non-reentrant ROM BIOS routines that might be executing.

If *HotFlag* is nonzero, the InDOS and critical error flags are both zero, no hardware interrupts are currently being serviced, and no non-reentrant ROM BIOS code has been interrupted, the Interrupt 08H handler activates the RAM-resident utility. Otherwise, nothing happens until the next timer tick, when the handler executes again.

While *HotFlag* is nonzero, the Interrupt 08H handler continues to monitor system status until MS-DOS, the ROM BIOS, and the hardware interrupts are all in a stable state. Often

the system status is stable at the time the hot-key sequence is detected, so the RAM-resident application runs immediately. Sometimes, however, system activities such as prolonged disk reads or writes can preclude the activation of the RAM-resident utility for several seconds after the hot-key sequence has been detected. The handler could be designed to detect this situation (for example, by decrementing *HotFlag* on each timer tick) and return an error status or display a message to the user.

A more serious difficulty arises when the MS-DOS default command processor (COMMAND.COM) is waiting for keyboard input. In this situation, Interrupt 21H Function 01H (Character Input with Echo) is executing, so InDOS is nonzero and the Interrupt 08H handler can never detect a state in which it can activate the RAM-resident utility. This problem is solved by providing a custom handler for Interrupt 28H (the MS-DOS idle interrupt), which is executed by Interrupt 21H Function 01H each time it loops as it waits for a keypress. The only difference between the Interrupt 28H handler and the Interrupt 08H handler is that the Interrupt 28H handler can activate the RAM-resident application when the value of InDOS is 1, which is reasonable because InDOS must have been incremented when Interrupt 21H Function 01H started to execute.

The interrupt service routines for ROM BIOS Interrupts 05H, 10H, and 13H do nothing more than increment and decrement flags that indicate whether these interrupts are being processed by ROM BIOS routines. These flags are inspected by the TSR's Interrupt 08H and 28H handlers.

Executing the RAM-resident application

When the RAM-resident application is first activated, it runs in the context of the program that was interrupted; that is, the contents of the registers, the video display mode, the current PSP, and the current DTA all belong to the interrupted program. The resident application is responsible for preserving the registers and updating MS-DOS with its PSP and DTA values.

The RAM-resident application preserves the previous contents of the CPU registers on its own stack to avoid overflowing the interrupted program's stack. It then installs its own handlers for Control-Break (Interrupt 1BH), Control-C (Interrupt 23H), and critical error (Interrupt 24H). (Otherwise, the interrupted program's handlers would take control if the user pressed Ctrl-Break or Ctrl-C or if an MS-DOS critical error occurred.) These handlers perform no action other than setting flags that can be inspected later by the RAM-resident application, which could then take appropriate action.

The application uses Interrupt 21H Functions 50H and 51H to update MS-DOS with the address of its PSP. If the application is running under MS-DOS versions 2.x, the critical error flag is set before Functions 50H and 51H are executed so that *AuxStack* is used for the call instead of *IOStack*, to avoid corrupting *IOStack* in the event that InDOS is 1.

The application preserves the current extended error information with a call to Interrupt 21H Function 59H. Otherwise, the RAM-resident application might be activated immediately after a critical error occurred in the interrupted program but before the interrupted

program had executed Function 59H and, if a critical error occurred in the TSR application, the interrupted program's extended error information would inadvertently be destroyed.

This example also shows how to update the MS-DOS default DTA using Interrupt 21H Functions 1AH and 2FH, although in this case this step is not necessary because the DTA is never used within the application. In practice, the DTA should be updated only if the RAM-resident application includes calls to Interrupt 21H functions that use a DTA (Functions 11H, 12H, 14H, 15H, 21H, 22H, 27H, 28H, 4EH, and 4FH).

After the resident interrupt handlers are installed and the PSP, DTA, and extended error information have been set up, the RAM-resident application can safely execute any Interrupt 21H function calls except those that use *IOWrite* (Functions 01H through 0CH). These functions cannot be used within a RAM-resident application even if the application sets the critical error flag to force the use of the auxiliary stack, because they also use other non-reentrant data structures such as input/output buffers. Thus, a RAM-resident utility must rely either on user-written console input/output functions or, as in the example, on ROM BIOS functions.

The application terminates by returning the interrupted program's extended error information, DTA, and PSP to MS-DOS, restoring the previous Interrupt 1BH, 23H, and 24H handlers, and restoring the previous CPU registers and stack.

Richard Wilton

Article 12

Exception Handlers

Exceptions are system events directly related to the execution of an application program; they ordinarily cause the operating system to abort the program. Exceptions are thus different from errors, which are minor unexpected events (such as failure to find a file on disk) that the program can be expected to handle appropriately. Likewise, they differ from external hardware interrupts, which are triggered by events (such as a character arriving at the serial port) that are not directly related to the program's execution.

The computer hardware assists MS-DOS in the detection of some exceptions, such as an attempt to divide by zero, by generating an internal hardware interrupt. Exceptions related to peripheral devices, such as an attempt to read from a disk drive that is not ready or does not exist, are called *critical* errors. Instead of causing a hardware interrupt, these exceptions are typically reported to the operating system by device drivers. MS-DOS also supports a third type of exception, which is triggered by the entry of a Control-C or Control-Break at the keyboard and allows the user to signal that the current program should be terminated immediately.

MS-DOS contains built-in handlers for each type of exception and so guarantees a minimum level of system stability that requires no effort on the part of the application programmer. For some applications, however, these default handlers are inadequate. For example, if a communications program that controls the serial port directly with custom interrupt handlers is terminated by the operating system without being given a chance to turn off serial-port interrupts, the next character that arrives on the serial line will trigger an interrupt for which a handler is no longer present in memory. The result will be a system crash. Accordingly, MS-DOS allows application programs to install custom exception handlers so that they can shut down operations in an orderly way when an exception occurs.

This article examines the default exception handlers provided by MS-DOS and discusses methods programmers can use to replace those routines with handlers that are more closely matched to specific application requirements.

Overview

Two major exception handlers of importance to application programmers are supported under all versions of MS-DOS. The first, the Control-C exception handler, terminates the program and is invoked when the user enters a Ctrl-C or Ctrl-Break keystroke; the address

of this handler is found in the vector for Interrupt 23H. The second, the critical error exception handler, is invoked if MS-DOS detects a critical error while servicing an I/O request. (A critical error is a hardware error that makes normal completion of the request impossible.) This exception handler displays the familiar *Abort, Retry, Ignore* prompt; its address is saved in the vector for Interrupt 24H.

When a program begins executing, the addresses in the Interrupt 23H and 24H vectors usually point to the system's default Control-C and critical error handlers. If the program is a child process, however, the vectors might point to exception handlers that belong to the parent process, if the immediate parent is not COMMAND.COM. In any case, the application program can install its own custom handler for Control-C or critical error exceptions simply by changing the address in the vector for Interrupt 23H or Interrupt 24H so that the vector points to the application's own routine. When the program performs a final exit by means of Interrupt 21H Function 00H (Terminate Process), Function 31H (Terminate and Stay Resident), Function 4CH (Terminate Process with Return Code), Interrupt 20H (Terminate Process), or Interrupt 27H (Terminate and Stay Resident), MS-DOS restores the previous contents of the Interrupt 23H and 24H vectors.

Note that Interrupts 23H and 24H *never* occur as externally generated hardware interrupts in an MS-DOS system. The vectors for these interrupts are used simply as storage areas for the addresses of the exception handlers.

MS-DOS also contains default handlers for the Control-Break event detected by the ROM BIOS in IBM PCs and compatible computers and for some of the Intel microprocessor exceptions that generate actual hardware interrupts. These exception handlers are not replaced by application programs as often as the Control-C and critical error handlers. The interrupt vectors that contain the addresses of these handlers are *not* restored by MS-DOS when a program exits.

The address of the Control-Break handler is saved in the vector for Interrupt 1BH and is invoked by the ROM BIOS whenever the Ctrl-Break key combination is detected. The default MS-DOS handler normally flushes the keyboard input buffer and substitutes Control-C for Control-Break, and the Control-C is later handled by the Control-C exception handler. The default handlers for exceptions that generate hardware interrupts either abort the current program (as happens with Divide by Zero) or bring the entire system to a halt (as for a memory parity error).

The Control-C Handler

The vector for Interrupt 23H points to code that is executed whenever MS-DOS detects a Control-C character in the keyboard input buffer. When the character is detected, MS-DOS executes a software Interrupt 23H.

In response to Interrupt 23H, the default Control-C exception handler aborts the current process. Files that were opened with handles are closed (FCB-based files are not), but no

other cleanup is performed. Thus, unsaved data can be left in buffers, some files might not be processed, and critical addresses, such as the vectors for custom interrupt handlers, might be left pointing into free RAM. If more complete control over process termination is wanted, the application should replace the default Control-C handler with custom code. See Customizing Control-C Handling below.

The Control-Break exception handler, pointed to by the vector for Interrupt 1BH, is closely related to the Control-C exception handler in MS-DOS systems on the IBM PC and close compatibles but is called by the ROM BIOS keyboard driver on detection of the Ctrl-Break keystroke combination. Because the Control-Break exception is generated by the ROM BIOS, it is present only on IBM PC-compatible machines and is not a standard feature of MS-DOS. The default ROM BIOS handler for Control-Break is a simple interrupt return — in other words, no action is taken to handle the keystroke itself, other than converting the Ctrl-Break scan code to an extended character and passing it through to MS-DOS as normal keyboard input.

To account for as many hardware configurations as possible, MS-DOS redirects the ROM BIOS Control-Break interrupt vector to its own Control-Break handler during system initialization. The MS-DOS Control-Break handler sets an internal flag that causes the Ctrl-Break keystroke to be interpreted as a Ctrl-C keystroke and thus causes Interrupt 23H to occur.

Customizing Control-C handling

The exception handlers most often neglected by application programmers — and most often responsible for major program failures — are the default exception handlers invoked by the Ctrl-C and Ctrl-Break keystrokes. Although the user must be able to recover from a runaway condition (the reason for Ctrl-C capability in the first place), any exit from a complex program must also be orderly, with file buffers flushed to disk, directories and indexes updated, and so on. The default Control-C and Control-Break handlers do not provide for such an orderly exit.

The simplest and most direct way to deal with Ctrl-C and Ctrl-Break keystrokes is to install new exception handlers that do nothing more than an IRET and thus take MS-DOS out of the processing loop entirely. This move is not as drastic as it sounds: It allows an application to check for and handle the Ctrl-C and Ctrl-Break keystrokes at its convenience when they arrive through the normal keyboard input functions and prevents MS-DOS from terminating the program unexpectedly.

The following example sets the Interrupt 23H and Interrupt 1BH vectors to point to an IRET instruction. When the user presses Ctrl-C or Ctrl-Break, the keystroke combination is placed into the keyboard buffer like any other keystroke. When it detects the Ctrl-C or Ctrl-Break keystroke, the executing program should exit properly (if that is the desired action) after an appropriate shutdown procedure.

To install the new exception handlers, the following procedure (*set_int*) should be called while the main program is initializing:

```

_DATA segment para public 'DATA'
oldint1b dd 0 ; original INT 1BH vector
oldint23 dd 0 ; original INT 23H vector
_DATA ends
_TEXT segment byte public 'CODE'
assume cs:_TEXT,ds:_DATA,es:NOTHING
myint1b: ; handler for Ctrl-Break
myint23: ; handler for Ctrl-C
iret

set_int proc near
mov ax,351bh ; get current contents of
int 21h ; Int 1BH vector and save it
mov word ptr oldint1b,bx
mov word ptr oldint1b+2,es
mov ax,3523h ; get current contents of
int 21h ; Int 23H vector and save it
mov word ptr oldint23,bx
mov word ptr oldint23+2,es
push ds ; save our data segment
push cs ; let DS point to our
pop ds ; code segment
mov dx,offset myint1b
mov ax,251bh ; set interrupt vector 1BH
int 21h ; to point to new handler
mov dx,offset myint23
mov ax,2523h ; set interrupt vector 23H
int 21h ; to point to new handler
pop ds ; restore our data segment
ret ; back to caller
set_int endp
_TEXT ends

```

The application can use the following routine to restore the original contents of the vectors pointing to the Control-C and Control-Break exception handlers before making a final exit back to MS-DOS. Note that, although MS-DOS restores the Interrupt 23H vector to its previous contents, the application *must* restore the Interrupt 1BH vector itself.

```

rest_int proc near
push ds ; save our data segment
mov dx,word ptr oldint23
mov ds,word ptr oldint23+2
mov ax,2523h ; restore original contents
int 21h ; of Int 23H vector
pop ds ; restore our data segment
push ds ; then save it again
mov dx,word ptr oldint1B
mov ds,word ptr oldint1B+2
mov ax,251Bh ; restore original contents
int 21h ; of Int 1BH vector
pop ds ; get back our data segment
ret ; return to caller
rest_int endp

```

The preceding example simply prevents MS-DOS from terminating an application when a Ctrl-C or Ctrl-Break keystroke is detected. Program termination is still often the ultimate goal, but after a more orderly shutdown than is provided by the MS-DOS default Control-C handler. The following exception handler allows the program to exit more gracefully:

```
myint1b:                ; Control-Break exception handler
    iret                ; do nothing
myint23:                ; Control-C exception handler
    call    safe_shut_down ; release interrupt vectors,
                                ; close files, etc.
    jmp     program_exit_point
```

Note that because the Control-Break handler is invoked by the ROM BIOS keyboard driver and MS-DOS is not reentrant, MS-DOS services (such as closing files and terminating with return code) cannot be invoked during processing of a Control-Break exception. In contrast, any MS-DOS Interrupt 21H function call can be used during the processing of a Control-C exception. Thus, the Control-Break handler in the preceding example does nothing, whereas the Control-C handler performs orderly shutdown of the application.

Most often, however, neither a handler that does nothing nor a handler that shuts down and terminates is sufficient for processing a Ctrl-C (or Ctrl-Break) keystroke. Rather than simply prevent Control-C processing, software developers usually prefer to have a Ctrl-C keystroke signal some important action without terminating the program. Using methods similar to those above, the programmer can replace Interrupts 1BH and 23H with a routine like the following:

```
myint1b:                ; Control-Break exception handler
myint23:                ; Control-C exception handler
    call    control_c_happened
    iret
```

Notes on processing Control-C

The preceding examples assume the programmer wants to treat Control-C and Control-Break the same way, but this is not always desirable. Control-C and Control-Break are not the same, and the difference between the two should be kept in mind: The Control-Break handler is invoked by a keyboard-input interrupt and can be called at any time; the Control-C handler is called only at "safe" points during the processing of MS-DOS Interrupt 21H functions. Also, even though MS-DOS restores the Interrupt 23H vector on exit from a program, the *application* must restore the previous contents of the Interrupt 1BH vector before exiting. If this interrupt vector is not restored, the next Ctrl-Break keystroke will cause the machine to attempt to execute an undetermined piece of code or data and will probably crash the system.

Although it is generally desirable to take control of the Control-C and Control-Break interrupts, control should be retained only as long as necessary. For example, a RAM-resident pop-up application should take over Control-C and Control-Break handling only when it is activated, and it should restore the previous contents of the Interrupt 1BH and Interrupt 23H vectors before it returns control to the foreground process.

The Critical Error Handler

When MS-DOS detects a critical error — an error that prevents successful completion of an I/O operation — it calls the exception handler whose address is stored in the vector for Interrupt 24H. Information about the operation in progress and the nature of the error is passed to the exception handler in the CPU registers. In addition, the contents of all the registers at the point of the original MS-DOS call are pushed onto the stack for inspection by the exception handler.

The action of MS-DOS's default critical error handler is to present a message such as

```
Error type error action device
Abort, Retry, Ignore?
```

This message signals a hardware error from which MS-DOS cannot recover without user intervention. For example, if the user enters the command

```
C>DIR A: <Enter>
```

but drive A either does not contain a disk or the disk drive door is open, the MS-DOS critical error handler displays the message

```
Not ready error reading drive A
Abort, Retry, Ignore?
```

I (Ignore) simply tells MS-DOS to forget that an error occurred and continue on its way. (Of course, if the error occurred during the writing of a file to disk, the file is generally corrupted; if the error occurred during reading, the data might be incorrect.)

R (Retry) gives the application a second chance to access the device. The critical error handler returns information to MS-DOS that says, in effect, "Try again; maybe it will work this time." Sometimes, the attempt succeeds (as when the user closes an open drive door), but more often the same or another critical error occurs.

A (Abort) is the problem child of Interrupt 24H. If the user responds with *A*, the application is terminated immediately. The directory structure is not updated for open files, interrupt vectors are left pointing to inappropriate locations, and so on. In many cases, restarting the system is the only safe thing to do at this point.

Note: Beginning with version 3.3, an *F (Fail)* option appears in the message displayed by MS-DOS's default critical error handler. When *Fail* is selected, the current MS-DOS function is terminated and an error condition is returned to the calling program. For example, if a program calls Interrupt 21H Function 3DH to open a file on drive A but the drive door is open, choosing *F* in response to the error message causes the function call to return with the carry flag set, indicating that an error occurred but processing continues.

Like the Control-C exception handler, the default critical error exception handler can and should be replaced by an application program when complete control of the system is desired. The program installs its own handler simply by placing the address of the new handler in the vector for Interrupt 24H; MS-DOS restores the previous contents of the Interrupt 24H vector when the program terminates.

Unlike the Control-C handler, however, the critical error handler must be kept within carefully defined limits to preserve the stability of the operating system. Programmers must rigidly adhere to the structure described in the following pages for passing information to and from an Interrupt 24H handler.

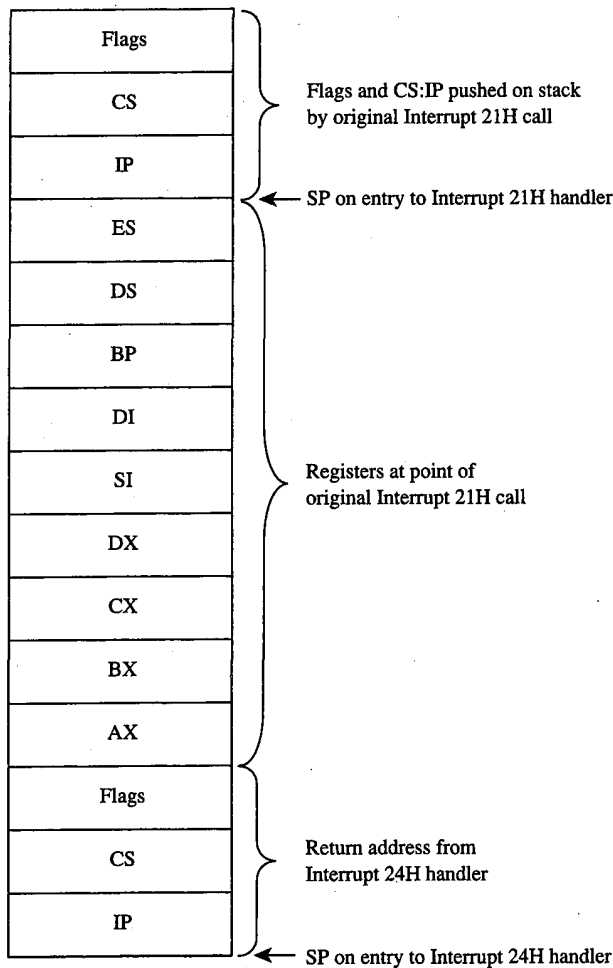


Figure 12-1. The stack contents at entry to a critical error exception handler.

Mechanics of critical error handling

MS-DOS critical error handling has two components: the exception handler, whose address is saved in the Interrupt 24H vector and which can be replaced by an application program; and an internal routine inside MS-DOS. The internal routine sets up the information to be passed to the exception handler on the stack and in registers and, in turn, calls the exception handler itself. The internal routine also responds to the values returned by the critical error handler when that handler executes an IRET to return to the MS-DOS kernel.

Before calling the exception handler, MS-DOS arranges the stack (Figure 12-1 on the preceding page) so the handler can inspect the location of the error and register contents at the point in the original MS-DOS function call that led to the critical error.

When the critical error handler is called by the internal routine, four registers may contain important information: AX, DI, BP, and SI. (With MS-DOS versions 1.x, only the AX and DI registers contain significant information.) The information passed to the handler in the registers differs somewhat, depending on whether a character device or a block device is causing the error.

Block-device (disk-based) errors

If the critical error handler is entered in response to a block-device (disk-based) error, registers BP:SI contain the segment:offset of the device driver header for the device causing the error and bit 7 (the high-order bit) of the AH register is zero. The remaining bits of the AH register contain the following information (bits 3 through 5 apply only to MS-DOS versions 3.1 and later):

Bit	Value	Meaning
0	0	Read operation
	1	Write operation
1-2		Indicate the affected disk area:
	00	MS-DOS
	01	File allocation table
	10	Root directory
	11	Files area
3	0	Fail response not allowed
	1	Fail response allowed
4	0	Retry response not allowed
	1	Retry response allowed
5	0	Ignore response not allowed
	1	Ignore response allowed
6	0	Undefined

The AL register contains the designation of the drive where the error occurred; for example, AL = 00H (drive A), AL = 01H (drive B), and so on.

The lower half of the DI register contains the following error codes (the upper half of this register is undefined):

Error Code	Meaning
00H	Write-protected disk
01H	Unknown unit
02H	Drive not ready
03H	Invalid command
04H	Data error (CRC)
05H	Length of request structure invalid
06H	Seek error
07H	Non-MS-DOS disk
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0FH	Invalid disk change (version 3.0 or later)

Note: With versions 1.x, the only valid error codes are 00H, 02H, 04H, 06H, 08H, 0AH, and 0CH.

Before calling the critical error handler for a disk-based error, MS-DOS tries from one to five times to perform the requested read or write operation, depending on the type of operation. Critical disk errors result only from Interrupt 21H operations, not from failed sector-read and sector-write operations attempted with Interrupts 25H and 26H.

Character-device errors

If the critical error handler is called from the MS-DOS kernel with bit 7 of the AH register set to 1, either an error occurred on a character device or the memory image of the file allocation table is bad (a rare occurrence). Again, registers BP:SI contain the segment and offset of the device driver header for the device causing the critical error. The exception handler can inspect bit 15 of the device attribute word at offset 04H in the device header to confirm that the error was caused by a character device — this bit is 0 for block devices and 1 for character devices. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

If the error was caused by a character device, the lower half of the DI register contains error codes as described above and the contents of the AL register are undefined. The exception handler can inspect the other fields of the device header to obtain the logical name of the character device; to determine whether that device is the standard input, standard output, or both; and so on.

Critical error processing

The critical error exception handler is entered from MS-DOS with interrupts disabled. Because an MS-DOS system call is already in progress and MS-DOS is not reentrant, the

handler cannot request any MS-DOS system services other than Interrupt 21H Functions 01 through 0CH (character I/O functions), Interrupt 21H Function 30H (Get MS-DOS Version Number), and Interrupt 21H Function 59H (Get Extended Error Information). These functions use a special stack so that they can be called during error processing.

In general, the critical error handler must preserve all but the AL register. It must not change the contents of the device header pointed to by BP:SI. The handler must return to the MS-DOS kernel with an IRET, passing an action code in register AL as follows:

Value in AL	Meaning
00H	Ignore
01H	Retry
02H	Terminate process
03H	Fail current system call

These values correspond to the options presented by the MS-DOS default critical error handler. The default handler prompts the user for input, places the appropriate return information in the AL register, and immediately issues an IRET instruction.

Note: Although the *Fail* option is displayed by the MS-DOS default critical error handler in versions 3.3 and later, the *Fail* option inside the handler was added in version 3.1.

With MS-DOS versions 3.1 and later, if the handler returns an action code in AL that is not allowed for the error in question (bits 3 through 5 of the AH register at the point of call), MS-DOS reacts according to the following rules:

If *Ignore* is specified by AL = 00H but is not allowed because bit 5 of AH = 0, the response defaults to *Fail* (AL = 03H).

If *Retry* is specified by AL = 01H but is not allowed because bit 4 of AH = 0, the response defaults to *Fail* (AL = 03H).

If *Fail* is specified by AL = 03H but is not allowed because bit 3 of AH = 0, the response defaults to *Abort*.

Custom critical error handlers

Each time it receives control, COMMAND.COM restores the Interrupt 24H vector to point to the system's default critical error handler and displays a prompt to the user. Consequently, a single custom handler cannot terminate and stay resident to provide critical error handling services for subsequent application programs. Each program that needs better critical error handling than MS-DOS provides must contain its own critical error handler.

Figure 12-2 contains a simple critical error handler, INT24.ASM, written in assembly language. In the form shown, INT24.ASM is no more than a functional replacement for the MS-DOS default critical error handler, but it can be used as the basis for more sophisticated handlers that can be incorporated into application programs.

INT24.ASM contains three routines:

Routine	Action
<i>get24</i>	Saves the previous contents of the Interrupt 24H critical error handler vector and stores the address of the new critical error handler into the vector.
<i>res24</i>	Restores the address of the previous critical error handler, which was saved by a call to <i>get24</i> , into the Interrupt 24 vector.
<i>int24</i>	Replaces the MS-DOS critical error handler.

A program wishing to substitute the new critical error handler for the system's default handler should call the *get24* routine during its initialization sequence. If the program wishes to revert to the system's default handler during execution, it can accomplish this with a call to the *res24* routine. Otherwise, a call to *res24* (and the presence of the routine itself in the program) is not necessary, because MS-DOS automatically restores the Interrupt 24H vector to its previous value when the program exits, from information stored in the program segment prefix (PSP).

The replacement critical error handler, *int24*, is simple. First it saves all registers; then it displays a message that a critical error has occurred and prompts the user to enter a key selecting one of the four possible options: *Abort*, *Retry*, *Ignore*, or *Fail*. If an illegal key is entered, the prompt is displayed again; otherwise, the action code corresponding to the key is extracted from a table and placed in the AL register, the other registers are restored, and control is returned to the MS-DOS kernel with an IRET instruction.

Note that the handle read and write functions (Interrupt 21H Functions 3FH and 40H), which would normally be preferred for interaction with the display and keyboard, cannot be used in a critical error handler.

```

name      int24
title     INT24 Critical Error Handler

;
; INT24.ASM - Replacement critical error handler
; by Ray Duncan, September 1987
;

cr        equ     0dh           ; ASCII carriage return
lf        equ     0ah           ; ASCII linefeed

DGROUP   group   _DATA

_DATA    segment word public 'DATA'

save24   dd      0             ; previous contents of Int 24H
                                ; critical error handler vector

```

Figure 12-2. INT24.ASM, a replacement Interrupt 24H handler.

(more)

```

                                ; prompt message used by
                                ; critical error handler
prompt db      cr,lf,'Critical Error Occurred: '
       db      'Abort, Retry, Ignore, Fail? $'

keys   db      'aArRiIf'      ; possible user response keys
keys_len equ   $-keys        ; (both cases of each allowed)

codes  db      2,2,1,1,0,0,3,3 ; codes returned to MS-DOS kernel
                                ; for corresponding response keys

_DATA  ends

_TEXT  segment word public 'CODE'

       assume  cs:_TEXT,ds:DGROUP

       public  get24
get24  proc    near            ; set Int 24H vector to point
                                ; to new critical error handler

       push   ds              ; save segment registers
       push   es

       mov    ax,3524h        ; get address of previous
       int    21h            ; INT 24H handler and save it

       mov    word ptr save24,bx
       mov    word ptr save24+2,es

       push   cs              ; set DS:DX to point to
       pop    ds              ; new INT 24H handler
       mov    dx,offset _TEXT:int24
       mov    ax,2524h        ; then call MS-DOS to
       int    21h            ; set the INT 24H vector

       pop    es              ; restore segment registers
       pop    ds
       ret                    ; and return to caller

get24  endp

       public  res24
res24  proc    near            ; restore original contents
                                ; of Int 24H vector

       push   ds              ; save our data segment

```

Figure 12-2. Continued.

(more)

```

        lds    dx,save24    ; put address of old handler
        mov    ax,2524h    ; back into INT 24H vector
        int    21h

        pop    ds          ; restore data segment
        ret              ; and return to caller

res24   endp

;
; This is the replacement critical error handler.  It
; prompts the user for Abort, Retry, Ignore, or Fail and
; returns the appropriate code to the MS-DOS kernel.
;
int24   proc    far          ; entered from MS-DOS kernel

        push   bx          ; save registers
        push   cx
        push   dx
        push   si
        push   di
        push   bp
        push   ds
        push   es

int24a: mov    ax,DGROUP    ; display prompt for user
        mov    ds,ax        ; using function 09H (print string
        mov    es,ax        ; terminated by $ character)
        mov    dx,offset prompt
        mov    ah,09h
        int    21h

        mov    ah,01h      ; get user's response
        int    21h        ; function 01H = read one character

        mov    di,offset keys ; look up code for response key
        mov    cx,keys_len
        cld
        repne scasb
        jnz    int24a      ; prompt again if bad response

        ; set AL = action code for MS-DOS
        ; according to key that was entered:
        ; 0 = ignore, 1 = retry, 2 = abort, 3 = fail
        mov    al,[di+keys_len-1]

        pop    es          ; restore registers
        pop    ds
        pop    bp
        pop    di
        pop    si

```

Figure 12-2. Continued.

(more)

```

        pop     dx
        pop     cx
        pop     bx
        iret                    ; exit critical error handler

int24   endp

_TEXT   ends

        end

```

Figure 12-2. Continued.

Hardware-generated Exception Interrupts

Intel reserved the vectors for Interrupts 00H through 1FH (Table 12-1) for exceptions generated by the execution of various machine instructions. Handling of these chip-dependent internal interrupts can vary from one make of MS-DOS machine to another; some such differences are mentioned in the discussion.

Table 12-1. Intel Reserved Exception Interrupts.

Interrupt Number	Definition
00H	Divide by Zero
01H	Single-Step
02H	Nonmaskable Interrupt (NMI)
03H	Breakpoint Trap
04H	Overflow Trap
05H	BOUND Range Exceeded*
06H	Invalid Opcode*
07H	Coprocessor not Available†
08H	Double-Fault Exception†
09H	Coprocessor Segment Overrun†
0AH	Invalid Task State Segment (TSS)†
0BH	Segment not Present†
0CH	Stack Exception†
0DH	General Protection Exception†
0EH	Page Fault‡
0FH	(Reserved)
10H	Coprocessor Error†
11–1FH	(Reserved)

* The 80186, 80286, and 80386 microprocessors only.

† The 80286 and 80386 microprocessors only.

‡ The 80386 microprocessor only.

Note: A number of these reserved exception interrupts generally do not occur in MS-DOS because they are generated only when the 80286 or 80386 microprocessor is operating in protected mode. The following discussions do not cover these interrupts.

Divide by Zero (Interrupt 00H)

An Interrupt 00H occurs whenever a DIV or IDIV operation fails to terminate within a reasonable period of time. The interrupt is triggered by a mathematical anomaly: Division by zero is inherently undefined. To handle such situations, Intel built special processing into the DIV and IDIV instructions to ensure that the condition does not cause the processor to lock up. Although the assumption underlying Interrupt 00H is an attempt to divide by zero (a condition that will never terminate), the interrupt can also be triggered by other error conditions, such as a quotient that is too large to fit in the designated register (AX or AL).

The ROM BIOS handler for Interrupt 00H in the IBM PC and close compatibles is a simple IRET instruction. During the MS-DOS startup process, however, MS-DOS modifies the interrupt vector to point to its own handler—a routine that issues the warning message *Divide by Zero* and aborts the current application. This abort procedure can leave the computer and operating system in an extremely unstable state. If the default handler is used, the system should be restarted immediately and an attempt should be made to find and eliminate the cause of the error. A better approach, however, is to provide a replacement handler that treats Interrupt 00H much as MS-DOS treats Interrupt 24H.

Single-Step (Interrupt 01H)

If the trap flag (bit 8 of the microprocessor's 16-bit flags register) is set, Interrupt 01H occurs at the end of every instruction executed by the processor. By default, Interrupt 01H points to a simple IRET instruction, so the net effect is as if nothing happened. However, debugging programs, which are the only applications that use this interrupt, modify the interrupt vector to point to their own handlers. The interrupt can then be used to allow a debugger to single-step through the machine instructions of the program being debugged, as DEBUG does with its T (Trace) command.

Nonmaskable Interrupt, or NMI (Interrupt 02H)

In the hardware architecture of IBM PCs and close compatibles, Interrupt 02H is invoked whenever a memory parity error is detected. MS-DOS provides no handler, because this error, as a hardware-related problem, is in the domain of the ROM BIOS.

In response to the Interrupt 02H, the default ROM BIOS handler displays a message and locks the machine, on the assumption that bad memory prevents reliable system operation. Many programmers, however, prefer to include code that permits orderly shutdown of the system. Replacing the ROM BIOS parity trap routine can be dangerous, though, because a parity error detected in memory means the contents of RAM are no longer reliable—even the memory locations containing the NMI handler itself might be defective.

Breakpoint Trap (Interrupt 03H)

Interrupt 03H, which is used in conjunction with Interrupt 01H for debugging, is invoked by a special 1-byte opcode (0CCH). During a debugging session, a debugger modifies the vector for Interrupt 03H to point to its own handler and then replaces 1 byte of program opcode with the 0CCH opcode at any location where a breakpoint is needed.

When a breakpoint is reached, the 0CCH opcode triggers Interrupt 03H and the debugger regains control. The debugger then restores the original opcode in the program being debugged and issues a prompt so that the user can display or alter the contents of memory or registers. The use of Interrupt 03H is illustrated by DEBUG and SYMDEB's breakpoint capabilities.

Overflow Trap (Interrupt 04H)

If the overflow bit (bit 11) in the microprocessor's flags register is set, Interrupt 04H occurs when the INTO (Interrupt on Overflow) instruction is executed. The overflow bit can be set during prior execution of any arithmetic instruction (such as MUL or IMUL) that can produce an overflow error.

The ROM BIOS of the IBM PC and close compatibles initializes the Interrupt 04H vector to point to an IRET, so this interrupt becomes invisible to the user if it is executed. MS-DOS does not have its own handler for Interrupt 04H. However, because the Intel microprocessors also include JO (Jump if Overflow) and JNO (Jump if No Overflow) instructions, applications rarely need the INTO instruction and, hence, seldom have to provide their own Interrupt 04H handlers.

BOUND Range Exceeded (Interrupt 05H)

Interrupt 05H is generated on 80186, 80286, and 80386 microprocessors if a BOUND instruction is executed to test the value of an array index and the index falls outside the limits specified by the instruction's operand. The exception handler is expected to alter the index so that it is correct — when the handler performs an interrupt return (IRET), the CPU reexecutes the BOUND instruction that caused the interrupt.

On IBM PC/AT-compatible machines, the ROM BIOS assignment of the PrtSc (print screen) routine to Interrupt 05H is in conflict with the CPU's use of Interrupt 05H for BOUND exceptions.

Invalid opcode (Interrupt 06H)

Interrupt 06H is generated by the 80186, 80286, and 80386 microprocessors if the current instruction is not a valid opcode — for example, if the machine tries to execute a data statement.

On IBM PC/ATs, Interrupt 06H simply points to an IRET instruction. The ROM BIOS routines of some IBM PC/AT-compatibles, however, provide an interrupt handler that reports an unexpected software Interrupt 06H and asks if the user wants to continue. A *Y* response causes the interrupt handler to skip over the invalid opcode. Unfortunately, because the succeeding opcode is often invalid as well, the user may have the feeling of being trapped in a loop.

Extended Error Information

Under MS-DOS versions 1.x, the operating system provided limited information about errors that occurred during calls to the Interrupt 21H system functions. For example, if a program called Function 0FH to open a file, there were only two possible results: On return, the AL register either contained 00H for a successful open or 0FFH for failure. No further detail was available from the operating system. Although some of these early system calls (such as the read and write functions) returned somewhat more information, the 1.x versions of MS-DOS were essentially limited to success/failure return codes.

Beginning with version 2.0 and the introduction of the handle concept, additional error information became available. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management. For example, if a program attempts to open a file with Interrupt 21H Function 3DH (Open File with Handle), it can check the status of the carry flag on return to detect whether an error occurred. If the carry flag is not set, the call was successful and the AX register contains the file handle. If the carry flag is set, the AX register contains one of the following possible error codes:

Error Code	Meaning
01H	Invalid function code
02H	File not found
03H	Path not found
04H	Too many open files (no more handles available)
05H	Access denied
0CH	Invalid access code

In some circumstances, however, even these error codes do not provide enough information. Therefore, beginning with version 3.0, MS-DOS made extended error information available through Interrupt 21H Function 59H (Get Extended Error Information). This function can be called after any other Interrupt 21H function fails, or it can be called from a critical error handler. The extended error codes, briefly described below, maintain compatibility with the MS-DOS versions 2.x error returns and are grouped as follows:

Error Code	Error Group
00H	No error encountered.
01–12H	MS-DOS versions 2.x and 3.x Interrupt 21H errors. These error codes are identical to those returned in the AX register by Functions 38H through 57H if the carry flag is set on return from the function call.
13–1FH	MS-DOS versions 2.x and 3.x Interrupt 24H errors. These error codes are 13H (19) greater than the codes passed to a critical error handler in the lower half of the DI register; that is, if the critical error handler receives error code 04H (CRC error), Interrupt 21H Function 59H returns 17H.
20–58H	Extended error codes, many related to networking and file sharing, for MS-DOS versions 3.0 and later.

Note: The contents of the CPU registers (except CS:IP and SS:SP) are destroyed by a call to Function 59H. Also, as mentioned earlier, this function is available only with MS-DOS versions 3.x, even though it maintains compatibility with error returns in versions 2.x.

On return, Function 59H provides the extended error code in the AX register, the error class (type) in the BH register, a code for the suggested corrective action in the BL register, and the locus of the error in the CH register. These values are defined in the following paragraphs. With MS-DOS or PC-DOS versions 3.x, if an error 22H (invalid disk change) occurs and if the capability is supported by the system's block-device drivers, ES:DI points to an ASCIIZ volume label that designates the disk to be inserted in the drive before the operation is retried.

Error Code (AX register). This value is defined as follows:

Value in AX	Meaning
--------------------	----------------

Interrupt 21H errors (MS-DOS versions 2.0 and later):

01H	Invalid function number
02H	File not found
03H	Path not found
04H	Too many open files (no handles available)
05H	Access denied
06H	Invalid handle
07H	Memory control blocks destroyed
08H	Insufficient memory
09H	Invalid memory-block address
0AH	Invalid environment
0BH	Invalid format
0CH	Invalid access code
0DH	Invalid data
0EH	Reserved
0FH	Invalid disk drive specified
10H	Attempt to remove the current directory
11H	Not same device
12H	No more files

Interrupt 24H errors (MS-DOS versions 2.0 and later):

13H	Attempt to write on write-protected disk
14H	Unknown unit
15H	Drive not ready
16H	Invalid command
17H	Data error based on cyclic redundancy check (CRC)
18H	Length of request structure invalid
19H	Seek error

(more)

Value in AX Meaning

Interrupt 24H errors *(continued)*

1AH	Unknown media type (non-MS-DOS disk)
1BH	Sector not found
1CH	Printer out of paper
10H	Write fault
1EH	Read fault
1FH	General failure

MS-DOS versions 3.x extended errors:

20H	Sharing violation
21H	Lock violation
22H	Invalid disk change
23H	FCB unavailable
24H	Sharing buffer exceeded
25H-31H	Reserved
32H	Network request not supported
33H	Remote computer not listening
34H	Duplicate name on network
35H	Network name not found
36H	Network busy
37H	Device no longer exists on network
38H	Net BIOS command limit exceeded
39H	Error in network adapter hardware
3AH	Incorrect response from network
3BH	Unexpected network error
3CH	Incompatible remote adapter
3DH	Print queue full
3EH	Queue not full
3FH	Not enough room for print file
40H	Network name deleted
41H	Access denied
42H	Incorrect network device type
43H	Network name not found
44H	Network name limit exceeded
45H	Net BIOS session limit exceeded
46H	Temporary pause
47H	Network request not accepted
48H	Print or disk redirection paused
49H-4FH	Reserved
50H	File already exists
51H	Reserved

(more)

Value in AX Meaning

MS-DOS versions 3.x extended errors *(continued)*

52H	Cannot make directory
53H	Failure on Interrupt 24H
54H	Out of structures
55H	Already assigned
56H	Invalid password
57H	Invalid parameter
58H	Network write fault

Locus (CH register). This value provides information on the location of the error:

Value in CH Meaning

01H	Location unknown
02H	Block device; generally caused by a disk error
03H	Network
04H	Serial device; generally caused by a timeout from a character device
05H	Memory; caused by an error in RAM

Error Class (BH register). This value gives the general category of the error:

Value in BH Meaning

01H	Out of resource; out of storage space or I/O channels.
02H	Temporary situation; expected to clear, as in a file or record lock — generally occurs only in a network environment.
03H	Authorization; a problem with permission to access the requested device.
04H	Internal error in system software; generally reflects a system software bug rather than an application or system failure.
05H	Hardware failure; a serious hardware-related problem not the fault of the user program.
06H	System failure; a serious failure of the system software, not directly the fault of the application — generally occurs if configuration files are missing or incorrect.
07H	Application-program error; generally caused by inconsistent function requests from the user program.
08H	File or item not found.
09H	File or item of invalid format or type detected, or an otherwise unsuitable or invalid item requested.
0AH	File or item interlocked by the system.

(more)

Value in BH	Meaning
0BH	Media failure; generally occurs with a bad disk in a drive, a bad spot on the disk, or the like.
0CH	Already exists; generally occurs when application tries to declare a machine name or device that already exists.
0DH	Unknown.

Suggested Action (BL register). One of the most useful returns from Function 59H, this value suggests a corrective action to try:

Value in BL	Meaning
01H	Retry a few times before prompting the user to choose <i>Ignore</i> for the program to continue or <i>Abort</i> to terminate.
02H	Pause for a few seconds between retries and then prompt user as above.
03H	Ask user to reenter the input. In most cases, this solution applies when an incorrect drive specifier or filename was entered. Of course, if the value was hard-coded into the program, the user should not be prompted for input.
04H	Clean up as well as possible, then abort the application. This solution applies when the error is destructive enough that the application cannot safely proceed, but the system is healthy enough to try an orderly shut-down of the application.
05H	Exit from the application as soon as possible, without trying to close files and clean up. This means something is seriously wrong with either the application or the system.
06H	Ignore; error is informational.
07H	Prompt user to perform some action, such as changing floppy disks in a drive and then retry.

Function 59H and older system calls

The Interrupt 21H functions — primarily the FCB-related file and record calls — that return 0FFH in the AL register to indicate that an error has occurred but provide no further information about the type of error include

Function	Name
0FH	Open File with FCB
10H	Close File with FCB
11H	Find First File
12H	Find Next File

(more)

Function	Name
13H	Delete File
16H	Create File with FCB
17H	Rename File
23H	Get File Size

These function calls now exist only to maintain compatibility with MS-DOS versions 1.x. The preferred choices are the handle-style calls available in MS-DOS versions 2.0 and later, which offer full path support and much better error reporting. *See also* SYSTEM CALLS.

If the older calls *must* be used, the program can use Function 59H to obtain more detailed information under MS-DOS version 3.0 or later. For example:

```
myfcb  db      0          ; drive = default
        db     'MYFILE '  ; filename, 8 chars
        db     'DAT'     ; extension, 3 chars
        db     25 dup (0) ; remainder of FCB
        .
        .
        .
        mov    dx,seg myfcb ; DS:DX = FCB
        mov    ds,dx
        mov    dx,offset myfcb
        mov    ah,0fh     ; function 0FH = Open FCB

        int    21h       ; transfer to MS-DOS
        or     al,al     ; test status
        jz     success   ; jump, open succeeded
                        ; open failed, get
                        ; extended error info
        mov    bx,0      ; BX = 00H for ver. 2.x-3.x
        mov    ah,59h    ; function 59H = Get Info
        int    21h       ; transfer to MS-DOS
        or     ax,ax     ; really an error?
        jz     success   ; no error, jump
                        ; test recommended actions
        cmp    bl,01h    ; if BL = 01H retry operation
        jz     retry
        cmp    bl,04h    ; if BL = 04H clean up and exit
        jz     cleanup
        cmp    bl,05h    ; if BL = 05H exit immediately
        jz     panic
        .
        .
        .
```

Function 59H and newer system calls

The function calls listed below were added in MS-DOS versions 2.0 and later. These calls return with the carry flag set if an error occurs; in addition, the AX register contains an error value corresponding to error codes 01H through 12H of the extended error return codes:

Function	Name
MS-DOS versions 2.0 and later:	
38H	Get/Set Current Country
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File
3FH	Read File or Device
40H	Write File or Device
41H	Delete File
42H	Move File Pointer
43H	Get/Set File Attributes
44H	IOCTL (I/O Control for Devices)
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
4BH	Load and Execute Program (EXEC)
4EH	Find First File
4FH	Find Next File
56H	Rename File
57H	Get/Set Date/Time of File
<hr/>	
MS-DOS versions 3.0 and later:	
58H	Get/Set Allocation Strategy
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region
<hr/>	
MS-DOS versions 3.1 and later:	
5EH	Network Machine Name/Printer Setup
5FH	Get/Make Assign List Entry

Although these newer functions have much better error reporting than the older FCB functions, Function 59H is still useful. Regardless of the version of MS-DOS that is running, the error code returned in the AX register from an Interrupt 21H function call is always in the range 0–12H. If a program is running under MS-DOS versions 3.x and wants to obtain one or more of the more specific error codes in the range 20–58H, the program must

follow the failed Interrupt 21H call with a subsequent call to Interrupt 21H Function 59H. The program can then use the code returned by Function 59H in the BL register as a guide to the action to take in response to the error. For example:

```
myfile db      'MYFILE.DAT',0 ; ASCIIZ filename
      .
      .
      .
      mov     dx,seg myfile    ; DS:DX = ASCIIZ filename
      mov     ds,dx
      mov     dx,offset myfile
      mov     ax,3d02h        ; open, read/write
      int     21h             ; transfer to MS-DOS
      jnc     success         ; jump, open succeeded
      .
      .
      .
      mov     bx,0            ; BX = 00H for ver. 2.x-3.x
      mov     ah,59h          ; function 59H = Get Info
      int     21h             ; transfer to MS-DOS
      or      ax,ax           ; really an error?
      jz      success         ; no error, jump
      .
      .
      .
      cmp     bl,01h          ; test recommended actions
      jz      retry           ; if BL = 01H retry operation
      .
      .
      .
```

If the standard critical error handler is replaced with a customized critical handler, Function 59H can also be used to obtain more detailed information about an error inside the handler before either returning control to the application or aborting. The value in the BL register should be used to determine the appropriate action to take or the message to display to the user.

*Jim Kyle
Chip Rabinowitz*

Article 13

Hardware Interrupt Handlers

Unlike software interrupts, which are service requests initiated by a program, hardware interrupts occur in response to electrical signals received from a peripheral device such as a serial port or a disk controller, or they are generated internally by the microprocessor itself. Hardware interrupts, whether external or internal to the microprocessor, are given prioritized servicing by the Intel CPU architecture.

The 8086 family of microprocessors (which includes the 8088, 8086, 80186, 80286, and 80386) reserves the first 1024 bytes of memory (addresses 0000:0000H through 0000:03FFH) for a table of 256 interrupt vectors, each a 4-byte far pointer to a specific interrupt service routine (ISR) that is carried out when the corresponding interrupt is processed. The design of the 8086 family requires certain of these interrupt vectors to be used for specific functions (Table 13-1). Although Intel actually reserves the first 32 interrupts, IBM, in the original PC, redefined usage of Interrupts 05H to 1FH. Most, but not all, of these reserved vectors are used by software, rather than hardware, interrupts; the redefined IBM uses are listed in Table 13-2.

Table 13-1. Intel Reserved Exception Interrupts.

Interrupt Number	Definition
00H	Divide by zero
01H	Single step
02H	Nonmaskable interrupt (NMI)
03H	Breakpoint trap
04H	Overflow trap
05H	BOUND range exceeded*
06H	Invalid opcode*
07H	Coprocessor not available†
08H	Double-fault exception†
09H	Coprocessor segment overrun†
0AH	Invalid task state segment (TSS)†
0BH	Segment not present†
0CH	Stack exception†
0DH	General protection exception†
0EH	Page fault‡

(more)

Table 13-1. *Continued.*

Interrupt Number	Definition
0FH	(Reserved)
10H	Coprocessor error†

*The 80186, 80286, and 80386 microprocessors only.

†The 80286 and 80386 microprocessors only.

‡The 80386 microprocessor only.

Table 13-2. IBM Interrupt Usage.

Interrupt Number	Definition
05H	Print screen
06H	Unused
07H	Unused
08H	Hardware IRQ0 (timer-tick)*
09H	Hardware IRQ1 (keyboard)
0AH	Hardware IRQ2 (reserved)†
0BH	Hardware IRQ3 (COM2)
0CH	Hardware IRQ4 (COM1)
0DH	Hardware IRQ5 (fixed disk)
0EH	Hardware IRQ6 (floppy disk)
0FH	Hardware IRQ7 (printer)
10H	Video service
11H	Equipment information
12H	Memory size
13H	Disk I/O service
14H	Serial-port service
15H	Cassette/network service
16H	Keyboard service
17H	Printer service
18H	ROM BASIC
19H	Restart system
1AH	Get/Set time/date
1BH	Control-Break (user defined)
1CH	Timer tick (user defined)
1DH	Video parameter pointer
1EH	Disk parameter pointer
1FH	Graphics character table

*IRQ = Interrupt request line.

†See Table 13-4.

Nestled in the middle of Table 13-2 are the eight hardware interrupt vectors (08-0FH) IBM implemented in the original PC design. These eight vectors provide the maskable interrupts for the IBM PC-family and close compatibles. Additional IRQ lines built into the IBM PC/AT are discussed under The IRQ Levels below.

The conflicting uses of the interrupts listed in Tables 13-1 and 13-2 have created compatibility problems as the 8086 family of microprocessors has developed. For complete compatibility with IBM equipment, the IBM usage must be followed even when it conflicts with the chip design. For example, a BOUND error occurs if an array index exceeds the specified upper and lower limits (bounds) of the array, causing an Interrupt 05H to be generated. But the 80286 processor used in all AT-class computers will, if a BOUND error occurs, send the contents of the display to the printer, because IBM uses Interrupt 05H for the Print Screen function.

Hardware Interrupt Categories

The 8086 family of microprocessors can handle three types of hardware interrupts. First are the internal, microprocessor-generated exception interrupts (Table 13-1). Second is the nonmaskable interrupt, or NMI (Interrupt 02H), which is generated when the NMI line (pin 17 on the 8088 and 8086, pin 59 on the 80286, pin B8 on the 80386) goes high (active). In the IBM PC family (except the PCjr and the Convertible), the nonmaskable interrupt is designated for memory parity errors. Third are the maskable interrupts, which are usually generated by external devices.

Maskable interrupts are routed to the main processor through a chip called the 8259A Programmable Interrupt Controller (PIC). When it receives an interrupt request, the PIC signals the microprocessor that an interrupt needs service by driving the interrupt request (INTR) line of the main processor to high voltage level. This article focuses on the maskable interrupts and the 8259A because it is through the PIC that external I/O devices (disk drives, serial communication ports, and so forth) gain access to the interrupt system.

Interrupt priorities in the 8086 family

The Intel microprocessors have a built-in priority system for handling interrupts that occur simultaneously. Priority goes to the internal instruction exception interrupts, such as Divide by Zero and Invalid Opcode, because priority is determined by the interrupt number: Interrupt 00H takes priority over all others, whereas the last possible interrupt, 0FFH, would, if present, never be allowed to break in while another interrupt was being serviced. However, if interrupt service is enabled (the microprocessor's interrupt flag is set), any hardware interrupt takes priority over any software interrupt (INT instruction).

The priority sequencing by interrupt number must not be confused with the priority resolution performed by hardware external to the microprocessor. The numeric priority discussed here applies only to interrupts generated within the 8086 family of microprocessor chips and is totally independent of system interrupt priorities established for components external to the microprocessor itself.

Interrupt service routines

For the most part, programmers need not write hardware-specific program routines to service the hardware interrupts. The IBM PC BIOS routines, together with MS-DOS services, are usually sufficient. In some cases, however, MS-DOS and the ROM BIOS do not provide enough assistance to ensure adequate performance of a program. Most notable in this category is communications software, for which programmers usually must access the 8259A and the 8250 Universal Asynchronous Receiver and Transmitter (UART) directly. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Characteristics of Maskable Interrupts

Two major characteristics distinguish maskable interrupts from all other events that can occur in the system: They are totally unpredictable, and they are highly volatile. In general, a hardware interrupt occurs when a peripheral device requires the full attention of the system and data will be irretrievably lost unless the system responds rapidly.

All things are relative, however, and this is especially true of the speed required to service an interrupt request. For example, assume that two interrupt requests occur at essentially the same time. One is from a serial communications port receiving data at 300 bps; the other is from a serial port receiving data at 9600 bps. Data from the first serial port will not change for at least 30 milliseconds, but the second serial port must be serviced within one millisecond to avoid data loss.

Unpredictability

Because maskable interrupts generally originate in response to external physical events, such as the receipt of a byte of data over a communications line, the exact time at which such an interrupt will occur cannot be predicted. Even the timer interrupt request, which by default occurs approximately 18.2 times per second, cannot be predicted by any program that happens to be executing when the interrupt request occurs.

Because of this unpredictability, the system must, if it allows any interrupts to be recognized, be prepared to service all maskable interrupt requests. Conversely, if interrupts cannot be serviced, they must all be disabled. The 8086 family of microprocessors provides the Set Interrupt Flag (STI) instruction to enable maskable interrupt response and the Clear Interrupt Flag (CLI) instruction to disable it. The interrupt flag is also cleared automatically when a hardware interrupt response begins; the interrupt handler should execute STI as quickly as possible to allow higher priority interrupts to be serviced.

Volatility

As noted earlier, a maskable interrupt request must normally be serviced immediately to prevent loss of data, but the concept of immediacy is relative to the data transfer rate of the device requesting the interrupt. The rule is that the currently available unit of data must be processed (at least to the point of being stored in a buffer) before the next such item can

arrive. Except for such devices as disk drives, which always require immediate response, interrupts for devices that receive data are normally much more critical than interrupts for devices that transmit data.

The problems imposed by data volatility during hardware interrupt service are solved by establishing service priorities for interrupts generated outside the microprocessor chip itself. Devices with the slowest transfer rates are assigned lower interrupt service priorities, and the most time-critical devices are assigned the highest priority of interrupt service.

Handling Maskable Interrupts

The microprocessor handles all interrupts (maskable, nonmaskable, and software) by pushing the contents of the flags register onto the stack, disabling the interrupt flag, and pushing the current contents of the CS:IP registers onto the stack.

The microprocessor then takes the interrupt number from the data bus, multiplies it by 4 (the size of each vector in bytes), and uses the result as an offset into the interrupt vector table located in the bottom 1 KB (segment 0000H) of system RAM. The 4-byte address at that location is then used as the new CS:IP value (Figure 13-1).

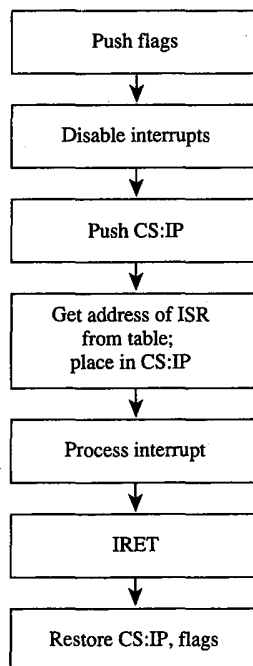


Figure 13-1. General interrupt sequence.

External devices are assigned dedicated interrupt request lines (IRQs) associated with the 8259A. See The IRQ Levels below. When a device requires attention, it sends a signal to the PIC via its IRQ line. The PIC, which functions as an “executive secretary” for the external devices, operates as shown in Figure 13-2. It evaluates the service request and, if appropriate, causes the microprocessor’s INTR line to go high. The microprocessor then checks whether interrupts are enabled (whether the interrupt flag is set). If they are, the flags are pushed onto the stack, the interrupt flag is disabled, and CS:IP is pushed onto the stack.

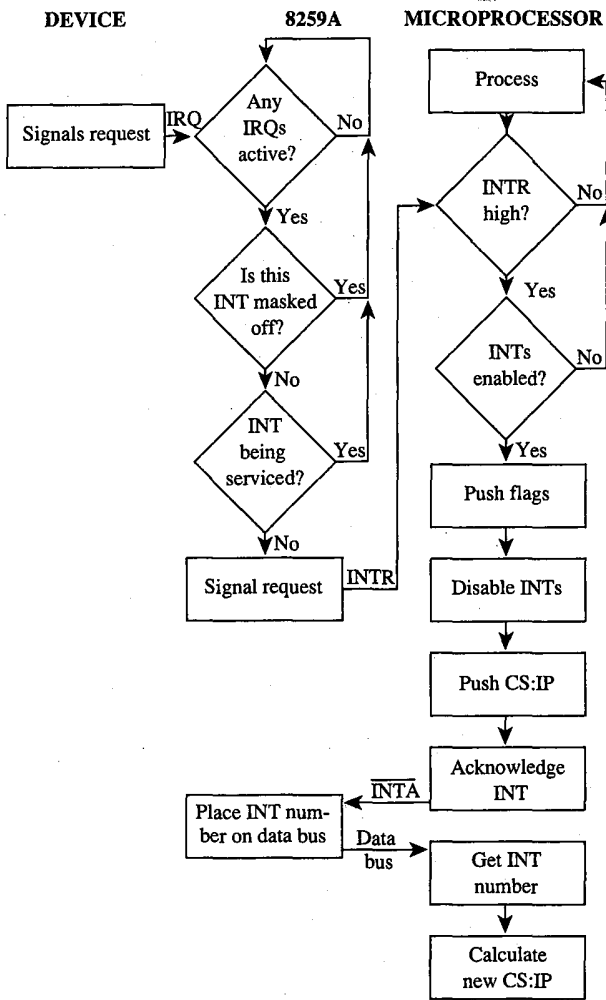


Figure 13-2. Maskable interrupt service.

The microprocessor acknowledges the interrupt request by signaling the 8259A via the interrupt acknowledge (INTA) line. The 8259A then places the interrupt number on the data bus. The microprocessor gets the interrupt number from the data bus and services the interrupt. Before issuing the IRET instruction, the interrupt service routine must issue an end-of-interrupt (EOI) sequence to the 8259A so that other interrupts can be processed. This is done by sending 20H to port 20H. (The similarity of numbers is pure coincidence.) The EOI sequence is covered in greater detail elsewhere. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

The 8259A Programmable Interrupt Controller

The 8259A (Figure 13-3) has a number of internal components, many of them under software control. Only the default settings for the IBM PC family are covered here.

Three registers influence the servicing of maskable interrupts: the interrupt request register (IRR), the in-service register (ISR), and the interrupt mask register (IMR).

The IRR is used to keep track of the devices requesting attention. When a device causes its IRQ line to go high to signal the 8259A that it needs service, a bit is set in the IRR that corresponds to the interrupt level of the device.

The ISR specifies which interrupt levels are currently being serviced; an ISR bit is set when an interrupt has been acknowledged by the CPU (via INTA) and the interrupt number has been placed on the data bus. The ISR bit associated with a particular IRQ remains set until an EOI sequence is received.

The IMR is a read/write register (at port 21H) that masks (disables) specific interrupts. When a bit is set in this register, the corresponding IRQ line is masked and no servicing for it is performed until the bit is cleared. Thus, a particular IRQ can be disabled while all others continue to be serviced.

The fourth major block in Figure 13-3, labeled *Priority resolver*, is a complex logical circuit that forms the heart of the 8259A. This component combines the statuses of the IMR, the ISR, and the IRR to determine which, if any, pending interrupt request should be serviced and then causes the microprocessor's INTR line to go high. The priority resolver can be programmed in a number of modes, although only the mode used in the IBM PC and close compatibles is described here.

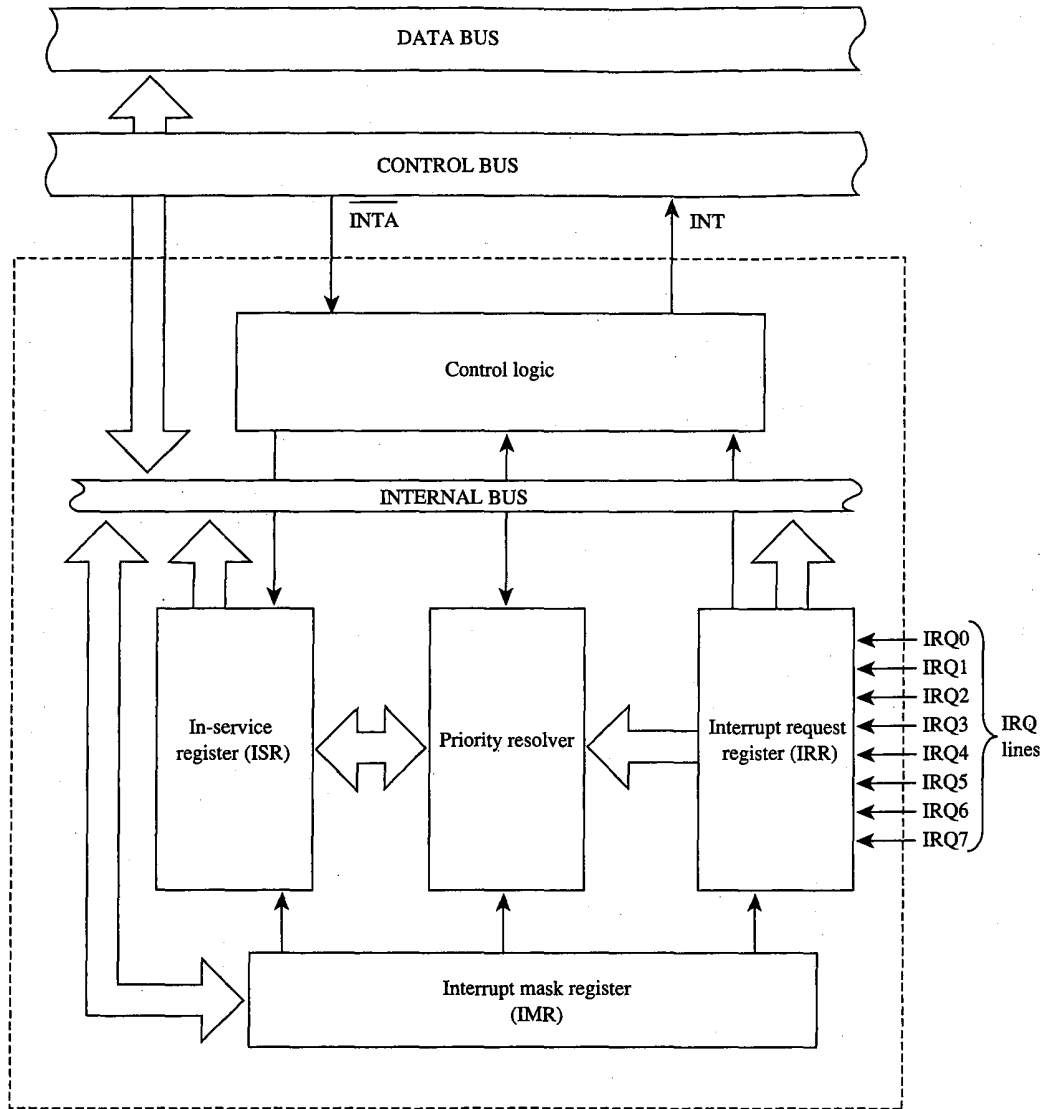


Figure 13-3. Block diagram of the 8259A Programmable Interrupt Controller.

The IRQ levels

When two or more unserved hardware interrupts are pending, the 8259A determines which should be serviced first. The standard mode of operation for the PIC is the fully nested mode, in which IRQ lines are prioritized in a fixed sequence. Only IRQ lines with higher priority than the one currently being serviced are permitted to generate new interrupts.

The highest priority is IRQ0, and the lowest is IRQ7. Thus, if an Interrupt 09H (signaled by IRQ1) is being serviced, only an Interrupt 08H (signaled by IRQ0) can break in. All other interrupt requests are delayed until the Interrupt 09H service routine is completed and has issued an EOI sequence.

Eight-level designs

The IBM PC, PCjr, and PC/XT (and port-compatible computers) have eight IRQ lines to the PIC chip — IRQ0 through IRQ7. These lines are mapped into interrupt vectors for Interrupts 08H through 0FH (that is, 8 + IRQ level). These eight IRQ lines and their associated interrupts are listed in Table 13-3.

Table 13-3. Eight-Level Interrupt Map.

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	I/O channel (unused on IBM PC/XT)
IRQ3	0BH	COM1 service required
IRQ4	0CH	COM2 service required
IRQ5	0DH	Fixed-disk service required
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from parallel printer*

*This request cannot be reliably generated by older versions of the IBM Monochrome/Printer Adapter and compatibles. Printer drivers that depend on this signal for operation with these cards are subject to failure.

Sixteen-level designs

In the IBM PC/AT, 8 more IRQ levels have been added by using a second 8259A PIC (the "slave") and a cascade effect, which gives 16 priority levels.

The cascade effect is accomplished by connecting the INT line of the slave to the IRQ2 line of the first, or "master," 8259A instead of to the microprocessor. When a device connected to one of the slave's IRQ lines makes an interrupt request, the INT line of the slave goes high and causes the IRQ2 line of the master 8259A to go high, which, in turn, causes the INT line of the master to go high and thus interrupts the microprocessor.

The microprocessor, ignorant of the second 8259A's presence, simply generates an interrupt acknowledge signal on receipt of the interrupt from the master 8259A. This signal initializes *both* 8259As and also causes the master to turn control over to the slave. The slave then completes the interrupt request.

On the IBM PC/AT, the eight additional IRQ lines are mapped to Interrupts 70H through 77H (Table 13-4). Because the eight additional lines are effectively connected to the master

8259A's IRQ2 line, they take priority over the master's IRQ3 through IRQ7 events. The cascade effect is graphically represented in Figure 13-4.

Table 13-4. Sixteen-Level Interrupt Map.

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	INT from slave 8259A:
IRQ8	70H	Real-time clock service
IRQ9	71H	Software redirected to IRQ2
IRQ10	72H	Reserved
IRQ11	73H	Reserved
IRQ12	74H	Reserved
IRQ13	75H	Numeric coprocessor
IRQ14	76H	Fixed-disk controller
IRQ15	77H	Reserved
IRQ3	0BH	COM2 service required
IRQ4	0CH	COM1 service required
IRQ5	0DH	Data request from LPT2
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from LPT1

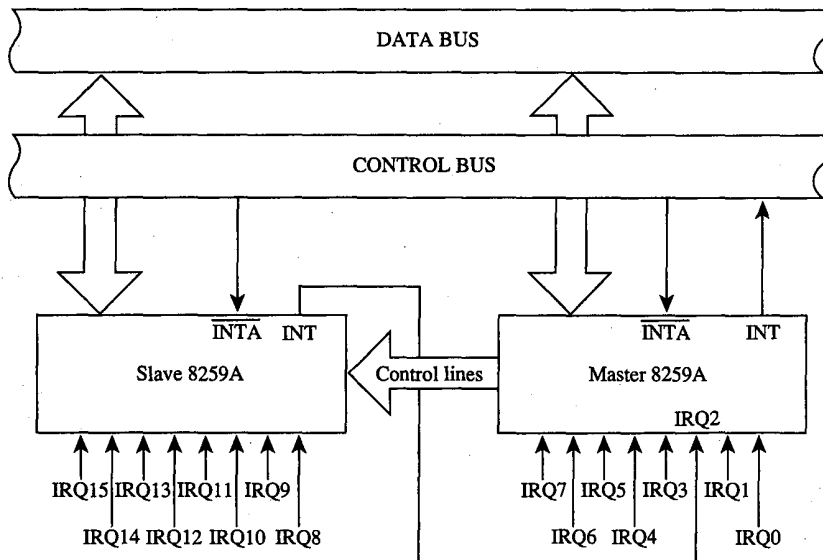


Figure 13-4. A graphic representation of the cascade effect for IRQ priorities.

Note: During the INTA sequence, the corresponding bit in the ISR register of both 8259As is set, so two EOIs must be issued to complete the interrupt service — one for the slave and one for the master.

Programming for the Hardware Interrupts

Any program that modifies an interrupt vector must restore the vector to its original condition before returning control to MS-DOS (or to its parent process). Any program that totally replaces an existing hardware interrupt handler with one of its own must perform all the handshaking and terminating actions of the original — re-enable interrupt service, signal EOI to the interrupt controller, and so forth. Failure to follow these rules has led to many hours of programmer frustration. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

When an existing interrupt handler is completely replaced with a new, customized routine, the existing vector must be saved so it can be restored later. Although it is possible to modify the 4-byte vector by directly addressing the vector table in low RAM (and many published programs have followed this practice), any program that does so runs the risk of causing system failure when the program is used with multitasking or multiuser enhancements or with future versions of MS-DOS. The only technique that can be recommended for either obtaining the existing vector values or changing them is to use the MS-DOS functions provided for this purpose: Interrupt 21H Functions 25H (Set Interrupt Vector) and 35H (Get Interrupt Vector).

After the existing vector has been saved, it can be replaced with a far pointer to the replacement routine. The new routine must end with an IRET instruction. It should also take care to preserve all microprocessor registers and conditions at entry and restore them before returning.

A sample replacement handler

Suppose a program performs many mathematical calculations of random values. To prevent abnormal termination of the program by the default MS-DOS Interrupt 00H handler when a DIV or IDIV instruction is attempted and the divisor is zero, a programmer might want to replace the Interrupt 00H (Divide by Zero) routine with one that informs the user of what has happened and then continues operation without abnormal termination. The .COM program DIVZERO.ASM (Figure 13-5) does just that. (Another example is included in the article on interrupt-driven communications. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.)

```

        name    divzero
        title   'DIVZERO - Interrupt 00H Handler'
;
; DIVZERO.ASM: Demonstration Interrupt 00H Handler
;
; To assemble, link, and convert to COM file:
;
;       C>MASM DIVZERO; <Enter>
;       C>LINK DIVZERO; <Enter>
;       C>EXE2BIN DIVZERO.EXE DIVZERO.COM <Enter>
;       C>DEL DIVZERO.EXE <Enter>
;

cr      equ     0dh           ; ASCII carriage return
lf      equ     0ah           ; ASCII linefeed
eos     equ     '$'           ; end of string marker

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT

        org    100h

entry:  jmp     start         ; skip over data area

intmsg  db      'Divide by Zero Occurred!',cr,lf,eos

divmsg  db      'Dividing '   ; message used by demo
par1    db      '0000h'       ; dividend goes here
        db      ' by '
par2    db      '00h'         ; divisor goes here
        db      ' equals '
par3    db      '00h'         ; quotient here
        db      ' remainder '
par4    db      '00h'         ; and remainder here
        db      cr,lf,eos

oldint0 dd     ?             ; save old Int 00H vector

intflag db     0             ; nonzero if divide by
                          ; zero interrupt occurred

oldip   dw     0             ; save old IP value

;
; The routine 'int0' is the actual divide by zero
; interrupt handler. It gains control whenever a
; divide by zero or overflow occurs. Its action
; is to set a flag and then increment the instruction
; pointer saved on the stack so that the failing

```

(more)

Figure 13-5. The Divide by Zero replacement handler, DIVZERO.ASM. This code is specific to 80286 and 80386 microprocessors. (See Appendix M: 8086/8088 Software Compatibility Issues.)

```

; divide will not be reexecuted after the IRET.
;
; In this particular case we can call MS-DOS to
; display a message during interrupt handling
; because the application triggers the interrupt
; intentionally. Thus, it is known that MS-DOS or
; other interrupt handlers are not in control
; at the point of interrupt.
;

int0:  pop    cs:oldip    ; capture instruction pointer

        push  ax
        push  bx
        push  cx
        push  dx
        push  di
        push  si
        push  ds
        push  es

        push  cs          ; set DS = CS
        pop   ds

        mov   ah,09h      ; print error message
        mov   dx,offset _TEXT:intmsg
        int   21h

        add   oldip,2     ; bypass instruction causing
                          ; divide by zero error

        mov   intflag,1   ; set divide by 0 flag

        pop   es          ; restore all registers
        pop   ds
        pop   si
        pop   di
        pop   dx
        pop   cx
        pop   bx
        pop   ax

        push  cs:oldip    ; restore instruction pointer

        iret              ; return from interrupt

```

```

;
; The code beginning at 'start' is the application
; program. It alters the vector for Interrupt 00H to
; point to the new handler, carries out some divide

```

Figure 13-5. Continued.

(more)

```

; operations (including one that will trigger an
; interrupt) for demonstration purposes, restores
; the original contents of the Interrupt 00H vector,
; and then terminates.
;

start: mov     ax,3500h      ; get current contents
       int     21h          ; of Int 00H vector

                               ; save segment:offset
                               ; of previous Int 00H handler
       mov     word ptr oldint0,bx
       mov     word ptr oldint0+2,es

                               ; install new handler...
       mov     dx,offset int0 ; DS:DX = handler address
       mov     ax,2500h      ; call MS-DOS to set
       int     21h          ; Int 00H vector

                               ; now our handler is active,
                               ; carry out some test divides.

       mov     ax,20h        ; test divide
       mov     bx,1          ; divide by 1
       call    divide

       mov     ax,1234h      ; test divide
       mov     bx,5eh        ; divide by 5EH
       call    divide

       mov     ax,5678h      ; test divide
       mov     bx,7fh        ; divide by 127
       call    divide

       mov     ax,20h        ; test divide
       mov     bx,0          ; divide by 0
       call    divide        ; (triggers interrupt)

                               ; demonstration complete,
                               ; restore old handler

       lds     dx,oldint0    ; DS:DX = handler address
       mov     ax,2500h      ; call MS-DOS to set
       int     21h          ; Int 00H vector

       mov     ax,4c00h      ; final exit to MS-DOS
       int     21h          ; with return code = 0

;
; The routine 'divide' carries out a trial division,
; displaying the arguments and the results. It is

```

Figure 13-5. Continued.

(more)

```

; called with AX = dividend and BL = divisor.
;
divide proc near
    push ax          ; save arguments
    push bx
    mov di,offset par1 ; convert dividend to
    call wtoa        ; ASCII for display
    mov ax,bx        ; convert divisor to
    mov di,offset par2 ; ASCII for display
    call btoa
    pop bx           ; restore arguments
    pop ax
    div bl           ; perform the division
    cmp intflag,0    ; divide by zero detected?
    jne nodiv        ; yes, skip display
    push ax          ; no, convert quotient to
    mov di,offset par3 ; ASCII for display
    call btoa
    pop ax           ; convert remainder to
    xchg ah,al       ; ASCII for display
    mov di,offset par4
    call btoa
    mov ah,09h       ; show arguments, results
    mov dx,offset divmsg
    int 21h
nodiv: mov intflag,0 ; clear divide by 0 flag
    ret              ; and return to caller
divide endp
wtoa proc near
    ; convert word to hex ASCII
    ; call with AX = binary value
    ; DI = addr for string
    ; returns AX, CX, DI destroyed
    push ax          ; save original value
    mov al,ah
    call btoa        ; convert upper byte
    add di,2         ; increment output address

```

Figure 13-5. Continued.

(more)

```

        pop     ax
        call   btoa          ; convert lower byte
        ret     ; return to caller

wtoa   endp

btoa   proc    near        ; convert byte to hex ASCII
        ; call with AL = binary value
        ;         DI = addr to store string
        ; returns AX, CX destroyed

        mov    ah,al        ; save lower nibble
        mov    cx,4         ; shift right 4 positions
        shr    al,cl        ; to get upper nibble
        call   ascii        ; convert 4 bits to ASCII
        mov    [di],al      ; store in output string
        mov    al,ah        ; get back lower nibble

        and    al,0fh       ; blank out upper one
        call   ascii        ; convert 4 bits to ASCII
        mov    [di+1],al    ; store in output string
        ret     ; back to caller

btoa   endp

ascii  proc    near        ; convert AL bits 0-3 to
        ; ASCII {0...9,A...F}
        ; and return digit in AL

        add    al,'0'       ;
        cmp    al,'9'       ;
        jle    ascii2      ;
        add    al,'A'-'9'-1 ; "fudge factor" for A-F
ascii2: ret     ; return to caller

ascii  endp

_TEXT  ends

        end    entry

```

Figure 13-5. Continued.

Supplementary handlers

In many cases, a custom interrupt handler augments, rather than replaces, the existing routine. The added routine might process some data before passing the data to the existing routine, or it might do the processing afterward. These cases require slightly different coding for the handler.

If the added routine is to process data before the existing handler does, the routine need only jump to the original handler after completing its processing. This jump can be done

indirectly, with the same pointer used to save the original content of the vector for restoration at exit. For example, a replacement Interrupt 08H handler that merely increments an internal flag at each timer tick can look something like the following:

```

myflag dw      ?                ; variable to be incremented
                                   ; on each timer-tick interrupt

oldint8 dd     ?                ; contains address of previous
                                   ; timer-tick interrupt handler

                                   ; get the previous contents
                                   ; of the Interrupt 08H vector...
mov     ax,3508h                ; AH = 35H (Get Interrupt Vector)
int     21h                     ; AL = Interrupt number (08H)
mov     word ptr oldint8,bx     ; save the address of
mov     word ptr oldint8+2,es   ; the previous Int 08H Handler
mov     dx,seg myint8          ; put address of the new
mov     ds,dx                  ; interrupt handler into DS:DX
mov     dx,offset myint8       ; and call MS-DOS to set vector
mov     ax,2508h               ; AH = 25H (Set Interrupt Vector)
int     21h                     ; AL = Interrupt number (08H)

myint8:                               ; this is the new handler
                                   ; for Interrupt 08H

inc     cs:myflag               ; increment variable on each
                                   ; timer-tick interrupt

jmp     dword ptr cs:[oldint8]    ; then chain to the
                                   ; previous interrupt handler

```

The added handler must preserve all registers and machine conditions, except those machine conditions it will modify, such as the value of *myflag* in the example (and the flags register, which is saved by the interrupt action), and it must restore those registers and conditions before performing the jump to the original handler.

A more complex situation arises when a replacement handler does some processing *after* the original routine executes, especially if the replacement handler is not reentrant. To allow for this processing, the replacement handler must prevent nested interrupts, so that even if the old handler (which is chained to the replacement handler by a CALL instruction) issues an EOI, the replacement handler will not be interrupted during postprocessing. For example, instead of using the preceding Interrupt 08H example routine, the programmer could use the following code to implement *myflag* as a semaphore and use the XCHG instruction to test it:

```

myint8:                                ; this is the new handler
                                        ; for Interrupt 08H

        mov     ax,1                    ; test and set interrupt-
xchg    cs:myflag,ax                    ; handling-in-progress semaphore

        push   ax                        ; save the semaphore

        pushf                             ; simulate interrupt, allowing
call    dword ptr cs:oldint8            ; the previous handler for the
                                        ; Interrupt 08H vector to run

        pop    ax                         ; get the semaphore back
or      ax,ax                            ; is our interrupt handler
                                        ; already running?

        jnz    myint8x                   ; yes, skip this one

        .
        .
        .
                                        ; now perform our interrupt
                                        ; processing here...

        mov    cs:myflag,0               ; clear the interrupt-handling-
                                        ; in-progress flag

myint8x:
        iret                             ; return from interrupt

```

Note that an interrupt handler of this type must simulate the original call to the interrupt routine by first doing a PUSHF, followed by a far CALL via the saved pointer to execute the original handler routine. The flags register pushed onto the stack is restored by the IRET of the original handler. Upon return from the original code, the new routine can preserve the machine state and do its own processing, finally returning to the caller by means of its own IRET.

The flags inside the new routine need not be preserved, as they are automatically restored by the IRET instruction. Because of the nature of interrupt servicing, the service routine should not depend on any information in the flags register, nor can it return any information in the flags register. Note also that the previous handler (invoked by the indirect CALL) will almost certainly have dismissed the interrupt by sending an EOI to the 8259A PIC. Thus, the machine state is not the same as in the first *myint8* example.

To remove the new vector and restore the original, the program simply replaces the new vector (in the vector table) with the saved copy. If the substituted routine is part of an application program, the original vector must be restored for every possible method of exiting from the program (including Control-Break, Control-C, and critical-error *Abort* exits). Failure to observe this requirement invariably results in system failure. Even though the system failure might be delayed for some time after the exit from the offending program, when some subsequent program overlays the interrupt handler code the crash will be imminent.

Summary

Hardware interrupt handler routines, although not strictly a part of MS-DOS, form an integral part of many MS-DOS programs and are tightly constrained by MS-DOS requirements. Routines of this type play important roles in the functioning of the IBM personal computers, and, with proper design and programming, significantly enhance product reliability and performance. In some instances, no other practical method exists for meeting performance requirements.

*Jim Kyle
Chip Rabinowitz*

Article 14

Writing MS-DOS Filters

A filter is, essentially, a program that operates on a stream of characters. The source and destination of the character stream can be files, another program, or almost any character device. The transformation applied by the filter to the character stream can range from an operation as simple as substituting a character set to an operation as elaborate as generating splines from sets of coordinates.

The standard MS-DOS package includes three simple filters: SORT, which alphabetically sorts text on a line-by-line basis; FIND, which searches a text stream to match a specified string; and MORE, which displays text one screenful at a time. This article describes how filters work and how new ones can be constructed. *See also* USER COMMANDS: FIND; MORE; SORT.

System Support for Filters

The operation of a filter program relies on two features that appeared in MS-DOS version 2.0: standard devices and redirectable I/O.

The standard devices are represented by five handles that are originally established when the system is initialized. Each process inherits these handles from its immediate parent. Thus, the standard device handles are already opened when a process acquires control of the system, and the process can use the handles with Interrupt 21H Functions 3FH and 40H for read and write operations without further preliminaries. The default assignments of the standard device handles are

Handle	Name	Default Device
0	<i>stdin</i> (standard input)	CON
1	<i>stdout</i> (standard output)	CON
2	<i>stderr</i> (standard error)	CON
3	<i>stdaux</i> (standard auxiliary)	AUX
4	<i>stdlst</i> (standard list)	PRN

The CON device is assigned by default to the system's keyboard and video display. AUX is assigned by default to COM1 (the first physical serial port), and PRN is assigned by default to LPT1 (the first physical parallel printer port); in some systems these assignments can be altered with the MODE command. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output; USER COMMANDS: MODE; CTTY.

When a program is executed by entering its name at the system (COMMAND.COM) prompt, the user can redirect either or both of the standard input and standard output handles from their default device (CON) to another file, a character device, or a process. This redirection is accomplished by including one of the special characters <, >, >>, or | in the command line, in the following form:

Redirection	Result
< <i>file</i>	Contents of the specified <i>file</i> are used instead of the keyboard as the program's standard input.
< <i>device</i>	Program takes its standard input from the named <i>device</i> instead of from the keyboard.
> <i>device</i>	Program sends its standard output to the named <i>device</i> instead of to the video display.
> <i>file</i>	Program sends its standard output to the specified <i>file</i> instead of to the video display.
>> <i>file</i>	Program appends its standard output to the current contents of the specified <i>file</i> instead of to the video display.
<i>p1</i> <i>p2</i>	Standard output of program <i>p1</i> is routed to become the standard input of program <i>p2</i> (output of <i>p1</i> is said to be piped to <i>p2</i>).

For example, the command

```
C>SORT < MYFILE.TXT > PRN <Enter>
```

causes the SORT filter to read its input from the file MYFILE.TXT, sort the lines alphabetically, and write resulting text to the character device PRN (the logical name for the system's list device).

The redirection requested by the <, >, >>, or | characters takes place at the level of COMMAND.COM and is invisible to the program it affects. Such redirection can also be put into effect by another process. See Using a Filter as a Child Process below.

Note that if a program "goes around" MS-DOS to perform its input and output, either by calling ROM BIOS functions or by manipulating the keyboard or video controller directly, redirection commands placed in the program's command line do not have the expected effect.

How Filters Work

By convention, a filter program reads its text from standard input and writes the results of its operations to standard output. When the end of the input stream is reached, the filter simply terminates, optionally writing an end-of-file mark (IAH) to the output stream. As a result, filters are both flexible and simple.

Filter programs are flexible because they do not know, and do not care, about the source of the data they process or the destination of their output. Any redirection that the user

specifies in the command line is invisible to the filter. Thus, any character device that has a logical name within the system (CON, AUX, COM1, COM2, PRN, LPT1, LPT2, LPT3, and so on), any file on any block device (local or network) known to the system, or any other program can supply a filter's input or accept its output. If necessary, several functionally simple filters can be concatenated with pipes to perform very complex operations.

Although flexible, filters are also simple because they rely on their parent process to supply standard input and standard output handles that have already been appropriately redirected. The parent is responsible for opening or creating any necessary files, checking the validity of logical character device names, and loading and executing the preceding or following process in a pipe. The filter need only concern itself with the transformation it will apply to the data; it can leave the I/O details to the operating system and to its parent.

Building a Filter

Creating a new filter for MS-DOS is a straightforward process. In its simplest form, a filter need only use the handle-oriented read (Interrupt 21H Function 3FH) and write (Interrupt 21H Function 40H) functions to get characters or lines from standard input and send them to standard output, performing any desired alterations on the text stream on a character-by-character or line-by-line basis.

Figures 14-1 through 14-4 contain template character-oriented and line-oriented filters in both assembly language and C. The C version of the character filter runs much faster than the assembly-language version, because the C run-time library provides hidden blocking and deblocking (buffering) of character reads and writes; the assembly-language program actually makes two calls to MS-DOS for each character processed. (Of course, if buffering is added to the assembly-language version it will be both faster and smaller than the C filter.) The C and assembly-language versions of the line-oriented filter run at roughly the same speed.

```

name      protoc
title     'PROTOC.ASM --- template character filter'
;
; PROTOC.ASM: a template for a character-oriented filter.
;
; Ray Duncan, June 1987
;

stdin    equ    0           ; standard input
stdout   equ    1           ; standard output
stderr   equ    2           ; standard error

cr       equ    0dh        ; ASCII carriage return
lf       equ    0ah        ; ASCII linefeed

```

Figure 14-1. Assembly-language template for a character-oriented filter (file PROTOC.ASM).

(more)

```

DGROUP group   _DATA,STACK   ; 'automatic data group'

_TEXT segment byte public 'CODE'

        assume cs:_TEXT,ds:DGROUP,ss:STACK

main    proc    far           ; entry point from MS-DOS

        mov    ax,DGROUP     ; set DS=our data segment
        mov    ds,ax

main1:

        ; read a character from standard input
        mov    dx,offset DGROUP:char ; address to place character
        mov    cx,1          ; length to read = 1
        mov    bx,stdin      ; handle for standard input
        mov    ah,3fh        ; function 3FH = read from file or device
        int    21h          ; transfer to MS-DOS
        jc     main3         ; error, terminate
        cmp    ax,1         ; any character read?
        jne    main2        ; end of file, terminate program

        call   transl        ; translate character if necessary

        ; now write character to standard output
        mov    dx,offset DGROUP:char ; address of character
        mov    cx,1          ; length to write = 1
        mov    bx,stdout     ; handle for standard output
        mov    ah,40h        ; function 40H = write to file or device
        int    21h          ; transfer to MS-DOS
        jc     main3         ; error, terminate
        cmp    ax,1         ; was character written?
        jne    main3        ; disk full, terminate program
        jmp    main1        ; go process another character

main2:  mov    ax,4c00h      ; end of file reached, terminate
        int    21h          ; program with return code = 0

main3:  mov    ax,4c01h      ; error or disk full, terminate
        int    21h          ; program with return code = 1

main    endp                ; end of main procedure

;
; Perform any necessary translation on character from input,
; stored in 'char'. Template action: leave character unchanged.
;
translt proc    near

        ret                 ; template action: do nothing

translt endp

```

Figure 14-1. Continued.

(more)


```

_TEXT ends

_DATA segment word public 'DATA'
char db 0 ; temporary storage for input character
_DATA ends

STACK segment para stack 'STACK'
dw 64 dup (?)

STACK ends

end main ; defines program entry point

```

Figure 14-1. Continued.

```

/*
   PROTOC.C: a template for a character-oriented filter.

   Ray Duncan, June 1987
*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    char ch;

    while ( (ch=getchar())!=EOF ) /* read a character */
    {
        ch=translate(ch); /* translate it if necessary */
        putchar(ch); /* write the character */
    }
    exit(0); /* terminate at end of file */
}

/*
   Perform any necessary translation on character from
   input file. Template action just returns same character.
*/

int translate(ch)
char ch;
{
    return (ch);
}

```

Figure 14-2. C template for a character-oriented filter (file PROTOC.C).

```

        name    protol
        title   'PROTOL.ASM --- template line filter'
;
; PROTOL.ASM: a template for a line-oriented filter.
;
; Ray Duncan, June 1987
;

stdinp equ    0      ; standard input
stdout equ    1      ; standard output
stderr equ    2      ; standard error

cr      equ    0dh    ; ASCII carriage return
lf      equ    0ah    ; ASCII linefeed

DGROUP group  _DATA,STACK ; 'automatic data group'

_TEXT segment byte public 'CODE'

        assume  cs:_TEXT,ds:DGROUP,es:DGROUP,ss:STACK

main    proc    far          ; entry point from MS-DOS

        mov    ax,DGROUP    ; set DS = ES = our data segment
        mov    ds,ax
        mov    es,ax

main1:                                ; read a line from standard input
        mov    dx,offset DGROUP:input ; address to place data
        mov    cx,256        ; max length to read = 256
        mov    bx,stdin      ; handle for standard input
        mov    ah,3fh        ; function 3FH = read from file or device
        int    21h          ; transfer to MS-DOS
        jc    main3          ; if error, terminate
        or     ax,ax         ; any characters read?
        jz    main2          ; end of file, terminate program

        call   transl        ; translate line if necessary
        or     ax,ax         ; anything to output after translation?
        jz    main1          ; no, get next line

                                ; now write line to standard output
        mov    dx,offset DGROUP:output ; address of data
        mov    cx,ax         ; length to write
        mov    bx,stdout     ; handle for standard output
        mov    ah,40h        ; function 40H = write to file or device
        int    21h          ; transfer to MS-DOS
        jc    main3          ; if error, terminate

```

Figure 14-3. Assembly-language template for a line-oriented filter (file PROTOL.ASM).

(more)

```

        cmp     ax,cx           ; was entire line written?
        jne     main3          ; disk full, terminate program
        jmp     main1          ; go process another line

main2:  mov     ax,4c00h        ; end of file reached, terminate
        int     21h            ; program with return code = 0

main3:  mov     ax,4c01h        ; error or disk full, terminate
        int     21h            ; program with return code = 1

main    endp                  ; end of main procedure

;
; Perform any necessary translation on line stored in
; 'input' buffer, leaving result in 'output' buffer.
;
; Call with:   AX = length of data in 'input' buffer.
;
; Return:     AX = length to write to standard output.
;
; Action of template routine is just to copy the line.
;
translt proc    near

                                ; just copy line from input to output
        mov     si,offset DGROUP:input
        mov     di,offset DGROUP:output
        mov     cx,ax
        rep movsb
        ret                                ; return length in AX unchanged

translt endp

_TEXT   ends

_DATA  segment word public 'DATA'

input  db     256 dup (?)   ; storage for input line
output db     256 dup (?)   ; storage for output line

_DATA  ends

STACK  segment para stack 'STACK'

        dw     64 dup (?)

STACK  ends

        end     main        ; defines program entry point

```

Figure 14-3. Continued.

```

/*
    PROTOL.C: a template for a line-oriented filter.

    Ray Duncan, June 1987.
*/

#include <stdio.h>

static char input[256];          /* buffer for input line */
static char output[256];        /* buffer for output line */

main(argc,argv)
int argc;
char *argv[];
{
    while( gets(input) != NULL ) /* get a line from input stream */
        /* perform any necessary translation
           and possibly write result */
        {
            if (translate()) puts(output);
        }
    exit(0);                      /* terminate at end of file */
}

/*
    Perform any necessary translation on input line, leaving
    the resulting text in output buffer.  Value of function
    is 'true' if output buffer should be written to standard output
    by main routine, 'false' if nothing should be written.
*/

translate()
{
    strcpy(output,input);          /* template action is copy input */
    return(1);                    /* line and return true flag */
}

```

Figure 14-4. C template for a line-oriented filter (file PROTOL.C).

Each of the four template filters can be assembled or compiled, linked, and run exactly as they are shown in Figures 14-1 through 14-4. Of course, in this form they function like an incredibly slow COPY command.

To obtain a filter that does something useful, a routine that performs some modification of the text stream that is flowing by must be inserted between the reads and writes. For example, Figures 14-5 and 14-6 contain the assembly-language and C source code for a character-oriented filter named LC. This program converts all uppercase input characters (A-Z) to lowercase (a-z) output, leaving other characters unchanged. The only difference between LC and the template character filter is the translation subroutine that operates on the text stream.

```

    name    lc
    title   'LC.ASM --- lowercase filter'
;
; LC.ASM:   a simple character-oriented filter to translate
;           all uppercase (A-Z) to lowercase (a-z).
;
; Ray Duncan, June 1987
;

stdin equ 0           ; standard input
stdout equ 1          ; standard output
stderr equ 2          ; standard error

cr equ 0dh            ; ASCII carriage return
lf equ 0ah            ; ASCII linefeed

DGROUP group _DATA,STACK ; 'automatic data group'

_TEXT segment byte public 'CODE'

    assume cs:_TEXT,ds:DGROUP,ss:STACK

main proc far          ; entry point from MS-DOS

    mov ax,DGROUP      ; set DS = our data segment
    mov ds,ax

main1:                  ; read a character from standard input
    mov dx,offset DGROUP:char ; address to place character
    mov cx,1           ; length to read = 1
    mov bx,stdin       ; handle for standard input
    mov ah,3fh         ; function 3FH = read from file or device
    int 21h            ; transfer to MS-DOS
    jc main3           ; error, terminate
    cmp ax,1           ; any character read?
    jne main2          ; end of file, terminate program

    call translt       ; translate character if necessary

                        ; now write character to standard output
    mov dx,offset DGROUP:char ; address of character
    mov cx,1           ; length to write = 1
    mov bx,stdout      ; handle for standard output
    mov ah,40h         ; function 40H = write to file or device
    int 21h            ; transfer to MS-DOS
    jc main3           ; error, terminate
    cmp ax,1           ; was character written?
    jne main3          ; disk full, terminate program
    jmp main1          ; go process another character

```

Figure 14-5. Assembly-language source code for the LC filter (file LC.ASM).

(more)

```

main2: mov    ax,4c00h    ; end of file reached, terminate
       int    21h        ; program with return code = 0

main3: mov    ax,4c01h    ; error or disk full, terminate
       int    21h        ; program with return code = 1

main   endp            ; end of main procedure

;
; Translate uppercase (A-Z) characters to corresponding
; lowercase characters (a-z). Leave other characters unchanged.
;
translt proc    near

       cmp    byte ptr char,'A'
       jb     transx
       cmp    byte ptr char,'Z'
       ja     transx
       add    byte ptr char,'a'-'A'
transx: ret

translt endp

_TEXT  ends

_DATA  segment word public 'DATA'
char   db     0          ; temporary storage for input character
_DATA  ends

STACK  segment para stack 'STACK'
       dw     64 dup (?)
STACK  ends

       end    main      ; defines program entry point

```

Figure 14-5. Continued.

```

/*
LC:    a simple character-oriented filter to translate
       all uppercase (A-Z) to lowercase (a-z) characters.

Usage: LC [< source] [> destination]

```

Figure 14-6. C source code for the LC filter (file LC.C).

(more)

```

Ray Duncan, June 1987

*/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    char ch;

    while ( (ch=getchar() ) != EOF )
    {
        ch=translate(ch);    /* perform any necessary
                             character translation */
        putchar(ch);        /* then write character */
    }
    exit(0);                /* terminate at end of file */
}

/*
   Translate characters A-Z to lowercase equivalents
*/

int translate(ch)
char ch;
{
    if (ch >= 'A' && ch <= 'Z') ch += 'a'-'A';
    return (ch);
}

```

Figure 14-6. Continued.

As another example, Figure 14-7 contains the C source code for a line-oriented filter called FIND. This simple filter is invoked with a command line in the form

FIND "pattern" < source > destination

FIND searches the input stream for lines containing the pattern specified in the command line. The line number and text of any line containing a match is sent to standard output, with any tabs expanded to eight-column tab stops.

```

/*
   FIND.C           Searches text stream for a string.

   Usage:          FIND "pattern" [< source] [> destination]

   by Ray Duncan, June 1987
*/

#include <stdio.h>

```

Figure 14-7. C source code for a new FIND filter (file FIND.C).

(more)

```

#define TAB      '\x09'          /* ASCII tab character (^I) */
#define BLANK    '\x20'          /* ASCII space character */

#define TAB_WIDTH 8              /* columns per tab stop */

static char input[256];         /* buffer for line from input */
static char output[256];        /* buffer for line to output */
static char pattern[256];       /* buffer for search pattern */

main(argc,argv)
int argc;
char *argv[];
{
    int line=0;                  /* initialize line variable */

    if ( argc < 2 )             /* was search pattern supplied? */
    {
        puts("find: missing pattern.");
        exit(1);                 /* abort if not */
    }
    strcpy(pattern,argv[1]);     /* save copy of string to find */
   strupr(pattern);             /* fold it to uppercase */
    while( gets(input) != NULL ) /* read a line from input */
    {
        line++;                  /* count lines */
        strcpy(output,input);    /* save copy of input string */
       strupr(input);           /* fold input to uppercase */
        /* if line contains pattern */
        if( strstr(input,pattern) )
            /* write it to standard output */
            writeline(line,output);
    }
    exit(0);                     /* terminate at end of file */
}

/*
WRITELINE: Write line number and text to standard output,
expanding any tab characters to stops defined by TAB_WIDTH.
*/

writeline(line,p)
int line;
char *p;
{
    int i=0;                     /* index to original line text */
    int col=0;                   /* actual output column counter */
    printf("\n%4d: ",line);      /* write line number */
    while( p[i]!=NULL )         /* while end of line not reached */
    {
        if(p[i]==TAB)           /* if current char = tab, expand it */
        {
            do putchar(BLANK);
            while(++col % TAB_WIDTH != 0);
        }
        else                     /* otherwise just send character */
        {
            putchar(p[i]);
            col++;                /* count columns */
        }
    }
}

```

Figure 14-7. Continued.

(more)


```

        i++;                               /* advance through output line */
    }
}

```

Figure 14-7. Continued.

This sample FIND filter differs from the FIND filter supplied by Microsoft with MS-DOS in several respects. It is not case sensitive, so the pattern "foobar" will match "FOOBAR", "FooBar", and so forth. Second, this filter supports no switches; these are left as an exercise for the reader. Third, unlike the Microsoft version of FIND, this program always reads from standard input; it is not able to open its own files.

Using a Filter as a Child Process

Instead of incorporating all the code necessary to do the job itself, an application program can load and execute a filter as a child process to carry out a specific task. Before the child filter is loaded, the parent must arrange for the standard input and standard output handles that will be inherited by the child to be attached to the files or character devices that will supply the filter's input and receive its output. This redirection is accomplished with the following steps using Interrupt 21H functions:

1. The parent process uses Function 45H (Duplicate File Handle) to create duplicates of its standard input and standard output handles and then saves the duplicates.
2. The parent opens (with Function 3DH) or creates (with Function 3CH) the files or devices that the child process will use for input and output.
3. The parent uses Function 46H (Force Duplicate File Handle) to force its own standard device handles to track the new file or device handles acquired in step 2.
4. The parent uses Function 4B00H (Load and Execute Program [EXEC]) to load and execute the child process. The child inherits the redirected standard input and standard output handles and uses them to do its work. The parent regains control after the child filter terminates.
5. The parent uses the duplicate handles created in step 1, together with Function 46H (Force Duplicate File Handle), to restore its own standard input and standard output handles to their original meanings.
6. The parent closes (with Function 3EH) the duplicate handles created in step 1, because they are no longer needed.

It might seem as though the parent process could just as easily close its own standard input and standard output (handles 0 and 1), open the input and output files needed by the child, load and execute the child, close the files upon regaining control, and then reopen the CON device twice. Because the open operation always assigns the first free handle, this approach would have the desired effect as far as the child process is concerned. However, it would throw away any redirection that had been established for the parent process by its parent. Thus, the need to preserve any preexisting redirection of the parent's standard

input and standard output, along with the desire to preserve the parent's usual output channel for informational messages right up to the actual point of the EXEC call, is the reason for the elaborate procedure outlined above in steps 1 through 6.

The program EXECSORT.ASM in Figure 14-8 demonstrates this redirection of input and output for a filter run as a child process. The parent, which is called EXECSORT, saves duplicates of its current standard input and standard output handles and then redirects those handles respectively to the files MYFILE.DAT (which it opens) and MYFILE.SRT (which it creates). EXECSORT then uses Interrupt 21H Function 4BH (EXEC) to run the SORT.EXE filter that is supplied with MS-DOS (this file must be in the current drive and directory for the demonstration to work correctly).

```

name      execsort
title     'EXECSORT --- demonstrate EXEC of filter'
.sall

;
; EXECSORT.ASM --- demonstration of use of EXEC to run the SORT
; filter as a child process, redirecting its input and output.
; This program requires the files SORT.EXE and MYFILE.DAT in
; the current drive and directory.
;
; Ray Duncan, June 1987
;

stdin    equ    0                ; standard input
stdout   equ    1                ; standard output
stderr   equ    2                ; standard error

stksize  equ    128              ; size of stack

cr       equ    0dh              ; ASCII carriage return
lf       equ    0ah              ; ASCII linefeed

jerr     macro target            ;; Macro to test carry flag
        local  notset           ;; and jump if flag set.
        jnc   notset            ;; Uses JMP DISP16 to avoid
        jmp   target            ;; branch out of range errors
notset:
        endm

DGROUP  group  _DATA,_STACK      ; 'automatic data group'

```

(more)

Figure 14-8. Assembly-language source code demonstrating use of a filter as a child process. This code redirects the standard input and standard output handles to files, invokes the EXEC function (Interrupt 21H Function 4BH) to run the SORT.EXE program, and then restores the original meaning of the standard input and standard output handles (file EXECSORT.ASM).

```

_TEXT segment byte public 'CODE'      ; executable code segment

assume cs:_TEXT,ds:DGROUP,ss:_STACK

stk_seg dw      ?                      ; original SS contents
stk_ptr dw      ?                      ; original SP contents

main proc far                          ; entry point from MS-DOS

    mov ax,DGROUP                      ; set DS = our data segment
    mov ds,ax

    ; now give back extra memory so
    ; child SORT has somewhere to run...
    mov ax,es                          ; let AX = segment of PSP base
    mov bx,ss                          ; and BX = segment of stack base
    sub bx,ax                          ; reserve seg stack - seg psp
    add bx,stksize/16                 ; plus paragraphs of stack
    mov ah,4ah                        ; fxn 4AH = modify memory block
    int 21h                          ; transfer to MS-DOS
    jerr main1                        ; jump if resize block failed

    ; prepare stdin and stdout
    ; handles for child SORT process

    mov bx,stdin                      ; dup the handle for stdin
    mov ah,45h
    int 21h                          ; transfer to MS-DOS
    jerr main1                        ; jump if dup failed
    mov oldin,ax                      ; save dup'd handle

    mov dx,offset DGROUP:infile      ; now open the input file
    mov ax,3d00h                    ; mode = read-only
    int 21h                          ; transfer to MS-DOS
    jerr main1                        ; jump if open failed

    mov bx,ax                        ; force stdin handle to
    mov cx,stdin                    ; track the input file handle
    mov ah,46h
    int 21h                          ; transfer to MS-DOS
    jerr main1                        ; jump if force dup failed

    mov bx,stdout                   ; dup the handle for stdout
    mov ah,45h
    int 21h                          ; transfer to MS-DOS
    jerr main1                        ; jump if dup failed
    mov oldout,ax                   ; save dup'd handle

    mov dx,offset DGROUP:outfile    ; now create the output file

```

Figure 14-8. Continued.

(more)

```

mov     cx,0             ; normal attribute
mov     ah,3ch
int     21h             ; transfer to MS-DOS
jerr    main1          ; jump if create failed

mov     bx,ax           ; force stdout handle to
mov     cx,stdout       ; track the output file handle
mov     ah,46h
int     21h             ; transfer to MS-DOS
jerr    main1          ; jump if force dup failed

; now EXEC the child SORT,
; which will inherit redirected
; stdin and stdout handles

push    ds              ; save EXEC SORT's data segment
mov     stk_seg,ss      ; save EXEC SORT's stack pointer
mov     stk_ptr,sp

mov     ax,ds           ; set ES = DS
mov     es,ax
mov     dx,offset DGROUP:cname ; DS:DX = child pathname
mov     bx,offset DGROUP:pars  ; EX:BX = parameter block
mov     ax,4b00h        ; function 4BH, subfunction 00H
int     21h             ; transfer to MS-DOS

cli                                           ; (for bug in some early 8088s)
mov     ss,stk_seg     ; restore execsort's stack pointer
mov     sp,stk_ptr
sti                                           ; (for bug in some early 8088s)
pop     ds              ; restore DS = our data segment

jerr    main1          ; jump if EXEC failed

mov     bx,oldin       ; restore original meaning of
mov     cx,stdin       ; standard input handle for
mov     ah,46h         ; this process
int     21h
jerr    main1          ; jump if force dup failed

mov     bx,oldout      ; restore original meaning
mov     cx,stdout      ; of standard output handle
mov     ah,46h         ; for this process
int     21h
jerr    main1          ; jump if force dup failed

mov     bx,oldin       ; close dup'd handle of
mov     ah,3eh         ; original stdin
int     21h           ; transfer to MS-DOS

```

Figure 14-8. Continued.

(more)

```

        jerr    main1                ; jump if close failed

        mov     bx,oldout            ; close dup'd handle of
        mov     ah,3eh              ; original stdout
        int     21h                 ; transfer to MS-DOS
        jerr    main1                ; jump if close failed

                                        ; display success message
        mov     dx,offset DGROUP:msg1 ; address of message
        mov     cx,msg1_len         ; message length
        mov     bx,stdout           ; handle for standard output
        mov     ah,40h              ; fxn 40H = write file or device
        int     21h                 ; transfer to MS-DOS
        jerr    main1

        mov     ax,4c00h            ; no error, terminate program
        int     21h                 ; with return code = 0

main1:  mov     ax,4c01h            ; error, terminate program
        int     21h                 ; with return code = 1

main    endp                        ; end of main procedure

_TEXT  ends

_DATA  segment para public 'DATA'   ; static & variable data segment

infile db    'MYFILE.DAT',0        ; input file for SORT filter
outfile db   'MYFILE.SRT',0        ; output file for SORT filter

oldin  dw    ?                      ; dup of old stdin handle
oldout dw    ?                      ; dup of old stdout handle

cname  db    'SORT.EXE',0          ; pathname of child SORT process

pars   dw    0                      ; segment of environment block
                                        ; (0 = inherit parent's)
        dd    tail                  ; long address, command tail
        dd    -1                    ; long address, default FCB #1
                                        ; (-1 = none supplied)
        dd    -1                    ; long address, default FCB #2
                                        ; (-1 = none supplied)

tail   db    0,cr                  ; empty command tail for child

msg1   db    cr,lf,'SORT was executed as child.',cr,lf
msg1_len equ $-msg1

_DATA  ends

```

Figure 14-8. Continued.

(more)

```
_STACK segment para stack 'STACK'
        db      stksize dup (?)
_STACK ends

        end      main                ; defines program entry point
```

Figure 14-8. Continued.

The MS-DOS SORT program reads the file MYFILE.DAT via its standard input handle, sorts the file alphabetically, and writes the sorted data to MYFILE.SRT via its standard output handle. When SORT terminates, MS-DOS closes SORT's inherited handles for standard input and standard output, which forces an update of the directory entries for the associated files. The program EXEC SORT then resumes execution, restores its own standard input and standard output handles (which are still open) to their original meanings, displays a success message on standard output, and exits to MS-DOS.

Ray Duncan

Article 15

Installable Device Drivers

The software that runs on modern computer systems is, by convention, organized into layers with varied degrees of independence from the underlying computer hardware. The purpose of this layering is threefold:

- To minimize the impact on programs of differences between hardware devices or changes in the hardware.
- To allow the code for common operations to be centralized and optimized.
- To ease the task of moving programs and their data from one machine to another.

The top and most hardware-independent layer is usually the transient, or application, program, which performs a specific job and deals with data in terms of files and records within those files. Such programs are called transient because they are brought into RAM for execution when needed and are discarded from memory when their job is finished. Examples of such programs are Microsoft Word, various programming tools such as the Microsoft Macro Assembler (MASM) and the Microsoft Object Linker (LINK), and even some of the standard MS-DOS utility programs such as CHKDSK and FORMAT.

The middle layer is the operating-system kernel, which manages the allocation of system resources such as memory and disk storage, provides a battery of services to application programs, and implements disk directories and the other housekeeping details of disk storage. The MS-DOS kernel is brought into memory from the file MSDOS.SYS (or IBMDOS.COM with PC-DOS) when the system is turned on or restarted and remains fixed in memory until the system is turned off. The system's default command processor, COMMAND.COM, and system manager programs such as Microsoft Windows bridge the categories of application program and operating system: Parts of them remain resident in memory at all times, but they rely on the MS-DOS kernel for services such as file I/O. *See PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: Components of MS-DOS.*

The modules in the lowest layer are called device drivers. These drivers are the components of the operating system that manage the controller, or adapter, of a peripheral device—a piece of hardware that the computer uses for such purposes as storage or communicating with the outside world. Thus, device drivers are responsible for transferring data between a peripheral device and the computer's RAM memory, where other programs can work on it. Drivers shield the operating-system kernel from the need to deal with hardware I/O port addresses, operating characteristics, and the peculiarities of a particular peripheral device, just as the kernel, in turn, shields application programs from the details of file management.

In MS-DOS versions 1.x, device drivers were integrated into the operating system and could be extended or replaced only by patching the files that contained the operating system itself. Because every third-party peripheral manufacturer evolved a different method of modifying these files to get its product to work, conflicts between products from different manufacturers were frequent and expansion of a PC with new disk drives and other devices (especially fixed disks) was often a chancy proposition.

In MS-DOS versions 2.0 and later, there is a clean separation between device drivers and the MS-DOS kernel. Device drivers have a straightforward structure and are interfaced to the kernel through a simple and clearly defined scheme that consists of far calls, function codes, and data packets. Given adequate information about the hardware, a programmer can write a new device driver that follows this structure and interface for almost any conceivable peripheral device; such a driver can subsequently be installed and used without any changes to the underlying operating system.

This article explains the anatomy, operation, and creation of drivers for MS-DOS versions 2.0 and later. Device drivers for versions 1.x are not discussed further here.

Resident and Installable Drivers

Every MS-DOS system contains built-in device drivers for the console (keyboard and video display), the serial port, the parallel printer port, the real-time clock, and at least one disk storage device (the system boot device). These drivers, known as the resident drivers, are loaded as a set from the file IO.SYS (or IBMBIO.COM with PC-DOS) when the system is turned on or restarted.

Drivers for additional peripheral devices occupy individual files on the disk. These drivers, called installable drivers, are loaded and linked into the system during its initialization as a result of DEVICE directives in the CONFIG.SYS file. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: Components of MS-DOS. Examples of such drivers are the ANSI.SYS and RAMDISK.SYS files included with MS-DOS version 3.2. In all other respects, installable drivers have the same structure and relationship to the MS-DOS kernel as the resident drivers. All drivers in the system are chained together so that MS-DOS can rapidly search the entire set to find a specific block or character device when an I/O operation is requested.

Device drivers as a whole are categorized into two groups: block-device drivers and character-device drivers. A driver's membership in one of these two groups determines how the associated device is viewed by MS-DOS and what functions the driver itself must support.

Character-device drivers

Character-device drivers control peripheral devices, such as a terminal or a printer, that perform input and output one character (or byte) at a time. Each character-device driver

ordinarily supports a single hardware unit. The device has a one-character to eight-character logical name that can be used by an application program to "open" the device for input or output as though it were a file. The logical name is strictly a means of identifying the driver to MS-DOS and has no physical equivalent on the device (unlike a volume label for block devices).

The three resident character-device drivers for the console, serial port, and printer carry the logical device names CON, AUX, and PRN, respectively. These three drivers receive special treatment by MS-DOS that allows application programs to address the associated devices in three different ways:

- They can be opened by name for input and output (like any other character device).
- They are supported by special-purpose MS-DOS function calls (Interrupt 21H Functions 01-0CH).
- They are assigned to default handles (standard input, standard output, standard error, standard auxiliary, and standard list) that need not be opened to be used.

See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Character Device Input and Output.

Other character devices can be supported by simply installing additional character-device drivers. The only significant restriction on the total number of devices that can be supported, other than the memory required to hold the drivers, is that each driver must have a unique logical name. When MS-DOS receives an open request for a character device, it searches the chain of device drivers in order from the last driver loaded to the first. Thus, if more than one driver uses the same logical name, the last driver to be loaded supersedes any others and receives all I/O requests addressed to that logical name. This behavior can be used to advantage in some situations. For example, it allows the more powerful ANSI.SYS display driver to supersede the system's default console driver, which does not support cursor positioning and character attributes.

The MS-DOS kernel's buffering and filtering of the characters that pass between it and a character-device driver are affected by whether MS-DOS regards the device to be in cooked mode or raw mode. During cooked mode input, MS-DOS requests characters one at a time from the driver and places them in its own internal buffer, echoing each character to the screen (if the input device is the keyboard) and checking each character for a Control-C (03H) or a Return (0DH). When either the number of characters requested by the application program has been received or a Return is detected, the input is terminated and the data is copied from MS-DOS's internal buffer into the requesting program's buffer. When a Control-C is detected, MS-DOS aborts the input operation and transfers to the routine whose address is stored in the Interrupt 23H (Control-C Handler Address) vector. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers. Similarly, during output in cooked mode, MS-DOS checks between each character for a Control-C pending at the keyboard and aborts the output operation if one is detected.

In raw mode, the exact number of bytes requested by the application program is read or written, without regard to any control characters such as Return or Control-C. MS-DOS passes the entire I/O request to the driver in a single operation, instead of breaking the request into single-character reads or writes, and the characters are transferred directly to or from the requesting program's buffer.

The mode for a specific device can be queried by an application program with the IOCTL Get Device Data function (Interrupt 21H Function 44H Subfunction 00H); the mode can be selected with the Set Device Data function (Interrupt 21H Function 44H Subfunction 01H). See SYSTEM CALLS: INTERRUPT 21H: Function 44H. The driver itself is not usually aware of its mode and the mode does not affect its operation.

Block-Device Drivers

Block-device drivers control peripheral devices that transfer data in chunks rather than 1 byte at a time. Block devices are usually randomly addressable devices such as floppy- or fixed-disk drives, but they can also be sequential devices such as magnetic-tape drives. A block driver can support more than one physical unit and can also map two or more logical units onto a single physical unit, as with a partitioned fixed disk.

MS-DOS assigns single-letter drive identifiers (A, B, and so forth) to block devices, instead of logical names. The first letter assigned to a block-device driver is determined solely by the driver's position in the chain of all drivers — that is, by the number of units supported by the block drivers loaded before it; the total number of letters assigned to the driver is determined by the number of logical drive units the driver supports.

MS-DOS does not associate a mode (cooked or raw) with block-device drivers. A block-device driver always reads or writes exactly the number of sectors requested (barring hardware or addressing errors) and never filters or otherwise manipulates the contents of the blocks being transferred.

Structure of an MS-DOS Device Driver

A device driver has three major components (Figure 15-1):

- The device header
- The Strategy routine (*Strat*)
- The Interrupt routine (*Intr*)

The device header

The device header (Figure 15-2) always lies at the beginning of the driver. It contains a link to the next driver in the chain, a word (16 bits) of device attribute flags, offsets to the executable Strategy and Interrupt routines for the device, and the logical device name if it is a character device such as PRN or COM1 or the number of logical units if it is a block device.

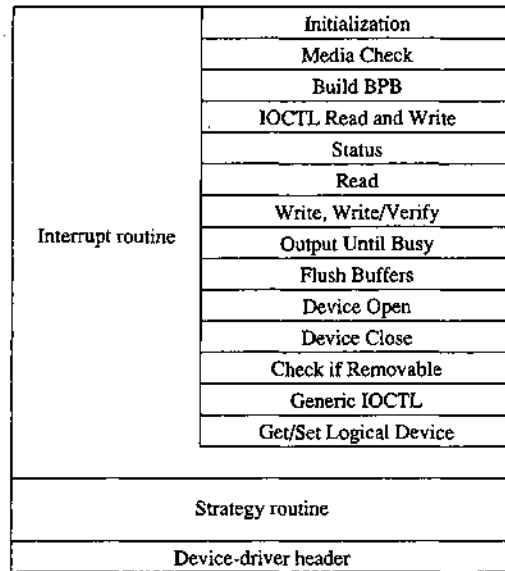


Figure 15-1. General structure of an MS-DOS installable device driver.

Offset	
00H	Link to next driver, offset
02H	Link to next driver, segment
04H	Device attribute word
06H	Offset, Strategy entry point
08H	Offset, Interrupt entry point
0AH	Logical name (8 bytes) if character device or Number of units (1 byte) followed by 7 bytes of reserved space if block device
12H	

Figure 15-2. Device header. The offsets to the Strat and Intr routines are offsets from the same segment used to point to the device header.

The device attribute flags word (Table 15-1) defines whether a driver controls a character or a block device, which of the optional subfunctions added in MS-DOS versions 3.0 and 3.2 are supported by the driver, and, in the case of block drivers, whether the driver supports IBM-compatible disk media. The least significant 4 bits of the device attribute flags word control whether MS-DOS should use the driver as the standard input, standard output, clock, or NUL device; each of these 4 bits should be set on only one driver in the system at a time.

Table 15-1. Device Attribute Word in Device Header.

Bit	Setting
15*	1 if character device, 0 if block device
14*	1 if IOCTL Read and Write supported
13*	1 if non-IBM format (block device) 1 if Output Until Busy supported (character device)
12	0 (reserved)
11*	1 if Open/Close/Removable Media supported (versions 3.0 and later)
10	0 (reserved)
9	0 (reserved)
8	0 (reserved)
7	0 (reserved)
6*	1 if Generic IOCTL and Get/Set Logical Drive supported (version 3.2)
5	0 (reserved)
4	1 if special fast output function for CON device supported
3	1 if current CLOCK device
2	1 if current NUL device
1	1 if current standard output (<i>stdout</i>)
0	1 if current standard input (<i>stdin</i>)

*Only bits 6, 11, and 13-15 have significance on block devices; the remainder should be zero.

The information in the device header is ordinarily used only by the MS-DOS kernel and is not available to application programs. However, the IOCTL subfunctions Get and Set Device Data (Interrupt 21H Function 44H Subfunctions 00H and 01H) can be used to inspect or modify some of the bits in the device attribute flags word. Note that there is not a one-to-one correspondence between the bits defined for those functions and the bits in the device header. For example, in the device information word used by the IOCTL subfunctions, bit 7 indicates a block or character device; in the device attribute word of the device header, bit 15 indicates a block or character device.

The Strategy routine (*Strat*)

MS-DOS calls the driver's Strategy routine as the first step of any operation, passing it the segment and offset of a data structure called a request header in registers ES:BX. The Strategy routine saves this pointer for subsequent processing by the Interrupt routine and returns to MS-DOS.

A request header is essentially a small buffer used for private communication between MS-DOS and the device driver. Both MS-DOS and the device driver read and write information in the request header.

The first 13 bytes of a request header are the same for all device-driver functions and are therefore referred to as the static portion of the header. The number and contents of the subsequent bytes vary according to the type of operation being requested by the MS-DOS

kernel (Figure 15-3). The request header's most important component is the command code passed in its third byte; this code selects a driver function such as Read or Write. Other information passed to the driver in the request header includes unit numbers, transfer addresses, and sector or byte counts.

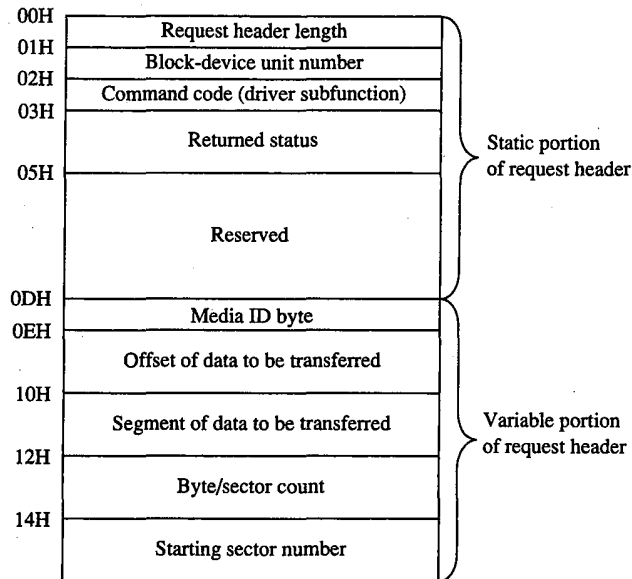


Figure 15-3. A typical driver request header. The bytes following the static portion are the format used for driver Read, Write, Write with Verify, IOCTL Read, and IOCTL Write operations.

The Interrupt routine (*Intr*)

The last and most complex part of a device driver is the Interrupt routine, which is called by MS-DOS immediately after the call to the Strategy routine. The bulk of the Interrupt routine is a collection of functions or subroutines, sometimes called command-code routines, that carry out each of the various operations the MS-DOS kernel requires a driver to support.

When the Interrupt routine receives control from MS-DOS, it saves any affected registers, examines the request header whose address was previously passed in the call to the Strategy routine, determines which command-code routine is needed, and branches to the appropriate function. When the operation is completed, the Interrupt routine stores the status (Table 15-2), error (Table 15-3), and any other applicable information into the request header, restores the previous contents of the affected registers, and returns to the MS-DOS kernel.

Table 15-2. The Request Header Status Word.

Bits	Meaning
15	Error
12-14	Reserved
9	Busy
8	Done
0-7	Error code if bit 15 = 1

Table 15-3. Device-Driver Error Codes.*

Code	Meaning
00H	Write-protect violation
01H	Unknown unit
02H	Drive not ready
03H	Unknown command
04H	CRC error
05H	Bad drive request structure length
06H	Seek error
07H	Unknown media
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0DH	Reserved
0EH	Reserved
0FH	Invalid disk change (versions 3.x)

*Returned in bits 0-7 of the request header status word.

The Interrupt routine's name is misleading in that it is never entered asynchronously as a hardware interrupt. The division of function between the Strategy and Interrupt routines is present for symmetry with UNIX/XENIX and MS OS/2 drivers but is essentially meaningless in single-tasking MS-DOS because there is never more than one I/O request in progress at a time.

The command-code functions

A total of twenty command codes are defined for MS-DOS device drivers. The command codes and the names of their associated Interrupt routines are shown in the following list:

Code	Routine
0	Init (initialization)
1	Media Check (block devices only)
2	Build BIOS Parameter Block (block devices only)
3	IOCTL Read
4	Read (Input)
5	Nondestructive Read (character devices only)
6	Input Status (character devices only)
7	Flush Input Buffers (character devices only)
8	Write (Output)
9	Write with Verify
10	Output Status (character devices only)
11	Flush Output Buffers (character devices only)
12	IOCTL Write
13*	Device Open
14*	Device Close
15*	Removable Media (block devices only)
16*	Output Until Busy (character devices only)
19†	Generic IOCTL Request
23†	Get Logical Device (block devices only)
24†	Set Logical Device (block devices only)

*MS-DOS versions 3.0 and later

†MS-DOS version 3.2

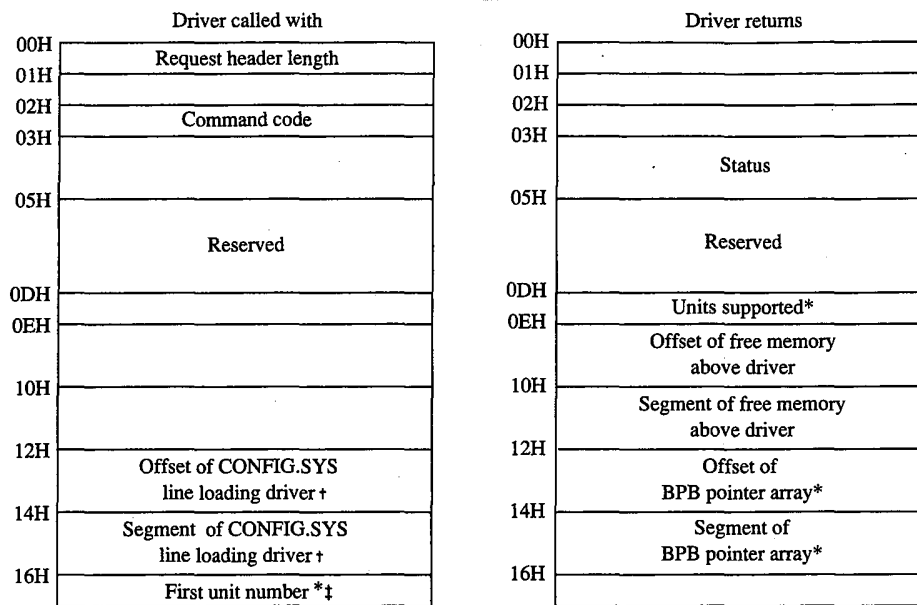
Functions 0 through 12 must be supported by a driver's Interrupt section under all versions of MS-DOS. Drivers tailored for versions 3.0 and 3.1 can optionally support an additional 4 functions defined under those versions of the operating system and drivers designed for version 3.2 can support 3 more, for a total of 20. MS-DOS inspects the bits in the device attribute word of the device header to determine which of the optional version 3.x functions a driver supports, if any.

As noted in the list above, some of the functions are relevant only for character drivers, some only for block drivers, and some for both. In any case, there must be an executable routine present for each function, even if the routine does nothing but set the done flag in the status word of the request header. The general requirements for each function routine are described below.

The Init function

The Init (initialization) function (command code 0) for a driver is called only once, when the driver is loaded (Figure 15-4). Init is responsible for checking that the hardware device controlled by the driver is present and functional, performing any necessary hardware initialization (such as a reset on a printer or a seek to the home track on a disk device), and capturing any interrupt vectors that the driver will need later.

The Init function is passed a pointer in the request header to the text of the DEVICE line in CONFIG.SYS that caused the driver to be loaded — specifically, the address of the next byte after the equal sign (=). The line is read-only and is terminated by a linefeed or carriage-return character; it can be scanned by the driver for switches or other parameters that might influence the driver's operation. (Alphabetic characters in the line are folded to uppercase.) With versions 3.0 and later, block drivers are also passed the drive number that will be assigned to their first unit (0 = A, 1 = B, and so on).



* Block-device drivers only

† Points to the character after DEVICE=

‡ MS-DOS 3.0 and later only

Figure 15-4. Initialization request header (command code 0).

When it returns to the kernel, the Init function must set the done flag in the status word of the request header and return the address of the start of free memory after the driver (sometimes called the break address). This address tells the kernel where it can build certain control structures of its own associated with the driver and then load the next driver. The Init routine of a block-device driver must also return the number of logical units supported by the driver and the address of a BPB pointer array.

The number of units returned by a block driver is used to assign device identifiers. For example, if at the time the driver is loaded there are already drivers present for four block devices (drive codes 0–3, corresponding to drive identifiers A through D) and the driver being initialized supports four units, it will be assigned the drive numbers 4 through 7

(corresponding to the drive names E through H). (Although there is also a field in the device header for the number of units, it is not inspected by MS-DOS; rather, it is set by MS-DOS from the information returned by the Init function.)

The BPB pointer array is an array of word offsets to BIOS parameter blocks. See *The Build BIOS Parameter Block Function* below; PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices. The array must contain one entry for each unit defined by the driver, although all entries can point to the same BPB to conserve memory. During the operating-system boot sequence, MS-DOS scans all the BPBs defined by all the units in all the resident block-device drivers to determine the largest sector size that exists on any device in the system; this information is used to set MS-DOS's cache buffer size. Thus, the sector size in the BPB of any installable block driver must be no larger than the largest sector size used by the resident block drivers.

If the Init routine finds that its hardware device is missing or defective, it can bypass the installation of the driver completely by returning the following values in the request header:

Item	Value
Number of units	0
Address of free memory	Segment and offset of the driver's own device header

A character-device driver must also clear bit 15 of the device attribute word in the device header so that MS-DOS will load the next driver in the same location as the one that just terminated itself.

The operating-system services that can be invoked by the Init routine are very limited. Only MS-DOS Interrupt 21H Functions 01-0CH (various character input and output services), 25H (Set Interrupt Vector), 30H (Get MS-DOS Version Number), and 35H (Get Interrupt Vector) can be called by the Init code. These functions assist the driver in configuring itself for the version of the host operating system it is to run under, capturing vectors for hardware interrupts, and displaying informational or error messages.

The amount of RAM required by a device driver can be reduced by positioning the Init routine at the end of the driver and returning that routine's starting address as the location of the first free memory.

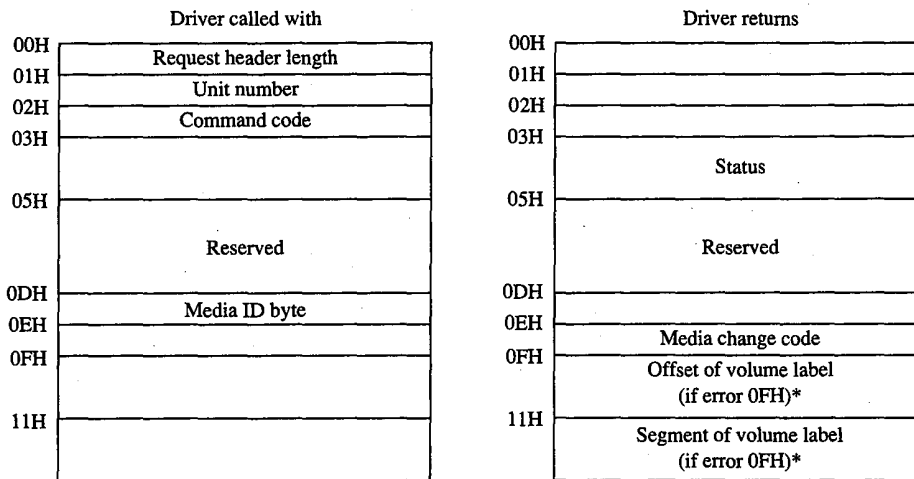
The Media Check function

The Media Check function (command code 1) is used only in block-device drivers. It is called by the MS-DOS kernel when there is a pending drive access call other than a simple file read or write (for example, a file open, close, rename, or delete), passing the media ID byte (Figure 15-5) for the disk that MS-DOS assumes is in the drive:

Description	Medium
0F9H	5.25-inch double-sided, 15 sectors
0FCH	5.25-inch single-sided, 9 sectors
0FDH	5.25-inch double-sided, 9 sectors
0FEH	5.25-inch single-sided, 8 sectors
0FFH	5.25-inch double-sided, 8 sectors
0F9H	3.5-inch double-sided, 9 sectors
0FOH	3.5-inch double-sided, 18 sectors
0F8H	Fixed disk

The function returns a code indicating whether the medium has been changed since the last transfer:

Code	Meaning
-1	Medium changed
0	Don't know if medium changed
1	Medium not changed



* MS-DOS 3.0 and later only

Figure 15-5. Media Check request header (command code 1).

If the Media Check routine asserts that the disk has not been changed, MS-DOS bypasses rereading the FAT and proceeds with the disk access. If the returned code indicates that the disk has been changed, MS-DOS invalidates all buffers associated with the drive, including buffers containing data waiting to be written (this data is simply lost), performs a Build BPB call, and then reads the disk's FAT and directory.

The action taken by MS-DOS when *Don't know* is returned depends on the state of its internal buffers. If data that needs to be written out is present in the buffers associated with the drive, MS-DOS assumes that no disk change has occurred. If the buffers are empty or have all been previously flushed to the disk, MS-DOS assumes that the disk was changed and proceeds as described above for the *Medium changed* return code.

If bit 11 of the device attribute word is set (that is, the driver supports the optional Open/Close/Removable Media functions), the host system is MS-DOS version 3.0 or later, and the function returns the *Medium changed* code (-1), the function must also return the segment and offset of the ASCIIZ volume label for the previous disk in the drive. (If the driver does not have the volume label, it can return a pointer to the ASCIIZ string *NO NAME*.) If MS-DOS determines that the disk was changed with unwritten data still present in the buffers, it issues a critical error 0FH (Invalid Disk Change). Application programs can trap this critical error and prompt the user to replace the original disk.

In character-device drivers, the Media Change function should simply set the done flag in the status word of the request header and return.

The Build BIOS Parameter Block function

The Build BPB function (command code 2) is supported only on block devices. MS-DOS calls this function when the *Medium changed* code has been returned by the Media Check routine or when the *Don't know* code has been returned and there are no dirty buffers (buffers that have not yet been written to disk). Thus, a call to this function indicates that the disk has been legally changed.

The Build BPB call receives a pointer to a one-sector buffer in the request header (Figure 15-6). If the non-IBM-format bit (bit 13) in the device attribute word in the device header is zero, the buffer contains the first sector of the disk's FAT, with the media ID byte in the first byte of the buffer. In this case, the contents of the buffer should not be modified by the driver. However, if the non-IBM-format bit is set, the buffer can be used by the driver as scratch space.

The Build BPB function must return the segment and offset of a BIOS parameter block (Table 15-4) for the disk format indicated by the media ID byte and set the done flag in the status word of the request header. The information in the BPB is used by the kernel to interpret the disk structure and is also used by the driver itself to translate logical sector addresses into physical track, sector, and head addresses. If bit 11 of the device attribute word is set (that is, the driver supports the optional Open/Close/Removable Media functions) and the host system is MS-DOS version 3.0 or later, this routine should also read the volume label from the disk and save it.

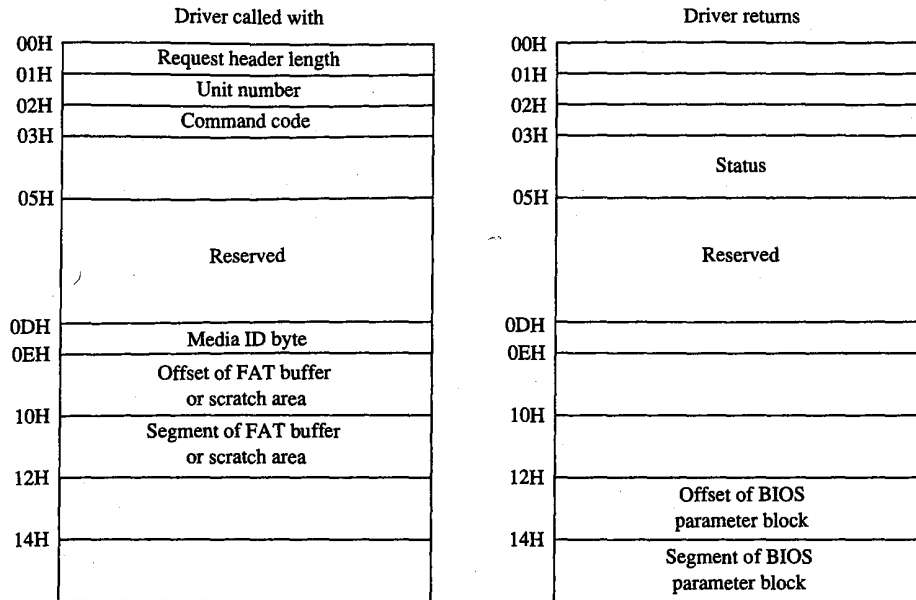


Figure 15-6. Build BPB request header (command code 2).

Table 15-4. Format of a BIOS Parameter Block (BPB).

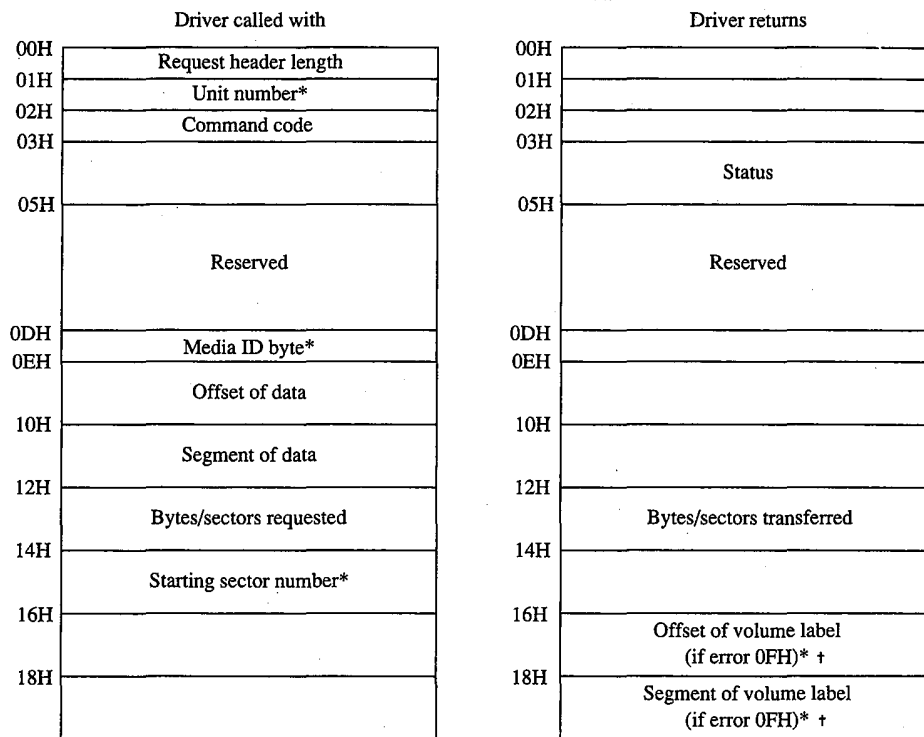
Bytes	Contents
00-01H	Bytes per sector
02H	Sectors per allocation unit (must be power of 2)
03-04H	Number of reserved sectors (starting at sector 0)
05H	Number of file allocation tables (FATs)
06-07H	Maximum number of root-directory entries
08-09H	Total number of sectors in medium
0AH	Media ID byte
0B-0CH	Number of sectors occupied by a single FAT
0D-0EH	Sectors per track (versions 3.0 and later)
0F-10H	Number of heads (versions 3.0 and later)
11-12H	Number of hidden sectors (versions 3.0 and later)
13-14H	High-order word of number of hidden sectors (version 3.2)
15-18H	If bytes 8-9 are zero, total number of sectors in medium (version 3.2)

In character-device drivers, the Build BPB function should simply set the done flag in the status word of the request header and return.

The Read, Write, and Write with Verify functions

The Read (Input) function (command code 4) transfers data from the device into a specified memory buffer. The Write (Output) function (command code 8) transfers data from a specified memory buffer to the device. The Write with Verify function (command code 9) works like the Write function but, if feasible, also performs a read-after-write verification that the data was transferred correctly. The MS-DOS kernel calls the Write with Verify function, instead of the Write function, whenever the system's global verify flag has been turned on with the VERIFY command or with Interrupt 21H Function 2EH (Set Verify Flag).

All three of these driver functions are called by the MS-DOS kernel with the address and length of the buffer for the data to be transferred. In the case of block-device drivers, the kernel also passes the drive unit code, the starting logical sector number, and the media ID byte for the disk (Figure 15-7).



* Block-device drivers only

† MS-DOS 3.0 and later, command codes 4, 8, and 9 only

Figure 15-7. The request header for IOCTL Read (command code 3), Read (command code 4), Write (command code 8), Write with Verify (command code 9), IOCTL Write (command code 12), and Output Until Busy (command code 16).

The Read and Write functions must perform the requested I/O, first translating each logical sector number for a block device into a physical track, head, and sector with the aid of the BIOS parameter block. Then the functions must return the number of bytes or sectors actually transferred in the appropriate field of the request header and also set the done flag in the request header status word. If an error is encountered during an operation, the functions must set the done flag, the error flag, and the error type in the status word and also report the number of bytes or sectors successfully transferred before the error; it is not sufficient to simply report the error.

Under MS-DOS versions 3.0 and later, the Read and Write functions can optionally use the reference count of open files maintained by the driver's Device Open and Device Close functions, together with the media ID byte, to determine whether the medium has been illegally changed. If the medium was changed with files open, the driver can return the error code 0FH and the segment and offset of the volume label for the correct disk so that the user can be prompted to replace the disk.

The Nondestructive Read function

The Nondestructive Read function (command code 5) is supported only on character devices. It allows MS-DOS to look ahead in the character stream by one character and is used to check for Control-C characters pending at the keyboard.

The function is called by the kernel with no parameters other than the command code itself (Figure 15-8). It must set the done bit in the status word of the request header and also set the busy bit in the status word to reflect whether the device's input buffer is empty (busy bit = 1) or contains at least one character (busy bit = 0). If the latter, the function must also return the next character that would be obtained by a kernel call to the Read function, without removing that character from the buffer (hence the term nondestructive).

In block-device drivers, the Nondestructive Read function should simply set the done flag in the status word of the request header and return.

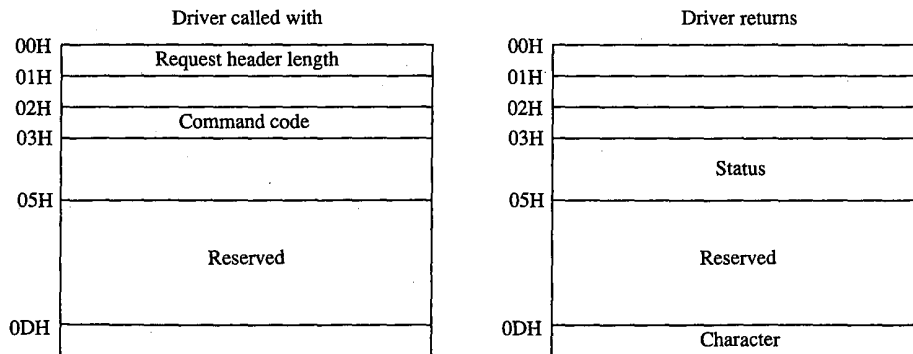


Figure 15-8. The Nondestructive Read request header.

The Input Status and Output Status functions

The Input Status and Output Status functions (command codes 6 and 10) are defined only for character devices. They are called with no parameters in the request header other than the command code itself and return their results in the busy bit of the request header status word (Figure 15-9). These functions constitute the driver-level support for the services the MS-DOS kernel provides to application programs by means of Interrupt 21H Function 44H Subfunctions 06H and 07H (Check Input Status and Check Output Status).

MS-DOS calls the Input Status function to determine whether there are characters waiting in a type-ahead buffer. The function sets the done bit in the status word of the request header and sets the busy bit to 0 if at least one character is already in the input buffer or to 1 if no characters are in the buffer and a read request would wait on a character from the physical device. If the character device does not have a type-ahead buffer, the Input Status routine should always return the busy bit set to 0 so that MS-DOS will not wait for something to arrive in the buffer before calling the Read function.

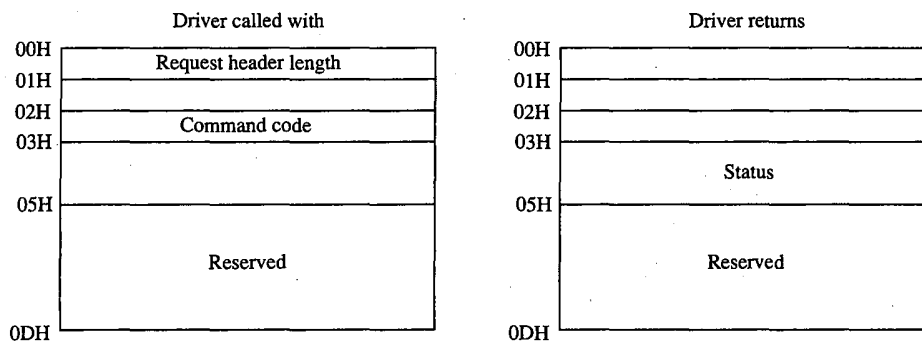


Figure 15-9. The request header for Input Status (command code 6), Flush Input Buffers (command code 7), Output Status (command code 10), and Flush Output Buffers (command code 11).

MS-DOS uses the Output Status function to determine whether a write operation is already in progress for the device. The function must set the done bit and the busy bit (0 if the device is idle and a write request would start immediately; 1 if a write is already in progress and a new write request would be delayed) in the status word of the request header.

In block-device drivers, the Input Status and Output Status functions should simply set the done flag in the status word of the request header and return.

The Flush Input Buffer and Flush Output Buffer functions

The Flush Input Buffer and Flush Output Buffer functions (command codes 7 and 11) are defined only for character devices. They simply terminate any read (for Flush Input) or write (for Flush Output) operations that are in progress and empty the associated buffer. The Flush Input Buffer function is used by MS-DOS to discard characters waiting in the type-ahead queue. This driver action corresponds to the MS-DOS service provided to application programs by means of Interrupt 21H Function 0CH (Flush Buffer, Read Keyboard).

These functions are called with no parameters in the request header other than the command code itself (see Figure 15-9) and return only the status word.

In block-device drivers, the Flush Buffer functions have no meaning. They should simply set the done flag in the status word of the request header and return.

The IOCTL Read and IOCTL Write functions

The IOCTL (I/O Control) Read and IOCTL Write functions (command codes 3 and 12) allow control information to be passed directly between a device driver and an application program. The IOCTL Read and Write driver functions are called by the MS-DOS kernel only if the IOCTL flag (bit 14) is set in the device attribute word of the device header.

The MS-DOS kernel passes the address and length of the buffer that contains or will receive the IOCTL information (see Figure 15-7). The driver must return the actual count of bytes transferred and set the done flag in the request header status word. Any error code returned by the driver is ignored by the kernel.

IOCTL Read and IOCTL Write operations are typically used to configure a driver or device or to report driver or device status and do not usually result in the transfer of data to or from the physical device. These functions constitute the driver support for the services provided to application programs by the MS-DOS kernel through Interrupt 21H Function 44H Subfunctions 02H, 03H, 04H, and 05H (Receive Control Data from Character Device, Send Control Data to Character Device, Receive Control Data from Block Device, and Send Control Data to Block Device).

The Device Open and Device Close functions

The Device Open and Device Close functions (command codes 13 and 14) are supported only in MS-DOS versions 3.0 and later and are called only if the open/close/removable media flag (bit 11) is set in the device attribute word of the device header. The Device Open and Device Close functions have no parameters in the request header other than the unit code for block devices and return nothing except the done flag and, if applicable, the error flag and number in the request header status word (Figure 15-10).

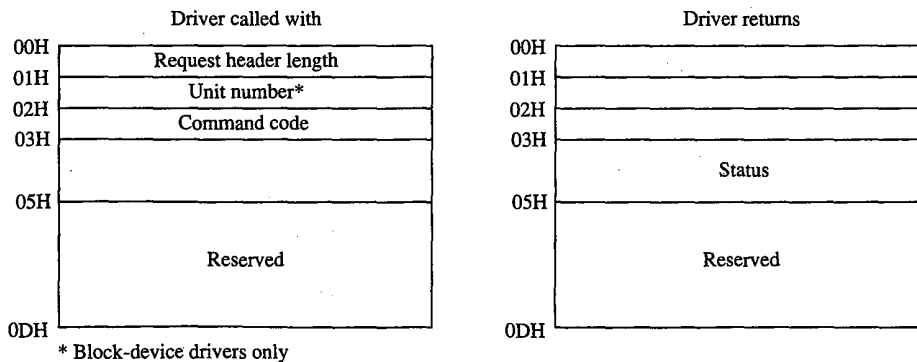


Figure 15-10. The request header for Device Open (command code 13), Device Close (command code 14), and Removable Media (command code 15).

Each Interrupt 21H request by an application to open or create a file or to open a character device for input or output results in a Device Open call by the kernel to the corresponding device driver. Similarly, each Interrupt 21H call by an application to close a file or device results in a Device Close call by the kernel to the appropriate device driver. These Device Open and Device Close calls are in addition to any directory read or write calls that may be necessary.

On block devices, the Device Open and Device Close functions can be used to manage local buffering and to maintain a reference count of the number of open files on a device. Whenever this reference count is decremented to zero, all files on the disk have been closed and the driver should flush any internal buffers so that data is not lost, as the user may be about to change disks. The reference count can also be used together with the media ID byte by the Read and Write functions to determine whether the disk has been changed while files are still open.

The reference count should be forced to zero when a Media Check call that returns the *Medium changed* code is followed by a Build BPB call, to provide for those programs that use FCBs to open files and then never close them. This problem does not arise with programs that use the handle functions for file management, because all handles are always closed automatically by MS-DOS on behalf of the program when it terminates. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

On character devices, the Device Open and Device Close functions can be used to send hardware-dependent initialization and post-I/O strings to the associated device (for example, a reset sequence or formfeed character to precede new output and a formfeed to follow it). Although these strings can be written directly by an application using ordinary write function calls, they can also be previously passed to the driver by application programs with IOCTL Write calls (Interrupt 21H Function 44H Subfunction 05H), which in turn are translated by the MS-DOS kernel into driver command code 12 (IOCTL Write) requests. The latter method makes the driver responsible for sending the proper control strings to the device each time a Device Open or Device Close is executed, but this method can be used only with drivers specifically written to support it.

The Removable Media function

The Removable Media function (command code 15) is defined only for block devices. It is supported in MS-DOS versions 3.0 and later and is called by MS-DOS only if the open/close/removable media flag (bit 11) is set in the device attribute word of the device header. This function constitutes the driver-level support for the service provided to application programs by MS-DOS by means of Interrupt 21H Function 44H Subfunction 08H (Check If Block Device Is Removable).

The only parameter for the Removable Media function is the unit code (see Figure 15-10). The function sets the done bit in the request header status word and sets the busy bit to 1 if the disk is not removable or to 0 if the disk is removable. This information can be used by MS-DOS to optimize its accesses to the disk and to eliminate unnecessary FAT and directory reads.

In character-device drivers, the Removable Media function should simply set the done flag in the status word of the request header and return.

The Output Until Busy function

The Output Until Busy function (command code 16) is defined only for character devices under MS-DOS versions 3.0 and later and is called by the MS-DOS kernel only if the corresponding flag (bit 13) is set in the device attribute word of the device header. This function is an optional driver-optimization function included specifically for the benefit of background print spoolers driving printers that have internal memory buffers. Such printers can accept data at a rapid rate until the buffer is full.

The Output Until Busy function is called with the address and length of the data to be written to the device (see Figure 15-7). It transfers data continuously to the device until the device indicates that it is busy or until the data is exhausted. The function then must set the done flag in the request header status word and return the actual number of bytes transferred in the appropriate field of the request header.

For this function to return a count of bytes transferred that is less than the number of bytes requested is not an error. MS-DOS will adjust the address and length of the data passed in the next Output Until Busy function request so that all characters are sent.

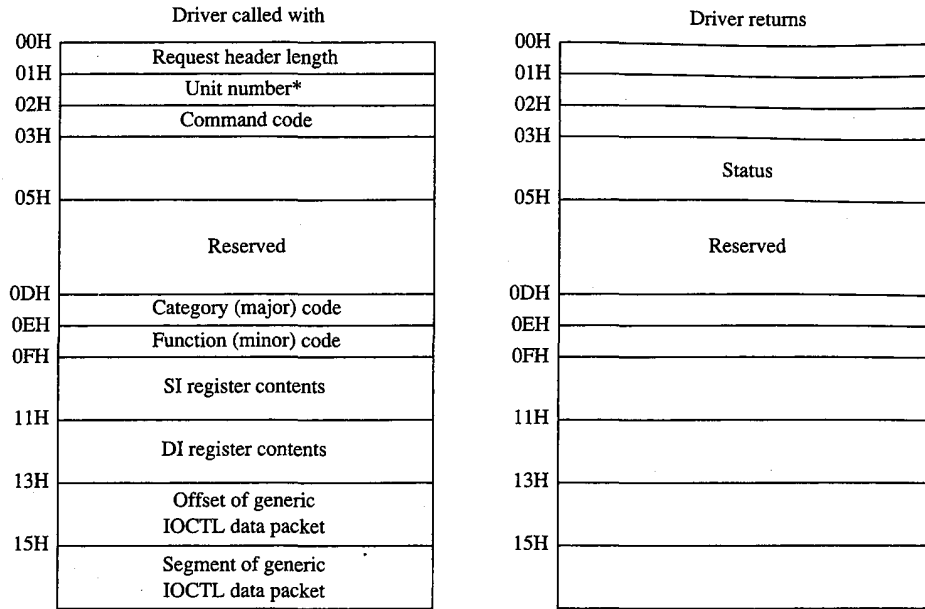
In block-device drivers, the Output Until Busy function should simply set the done flag in the status word of the request header and return.

The Generic IOCTL function

The Generic IOCTL function (command code 19) is defined under MS-DOS version 3.2 and is called only if the 3.2-functions-supported flag (bit 6) is set in the device attribute word of the device header. This driver function corresponds to the MS-DOS generic IOCTL service supplied to application programs by means of Interrupt 21H Function 44H Sub-functions 0CH (Generic I/O Control for Handles) and 0DH (Generic I/O Control for Block Devices).

In addition to the usual information in the static portion of the request header, the Generic IOCTL function is passed a category (major) code, a function (minor) code, the contents of the SI and DI registers at the point of the IOCTL call, and the segment and offset of a data buffer (Figure 15-11). This buffer in turn contains other information whose format depends on the major and minor IOCTL codes passed in the request header. The driver must interpret the major and minor codes in the request header and the contents of the additional buffer to determine which operation it will carry out and then set the done flag in the request header status word and return any other applicable information in the request header or the data buffer.

Services that can be invoked by the Generic IOCTL function, if the driver supports them, include configuring the driver for nonstandard disk formats, reading and writing entire disk tracks of data, and formatting and verifying tracks. The Generic IOCTL function has been designed to be open-ended so that it can be used to easily extend the device driver definition in future versions of MS-DOS.



* Block-device drivers only

Figure 15-11. Generic IOCTL request header.

The Get Logical Device and Set Logical Device functions

The Get and Set Logical Device functions (command codes 23 and 24) are defined only for block devices under MS-DOS version 3.2 and are called only if the 3.2-functions-supported flag (bit 6) is set in the device attribute word of the device header. They correspond to the Get and Set Logical Drive Map services supplied by MS-DOS to application programs by means of Interrupt 21H Function 44H Subfunctions 0EH and 0FH.

The Get and Set Logical Device functions are called with a drive unit number in the request header (Figure 15-12). Both functions return a status word for the operation in the request header; the Get Logical Device function also returns a unit number.

The Get Logical Device function is called to determine whether more than one drive letter is assigned to the same physical device. It returns a code for the last drive letter used to reference the device (1 = A, 2 = B, and so on); if only one drive letter is assigned to the device, the returned unit code should be 0.

The Set Logical Device function is called to inform the driver of the next logical drive identifier that will be used to reference the device. The unit code passed by the MS-DOS kernel in this case is zero based relative to the logical drives supported by this particular driver. For example, if the driver supports two logical floppy-disk-drive units (A and B), only one physical disk drive exists in the system, and Set Logical Device is called with a unit number of 1, the driver is being informed that the next read or write request from the MS-DOS kernel will be directed to drive B.

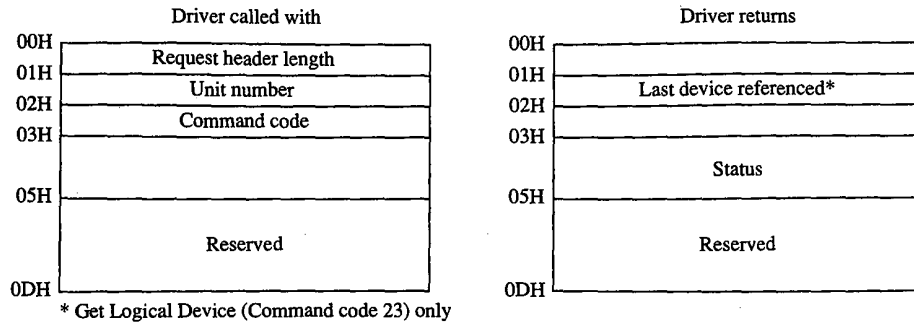


Figure 15-12. Get Logical Device and Set Logical Device request header.

In character-device drivers, the Get Logical Device and Set Logical Device functions should simply set the done flag in the status word of the request header and return.

The Processing of a Typical I/O Request

An application program requests an I/O operation from MS-DOS by loading registers with the appropriate values and addresses and executing a software Interrupt 21H. MS-DOS inspects its internal tables, searches the chain of device headers if necessary, and determines which device driver should receive the I/O request.

MS-DOS then creates a request header data packet in a reserved area of memory. Disk I/O requests are transformed from file and record information into logical sector requests by MS-DOS's interpretation of the disk directory and file allocation table. (MS-DOS locates these disk structures using the information returned by the driver from a previous Build BPB call and issues additional driver read requests, if necessary, to bring their sectors into memory.)

After the request header is prepared, MS-DOS calls the device driver's Strategy entry point, passing the address of the request header in registers ES:BX. The Strategy routine saves the address of the request header and performs a far return to MS-DOS.

MS-DOS then immediately calls the device driver's Interrupt entry point. The Interrupt routine saves all registers, retrieves the address of the request header that was saved by the Strategy routine, extracts the command code, and branches to the appropriate function to perform the operation requested by MS-DOS. When the requested function is complete, the Interrupt routine sets the done flag in the status word and places any other required information into the request header, restores all registers to their state at entry, and performs a far return.

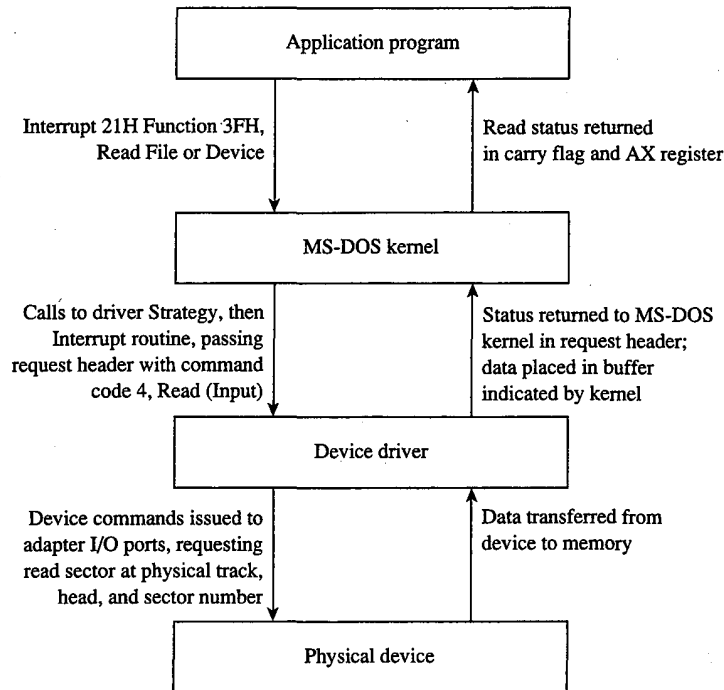


Figure 15-13. The processing of a typical I/O request from an application program.

MS-DOS translates the driver's returned status into the appropriate carry flag status, register values, and (possibly) error code for the MS-DOS Interrupt 21H function that was requested and returns control to the application program. Figure 15-13 sketches this entire flow of control and data.

Note that a single Interrupt 21H function request by an application program can result in many operation requests by MS-DOS to the device driver. For example, if the application invokes Interrupt 21H Function 3DH (Open File with Handle) to open a file, MS-DOS may have to issue multiple sector read requests to the driver while searching the directory for the filename. Similarly, an application program's request to write a string to the screen in cooked mode with Interrupt 21H Function 40H (Write File or Device) will result in a write request to the driver for each character in the string, because MS-DOS filters the characters and polls the keyboard for a pending Control-C between each character output.

Writing Device Drivers

Device drivers are traditionally coded in assembly language, both because of the rigid structural requirements and because of the need to keep driver execution speed high and memory overhead low. Although MS-DOS versions 3.0 and later are capable of loading

drivers in .EXE format, versions 2.x can load only pure memory-image device drivers that do not require relocation. Therefore, drivers are typically written as though they were .COM programs with an "origin" of zero and converted with EXE2BIN to .BIN or .SYS files so that they will be compatible with any version of MS-DOS (2.0 or later). See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

The device header must be located at the beginning of the file (offset 0). Both words in the header's link field should be set to -1, thus allowing MS-DOS to fix up the link field when the driver is loaded during system initialization so that it points to the next driver in the chain. When a single file contains more than one driver, the offset portion of each header link field should point to the next header in that file, all using the same segment base of zero, and only the link field of the last header in the file should be set to -1, -1.

The device attribute word must reflect the device-driver type (character or block) and the bits that indicate support for the various optional command codes must have appropriate values. The device header's offsets to the Strategy and Interrupt routines must be relative to the same segment base as the device header itself. If the driver is for a character device, the name field should be filled in properly with the device's logical name, which can be any legal eight-character uppercase filename padded with spaces and without a colon. Duplication of existing character-device names or existing disk-file names should be avoided (unless a resident character-device driver is being intentionally superseded).

The Strategy and Interrupt routines for the device are called by MS-DOS by means of an intersegment call (CALL FAR) and must return to MS-DOS with a far return. Both routines must preserve all CPU registers and flags. The MS-DOS kernel's stack has room for 40 to 50 bytes when the driver is called; if the driver makes heavy use of the stack, it should switch to an internal stack of adequate depth.

The Strategy routine is, of course, very simple. It need only save the address of the request header that is passed to it in registers ES:BX and exit back to the kernel.

The logic of the Interrupt routine is necessarily more complex. It must save the CPU registers and flags, extract the command code from the request header whose address was previously saved by the Strategy routine, and dispatch the appropriate command-code function. When that function is finished, the Interrupt routine must ensure that the appropriate status and other information is placed in the request header, restore the CPU registers and flags, and return control to the kernel.

Although the interface between the MS-DOS kernel and the command-code routines is fairly simple, it is also strict. The command-code functions must behave exactly as they are defined or the system will behave erratically. Even a very subtle discrepancy in the action of a driver function can have unexpectedly large global effects. For example, if a block driver Read function returns an error but does not return a correct value for the number of sectors successfully transferred, the MS-DOS kernel will be misled in its attempts to retry the read for only the failing sectors and disk data might be corrupted.

Example character driver: TEMPLATE

Figure 15-14 contains the source code for a skeleton character-device driver called TEMPLATE.ASM. This driver does nothing except display a sign-on message when it is loaded, but it demonstrates all the essential driver components, including the device header, Strategy routine, and Interrupt routine. The command-code functions take no action other than to set the done flag in the request header status word.

```

name    template
title   'TEMPLATE --- installable driver template'

;
; TEMPLATE.ASM:  A program skeleton for an installable
;               device driver (MS-DOS 2.0 or later)
;
; The driver command-code routines are stubs only and have
; no effect but to return a nonerror "Done" status.
;
; Ray Duncan, July 1987
;

_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:NOTHING

        org     0

MaxCmd  equ     24           ; maximum allowed command code
                          ; 12 for MS-DOS 2.x
                          ; 16 for MS-DOS 3.0-3.1
                          ; 24 for MS-DOS 3.2-3.3

cr      equ     0dh         ; ASCII carriage return
lf      equ     0ah         ; ASCII linefeed
eom     equ     '$'        ; end-of-message signal

Header:                                ; device driver header
      dd     -1            ; link to next device driver
      dw     0c840h        ; device attribute word
      dw     Strat         ; "Strategy" routine entry point
      dw     Intr          ; "Interrupt" routine entry point
      db     'TEMPLATE'    ; logical device name

RHPtr   dd     ?           ; pointer to request header, passed
                          ; by MS-DOS kernel to Strategy routine

```

Figure 15-14. TEMPLATE.ASM, the source file for the TEMPLATE.SYS driver.

(more)

```

Dispatch:                                ; Interrupt routine command-code
                                           ; dispatch table
dw    Init                                ; 0 = initialize driver
dw    MediaChk                            ; 1 = media check on block device
dw    BuildBPB                            ; 2 = build BIOS parameter block
dw    IoctlRd                             ; 3 = I/O control read
dw    Read                                ; 4 = read (input) from device
dw    NdRead                              ; 5 = nondestructive read
dw    InpStat                             ; 6 = return current input status
dw    InpFlush                            ; 7 = flush device input buffers
dw    Write                               ; 8 = write (output) to device
dw    WriteVfy                            ; 9 = write with verify
dw    OutStat                             ; 10 = return current output status
dw    OutFlush                            ; 11 = flush output buffers
dw    IoctlWt                             ; 12 = I/O control write
dw    DevOpen                             ; 13 = device open (MS-DOS 3.0+)
dw    DevClose                            ; 14 = device close (MS-DOS 3.0+)
dw    RemMedia                            ; 15 = removable media (MS-DOS 3.0+)
dw    OutBusy                             ; 16 = output until busy (MS-DOS 3.0+)
dw    Error                               ; 17 = not used
dw    Error                               ; 18 = not used
dw    GenIOCTL                            ; 19 = generic IOCTL (MS-DOS 3.2+)
dw    Error                               ; 20 = not used
dw    Error                               ; 21 = not used
dw    Error                               ; 22 = not used
dw    GetLogDev                           ; 23 = get logical device (MS-DOS 3.2+)
dw    SetLogDev                           ; 24 = set logical device (MS-DOS 3.2+)

Strat  proc  far                          ; device driver Strategy routine,
                                           ; called by MS-DOS kernel with
                                           ; ES:BX = address of request header

                                           ; save pointer to request header
mov    word ptr cs:[RHPtr],bx
mov    word ptr cs:[RHPtr+2],es

ret                                         ; back to MS-DOS kernel

Strat  endp

Intr   proc  far                          ; device driver Interrupt routine,
                                           ; called by MS-DOS kernel immediately
                                           ; after call to Strategy routine

push  ax                                  ; save general registers
push  bx
push  cx
push  dx
push  ds

```

Figure 15-14. Continued.

(more)


```

push    es
push    di
push    si
push    bp

push    cs          ; make local data addressable
pop     ds          ; by setting DS = CS

les     di,[RHPtr]  ; let ES:DI = request header

                                ; get BX = command code
mov     bl,es:[di+2]
xor     bh,bh
cmp     bx,MaxCmd   ; make sure it's valid
jle     Intr1       ; jump, function code is ok
call    Error       ; set error bit, "Unknown Command" code
jmp     Intr2

Intr1:  shl     bx,1   ; form index to dispatch table
                                ; and branch to command-code routine
call    word ptr [bx+Dispatch]

les     di,[RHPtr]  ; ES:DI = address of request header

Intr2:  or     ax,0100h ; merge Done bit into status and
mov     es:[di+3],ax ; store status into request header

pop     bp          ; restore general registers
pop     si
pop     di
pop     es
pop     ds
pop     dx
pop     cx
pop     bx
pop     ax
ret                                ; return to MS-DOS kernel

```

```

; Command-code routines are called by the Interrupt routine
; via the dispatch table with ES:DI pointing to the request
; header. Each routine should return AX = 00H if function was
; completed successfully or AX = 8000H + error code if
; function failed.

```

```

MediaChk proc near          ; function 1 = Media Check

xor     ax,ax
ret

MediaChk endp

```

Figure 15-14. Continued.

(more)

```
BuildBPB proc near ; function 2 = Build BPB
    xor ax,ax
    ret
BuildBPB endp

IoctlRd proc near ; function 3 = I/O Control Read
    xor ax,ax
    ret
IoctlRd endp

Read proc near ; function 4 = Read (Input)
    xor ax,ax
    ret
Read endp

NdRead proc near ; function 5 = Nondestructive Read
    xor ax,ax
    ret
NdRead endp

InpStat proc near ; function 6 = Input Status
    xor ax,ax
    ret
InpStat endp

InpFlush proc near ; function 7 = Flush Input Buffers
    xor ax,ax
    ret
InpFlush endp
```

Figure 15-14. Continued.

(more)

```
Write  proc  near          ; function 8 = Write (Output)
        xor   ax,ax
        ret

Write  endp

WriteVfy proc  near          ; function 9 = Write with Verify
        xor   ax,ax
        ret

WriteVfy endp

OutStat proc  near          ; function 10 = Output Status
        xor   ax,ax
        ret

OutStat endp

OutFlush proc  near          ; function 11 = Flush Output Buffers
        xor   ax,ax
        ret

OutFlush endp

IoctlWt proc  near          ; function 12 = I/O Control Write
        xor   ax,ax
        ret

IoctlWt endp

DevOpen proc  near          ; function 13 = Device Open
        xor   ax,ax
        ret

DevOpen endp
```

Figure 15-14. Continued.

(more)

```
DevClose proc near ; function 14 = Device Close
    xor ax,ax
    ret
DevClose endp

RemMedia proc near ; function 15 = Removable Media
    xor ax,ax
    ret
RemMedia endp

OutBusy proc near ; function 16 = Output Until Busy
    xor ax,ax
    ret
OutBusy endp

GenIOCTL proc near ; function 19 = Generic IOCTL
    xor ax,ax
    ret
GenIOCTL endp

GetLogDev proc near ; function 23 = Get Logical Device
    xor ax,ax
    ret
GetLogDev endp

SetLogDev proc near ; function 24 = Set Logical Device
    xor ax,ax
    ret
SetLogDev endp
```

Figure 15-14. Continued.

(more)

```

Error  proc  near          ; bad command code in request header
        mov  ax,8003h      ; error bit + "Unknown Command" code
        ret

Error  endp

Init   proc  near          ; function 0 = initialize driver
        push es           ; save address of request header
        push di

        mov  ah,9         ; display driver sign-on message
        mov  dx,offset Ident
        int  21h

        pop  di           ; restore request header address
        pop  es

                                ; set address of free memory
                                ; above driver (break address)
        mov  word ptr es:[di+14],offset Init
        mov  word ptr es:[di+16],cs

        xor  ax,ax        ; return status
        ret

Init   endp

Ident  db    cr,lf,lf
        db    'TEMPLATE Example Device Driver'
        db    cr,lf,eom

Intr   endp

_TEXT  ends

        end

```

Figure 15-14. Continued.

TEMPLATE.ASM can be assembled, linked, and converted into a loadable driver with the following commands:

```

C>MASM TEMPLATE; <Enter>
C>LINK TEMPLATE; <Enter>
C>EXE2BIN TEMPLATE.EXE TEMPLATE.SYS <Enter>

```

The Microsoft Object Linker (LINK) will display the warning message *No Stack Segment*; this message can be ignored. The driver can then be installed by adding the line

```

DEVICE=TEMPLATE.SYS

```

to the CONFIG.SYS file and restarting the system. The fact that the TEMPLATE.SYS driver also has the logical character-device name TEMPLATE allows the demonstration of an interesting MS-DOS effect: After the driver is installed, the file that contains it can no longer be copied, renamed, or deleted. The reason for this limitation is that MS-DOS always searches its list of character-device names first when an open request is issued, before it inspects the disk directory. The only way to erase the TEMPLATE.SYS file is to modify the CONFIG.SYS file to remove the associated DEVICE statement and then restart the system.

For a complete example of a character-device driver for interrupt-driven serial communications, See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Example block driver: TINYDISK

Figure 15-15 contains the source code for a simple 64 KB virtual disk (RAMdisk) called TINYDISK.ASM. This code provides a working example of a simple block-device driver. When its Initialization routine is called by the kernel, TINYDISK allocates itself 64 KB of RAM and maps a disk structure onto the RAM in the form of a boot sector containing a valid BPB, a FAT, a root directory, and a files area. See PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices.

```

name    tinydisk
title   TINYDISK example block-device driver

; TINYDISK.ASM --- 64 KB RAMdisk
;
; Ray Duncan, July 1987
; Example of a simple installable block-device driver.

_TEXT   segment public 'CODE'

        assume  cs:_TEXT,ds:_TEXT,es:_TEXT

        org     0

MaxCmd  equ     12           ; max driver command code
                          ; (no MS-DOS 3.x functions)

cr      equ     0dh         ; ASCII carriage return
lf      equ     0ah         ; ASCII linefeed
blank   equ     020h       ; ASCII space code
eom     equ     '$'        ; end-of-message signal

Secsize equ     512        ; bytes/sector, IBM-compatible media

```

Figure 15-15. TINYDISK.ASM, the source file for the TINYDISK.SYS driver.

(more)

```

Header dd      -1          ; device-driver header
      dw      0           ; link to next driver in chain
      dw      0           ; device attribute word
      dw      Strat       ; "Strategy" routine entry point
      dw      Intr        ; "Interrupt" routine entry point
      db      1           ; number of units, this device
      db      7 dup (0)   ; reserved area (block-device drivers)

RHPtr  dd      ?          ; segment:offset of request header

Secseg dw      ?          ; segment base of sector storage

Xfrsec dw      0          ; current sector for transfer
Xfrcnt dw      0          ; sectors successfully transferred
Xfrreq dw      0          ; number of sectors requested
Xfraddr dd     0          ; working address for transfer

Array  dw      BPB        ; array of pointers to BPB
      ; for each supported unit

Bootrec equ    $

      jmp     $           ; phony JMP at start of
      nop     ; boot sector; this field
      ; must be 3 bytes

      db     'MS  2.0'   ; OEM identity field

      ; BIOS Parameter Block (BPB)
BPB    dw      Secsize   ; 00H - bytes per sector
      db      1          ; 02H - sectors per cluster
      dw      1          ; 03H - reserved sectors
      db      1          ; 05H - number of FATs
      dw      32         ; 06H - root directory entries
      dw      128        ; 08H - sectors = 64 KB/secsize
      db      0f8h       ; 0AH - media descriptor
      dw      1          ; 0BH - sectors per FAT

Bootrec_len equ $-Bootrec

Strat  proc    far        ; RAMdisk strategy routine

      ; save address of request header
      mov     word ptr cs:RHPtr,bx
      mov     word ptr cs:[RHPtr+2],es
      ret     ; back to MS-DOS kernel

Strat  endp

```

Figure 15-15. Continued.

(more)

```

Intr  proc  far           ; RAMdisk interrupt routine

      push  ax           ; save general registers
      push  bx
      push  cx
      push  dx
      push  ds
      push  es
      push  di
      push  si
      push  bp

      mov   ax,cs        ; make local data addressable
      mov   ds,ax

      les   di,[RHPtr]   ; ES:DI = request header

      mov   bl,es:[di+2] ; get command code
      xor   bh,bh
      cmp   bx,MaxCmd    ; make sure it's valid
      jle   Intr1        ; jump, function code is ok
      mov   ax,8003h     ; set Error bit and
      jmp   Intr3        ; "Unknown Command" error code

Intr1: shl   bx,1        ; form index to dispatch table and
                        ; branch to command-code routine
      call  word ptr [bx+Dispatch]
                        ; should return AX = status

      les   di,[RHPtr]   ; restore ES:DI = request header

Intr3: or    ax,0100h    ; merge Done bit into status and store
      mov   es:[di+3],ax ; status into request header

Intr4: pop   bp         ; restore general registers
      pop   si
      pop   di
      pop   es
      pop   ds
      pop   dx
      pop   cx
      pop   bx
      pop   ax
      ret                ; return to MS-DOS kernel

Intr  endp

```

Figure 15-15. Continued.

(more)


```

Dispatch:                                ; command-code dispatch table
                                           ; all command-code routines are
                                           ; entered with ES:DI pointing
                                           ; to request header and return
                                           ; the operation status in AX
dw      Init                             ; 0 = initialize driver
dw      MediaChk                         ; 1 = media check on block device
dw      BuildBPB                         ; 2 = build BIOS parameter block
dw      Dummy                             ; 3 = I/O control read
dw      Read                             ; 4 = read (input) from device
dw      Dummy                             ; 5 = nondestructive read
dw      Dummy                             ; 6 = return current input status
dw      Dummy                             ; 7 = flush device input buffers
dw      Write                             ; 8 = write (output) to device
dw      Write                             ; 9 = write with verify
dw      Dummy                             ; 10 = return current output status
dw      Dummy                             ; 11 = flush output buffers
dw      Dummy                             ; 12 = I/O control write

MediaChk proc near                        ; command code 1 = Media Check
                                           ; return "not changed" code
mov     byte ptr es:[di+0eh],1
xor     ax,ax                            ; and success status
ret

MediaChk endp

BuildBPB proc near                       ; command code 2 = Build BPB
                                           ; put BPB address in request header
mov     word ptr es:[di+12h],offset BPB
mov     word ptr es:[di+14h],cs
xor     ax,ax                            ; return success status
ret

BuildBPB endp

Read proc near                           ; command code 4 = Read (Input)
                                           ; set up transfer variables
call    Setup

Read1:  mov     ax,Xfrcnt                  ; done with all sectors yet?
        cmp     ax,Xfrreq
        je     Read2                      ; jump if transfer completed
        mov     ax,Xfrsec                 ; get next sector number
        call    Mapsec                    ; and map it

```

Figure 15-15. Continued.

(more)

```

        mov     ax,es
        mov     si,di
        les     di,Xfraddr      ; ES:DI = requester's buffer
        mov     ds,ax          ; DS:SI = RAMdisk address
        mov     cx,Secsize     ; transfer logical sector from
        cld                     ; RAMdisk to requestor
        rep movsb
        push    cs              ; restore local addressing
        pop     ds
        inc     Xfrsec          ; advance sector number
                               ; advance transfer address
        add     word ptr Xfraddr,Secsize
        inc     Xfrcnt          ; count sectors transferred
        jmp     Read1

Read2:                                     ; all sectors transferred
        xor     ax,ax           ; return success status
        les     di,RHPtr        ; put actual transfer count
        mov     bx,Xfrcnt       ; into request header
        mov     es:[di+12h],bx
        ret

Read   endp

Write  proc   near              ; command code 8 = Write (Output)
                               ; command code 9 = Write with Verify

        call   Setup            ; set up transfer variables

Write1: mov     ax,Xfrcnt        ; done with all sectors yet?
        cmp     ax,Xfrreq
        je     Write2           ; jump if transfer completed

        mov     ax,Xfrsec       ; get next sector number
        call   Mapsec           ; and map it
        lds     si,Xfraddr
        mov     cx,Secsize     ; transfer logical sector from
        cld                     ; requester to RAMdisk
        rep movsb
        push    cs              ; restore local addressing
        pop     ds
        inc     Xfrsec          ; advance sector number
                               ; advance transfer address
        add     word ptr Xfraddr,Secsize
        inc     Xfrcnt          ; count sectors transferred
        jmp     Write1

Write2:                                     ; all sectors transferred
        xor     ax,ax           ; return success status
        les     di,RHPtr        ; put actual transfer count

```

Figure 15-15. Continued.

(more)

```

        mov     bx,Xfrcnt      ; into request header
        mov     es:[di+12h],bx
        ret

Write   endp

Dummy   proc   near          ; called for unsupported functions

        xor     ax,ax        ; return success flag for all
        ret

Dummy   endp

Mapsec  proc   near          ; map sector number to memory address
                                ; call with AX = logical sector no.
                                ; return ES:DI = memory address

        mov     di,Secsize/16 ; paragraphs per sector
        mul     di           ; * logical sector number
        add     ax,Secseg    ; + segment base of sector storage
        mov     es,ax
        xor     di,di        ; now ES:DI points to sector
        ret

Mapsec  endp

Setup   proc   near          ; set up for read or write
                                ; call ES:DI = request header
                                ; extracts address, start, count

        push    es          ; save request header address
        push    di
        mov     ax,es:[di+14h] ; starting sector number
        mov     Xfrsec,ax
        mov     ax,es:[di+12h] ; sectors requested
        mov     Xfrreq,ax
        les     di,es:[di+0eh] ; requester's buffer address
        mov     word ptr Xfraddr,di
        mov     word ptr Xfraddr+2,es
        mov     Xfrcnt,0     ; initialize sectors transferred count
        pop     di          ; restore request header address
        pop     es
        ret

Setup   endp

```

Figure 15-15. Continued.

(more)

```

Init    proc    near                ; command code 0 = Initialize driver
                                           ; on entry ES:DI = request header

        mov     ax,cs                ; calculate segment base for sector
        add     ax,Driver_len        ; storage and save it
        mov     Secseg,ax
        add     ax,1000h             ; add 1000H paras (64 KB) and
        mov     es:[di+10h],ax      ; set address of free memory
        mov     word ptr es:[di+0eh],0

        call    Format                ; format the RAMdisk

        call    Signon               ; display driver identification

        les     di,cs:RHPtr          ; restore ES:DI = request header
                                           ; set logical units = 1
        mov     byte ptr es:[di+0dh],1
                                           ; set address of BPB array
        mov     word ptr es:[di+12h],offset Array
        mov     word ptr es:[di+14h],cs

        xor     ax,ax                ; return success status
        ret

Init    endp

Format  proc    near                ; format the RAMdisk area

        mov     es,Secseg            ; first zero out RAMdisk
        xor     di,di
        mov     cx,8000h             ; 32 K words = 64 KB
        xor     ax,ax
        cld
        rep stosw

        mov     ax,0                 ; get address of logical
        call    Mapsec               ; sector zero
        mov     si,offset Bootrec
        mov     cx,Bootrec_len
        rep movsb                    ; and copy boot record to it

        mov     ax,word ptr BPB+3
        call    Mapsec               ; get address of 1st FAT sector
        mov     al,byte ptr BPB+0ah
        mov     es:[di],al           ; put media ID byte into it
        mov     word ptr es:[di+1],-1

        mov     ax,word ptr BPB+3
        add     ax,word ptr BPB+0bh
        call    Mapsec               ; get address of 1st directory sector

```

Figure 15-15. Continued.

(more)

```

        mov     si,offset Volname
        mov     cx,Volname_len
        rep movsb           ; copy volume label to it

        ret                ; done with formatting

Format endp

Signon proc near          ; driver identification message

        les     di,RHPtr   ; let ES:DI = request header
        mov     al,es:[di+22] ; get drive code from header,
        add     al,'A'     ; convert it to ASCII, and
        mov     drive,al   ; store into sign-on message

        mov     ah,30h     ; get MS-DOS version
        int     21h
        cmp     al,2
        ja     Signon1    ; jump if version 3.0 or later
        mov     Ident1,eom ; version 2.x, don't print drive

Signon1:                  ; print sign-on message
        mov     ah,09h     ; Function 09H = print string
        mov     dx,offset Ident ; DS:DX = address of message
        int     21h       ; transfer to MS-DOS

        ret                ; back to caller

Signon endp

Ident  db      cr,lf,lf    ; driver sign-on message
       db      'TINYDISK 64 KB RAMdisk'
       db      cr,lf
Ident1 db      'RAMdisk will be drive '
Drive  db      'X:'
       db      cr,lf,eom

Volname db      'DOSREF_DISK' ; volume label for RAMdisk
        db      08h          ; attribute byte
        db      10 dup (0)   ; reserved area
        dw      0            ; time = 00:00
        dw      0f01h       ; date = August 1, 1987
        db      6 dup (0)   ; reserved area

Volname_len equ $-volname

Driver_len dw (($-header)/16)+1 ; driver size in paragraphs

_TEXT ends

        end

```

Figure 15-15. Continued.

Subsequent driver Read and Write calls by the kernel to TINYDISK function as though they were transferring sectors to and from a physical storage device but actually only copy data from one area in memory to another. A programmer can learn a great deal about the operation of block-device drivers and MS-DOS's relationship to those drivers (such as the order and frequency of Media Change, Build BPB, Read, Write, and Write With Verify calls) by inserting software probes into TINYDISK at appropriate locations and monitoring its behavior.

TINYDISK.ASM can be assembled, linked, and converted into a loadable driver with the following commands:

```
C>MASM TINYDISK; <Enter>
C>LINK TINYDISK; <Enter>
C>EXE2BIN TINYDISK.EXE TINYDISK.SYS <Enter>
```

The linker will display the warning message *No Stack Segment*; this message can be ignored. The driver can then be installed by adding the line

```
DEVICE=TINYDISK.SYS
```

to the CONFIG.SYS file and restarting the system. When it is loaded, TINYDISK displays a sign-on message and the drive letter that it was assigned if it is running under MS-DOS version 3.0 or later. (If the host system is MS-DOS version 2.x, this information is not provided to the driver.) Files can then be copied to the RAMdisk as though it were a small but extremely fast disk drive.

Ray Duncan

Part D
Directions of MS-DOS

Article 16

Writing Applications for Upward Compatibility

One of the major concerns of the designers of Microsoft OS/2 was that it be backwardly compatible—that is, that programs written to run under MS-DOS versions 2 and 3 be able to run on MS OS/2. A major concern for present application programmers is that their programs run not only on current versions of MS-DOS (and MS OS/2) but also on future versions of MS-DOS. Ensuring such upward compatibility involves both hardware issues and operating-system issues.

Hardware Issues

A basic requirement for ensuring upward compatibility is hardware-independent code. If you bypass system services and directly program the hardware—such as the system interrupt controller, the system clock, and the enhanced graphics adapter (EGA) registers—your application will not run on future versions of MS-DOS.

Protected mode compatibility

The 80286 and the 80386 microprocessors can operate in two incompatible modes: real mode and protected mode. When either chip is operating in real mode, it is perceived by the operating system and programs as a fast 8088 chip. Applications written for the 8086 and 8088 run the same on the 80286 and the 80386—only faster. They cannot, however, take advantage of 80286 and 80386 features unless they can run in protected mode.

Following the guidelines below will minimize the work necessary to convert a real mode program to protected mode and will also allow a program to use a special subset of the MS OS/2 Applications Program Interface (API)—Family API. A binary program (.EXE) that uses the family API can run in either protected mode or real mode under MS OS/2 and subsequent systems, but it can run only in real mode under MS-DOS version 3.

Family API

The Family API requires that the application use a subset of the MS OS/2 Dynamic Link System API. Special tools link the application with a special library that implements the subset MS OS/2 system services in the MS-DOS version 3 environment. Many of these services are implemented by calling the appropriate Interrupt 21H subfunction; some are implemented in the special library itself.

When a Family API application is loaded under MS OS/2 protected mode, MS OS/2 ignores the special library code and loads only the application itself. MS OS/2 then provides the requested services in the normal fashion. However, MS-DOS version 3 loads the entire package — the application and the special library — because the Family API .EXE file is constructed to look like an MS-DOS 3 .EXE file.

Linear vs segmented memory

The protected mode and the real mode of the 80286 and the 80386 are compatible except in the area of segmentation. The 8086 has been described as a segmented machine, but it is actually a linear memory machine with offset registers. When a memory address is generated, the value in one of the “segment” registers is multiplied by 16 and added as a displacement to the offset value supplied by the instruction’s addressing mode. No length information is associated with each “segment”; the “segment” register supplies only a 20-bit addressing offset. Programs routinely use this by computing a 20-bit address and then decomposing it into a 16-bit “segment” value and a 16-bit displacement value so that the address can be referenced.

The protected mode of the 80286 and the 80386, however, is truly segmented. A value placed in a segment register selects an entry from a descriptor table; that entry contains the addressing offset, a segment length, and permission bits. On the 8086, the so-called segment component of an address is multiplied by 16 and added to the offset component, producing a 20-bit physical address. Thus, if you take an address in the *segment:offset* form, add 4 to the segment value, and subtract 64 (that is, $4 \cdot 16$) from the offset value, the new address references exactly the same location as the old address. On the 80286 and the 80386 in protected mode, however, segment values, called segment selectors, have no direct correspondence to physical addresses. In other words, in 8086 mode, the two address forms

$1000_{16} \cdot 0345_{16}$

and

$1004_{16} \cdot 0305_{16}$

reference the same memory location, but in protected mode these two forms reference totally different locations.

Creating segment values

This architectural difference gives rise to the most common cause of incompatibility — the program performs addressing arithmetic to compute “segment” values. Any program that uses the 20-bit addressing scheme to create or to compute a value to be loaded in a segment register cannot be converted to run in protected mode. To be protected mode compatible, a program must treat the 8086’s so-called segments as true segments.

To create a program that does this, write according to the following guidelines:

1. Do not generate any segment values. Use only the segment values supplied by MS-DOS calls and those placed in the segment registers when MS-DOS loaded your program. The exception is “huge objects” — memory objects larger than 64 KB. In

this case, MS OS/2 provides a base segment number and a "segment offset value." The returned segment number selects the first 64 KB of the object and the segment number, plus the segment offset value address the second 64 KB of the object. Likewise, the returned segment value plus $N \times$ (segment offset value) selects the $N+1$ 64 KB piece of the huge object. Write real mode code in this same fashion, using 4096 as the segment offset value. When you convert your program, you can substitute the value provided by MS OS/2.

2. Do not address beyond the allocated length of a segment.
3. Do not use segment registers as scratch registers by placing general data in them. Place only valid segment values, supplied by MS-DOS, in a segment register. The one exception is that you can place a zero value in a segment register, perhaps to indicate "no address." You can place the zero in the segment register, but you cannot reference memory using that register; you can only load/store or push/pop it.
4. Do not use CS: overrides on instructions that store into memory. It is impossible to store into a code segment in protected mode.

CPU speed

Because various microprocessors and machine configurations execute at different speeds, a program should not contain timing loops that depend on CPU speed. Specifically, a program should not establish CPU speed during initialization and then use that value for timing loops because the preemptive scheduling of MS OS/2 and future operating systems can "take away" the CPU at any time for arbitrary and unpredictable lengths of time. (In any case, time should not be wasted in a timing loop when other processes could be using system resources.)

Program timing

Programs must measure the passage of time carefully. They can use the system clock-tick interrupt while directly interfacing with the user, but no clock ticks will be seen by real mode programs when the user switches the screen interface to another program.

It is recommended that applications use the time-of-day system interface to determine elapsed time. To facilitate conversion to MS OS/2 protected mode, programs should encapsulate time-of-day or elapsed-time functions into subroutines.

BIOS

Avoid BIOS interrupt interfaces except for Interrupt 10H (the screen display functions) and Interrupt 16H (the keyboard functions). Interrupt 10H functions are contained in the MS OS/2 VIO package, and Interrupt 16H functions are in the MS OS/2 KBD package. Other BIOS interrupts provide functions that are available under MS OS/2 only in considerably modified forms.

Special operations

Uncommon, or special, operations and instructions can produce varied results, depending on the microprocessor. For example, when a "divide by 0" trap is taken on an 8086, the stack frame points to the instruction after the fault; when such action is taken on the 80286 and 80386, the return address points to the instruction that caused the fault. The effect of

pushing the SP register is different between the 80286 and the 80386 as well. See Appendix M: 8086/8088 Software Compatibility Issues. Write your program to avoid these problem areas.

Operating-System Issues

Basic to writing programs that will run on future operating systems is writing code that is not version specific. Incorporating special version-specific features in a program will virtually ensure that the program will be incompatible with future versions of MS-DOS and MS OS/2.

Following the guidelines below will not necessarily ensure your program's compatibility, but it will facilitate converting the program or using the Family API to produce a dual-mode binary program.

Filenames

MS-DOS versions 2 and 3 silently truncate a filename that is longer than eight characters or an extension that is longer than three characters. MS-DOS generates no error message when performing this task. In real mode, MS OS/2 also silently truncates a filename or extension that exceeds the maximum length; in protected mode, however, it does not. Therefore, a real mode application program needs to perform this truncating function. The program should check the length of the filenames that it generates or that it obtains from a user and refuse names that are longer than the eight-character maximum. This prevents improperly formatted names from becoming embedded in data and control files—a situation that could cause a protected mode version of the application to fail when it presents that invalid name to the operating system.

When you convert your program to protected mode API, remove the length-checking code; MS OS/2 will check the length and return an error code as appropriate. Future file systems will support longer filenames, so it's important that protected mode programs simply present filenames to the operating system, which is then responsible for judging their validity.

Other MS-DOS version 2 and 3 elements have fixed lengths, including the current directory path. To be upwardly compatible, your program should accept whatever length is provided by the user or returned from a system call and rely on MS OS/2 to return an error message if a length is inappropriate. The exception is filename length in real mode non-Family API programs: These programs should enforce the eight-character maximum because MS-DOS versions 2 and 3 fail to do so.

File truncation

Files are truncated by means of a zero-length write under MS-DOS versions 2 and 3; under MS OS/2 in protected mode, files are truncated with a special API. File truncation operations should be encapsulated in a special routine to facilitate conversion to MS OS/2 protected mode or the Family API.

File searches

MS-DOS versions 2 and 3 never close file-system searches (Find First File/Find Next File). The returned search contains the information necessary for MS-DOS to continue the search later, and if the search is never continued, no harm is done.

MS OS/2, however, retains the necessary search continuation information in an internal structure of limited size. For this reason, your program should not depend on more than about 10 simultaneous searches and it should be able to close searches when it is done. If your program needs to perform more than about 10 searches simultaneously, it should be able to close a search, restart it later, and advance to the place where the program left off, rather than depending on MS OS/2 to continue the search.

MS OS/2 further provides a Find Close function that releases the internal search information. Protected mode programs should use this call at the end of every search sequence. Because MS-DOS versions 2 and 3 have no such call, your program should call a dummy procedure by this name at the appropriate locations. Then you can convert your program to the protected mode API or to the Family API without reexamining your algorithms.

Note: Receiving a "No more files" return code from a search does not implicitly close the search; all search closes must be explicit.

The Family API allows only a single search at a time. To circumvent this restriction, code two different Find Next File routines in your program — one for MS OS/2 protected mode and one for MS-DOS real mode — and use the Family API function that determines the program's current environment to select the routine to execute.

MS-DOS calls

A program that uses only the Interrupt 21H functions listed below is guaranteed to work in the Compatibility Box of MS OS/2 and will be relatively easy to modify for MS OS/2 protected mode.

Function	Name
0DH	Disk Reset
0EH	Select Disk
19H	Get Current Disk
1AH	Set DTA Address
25H	Set Interrupt Vector
2AH	Get Date
2BH	Set Date
2CH	Get Time
2EH	Set/Reset Verify Flag
2FH	Get DTA Address

(more)

Function	Name
30H	Get MS-DOS Version Number
33H	Get/Set Control-C Check Flag
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Get/Set Current Country
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File
3FH	Read File or Device
40H	Write File or Device
41H	Delete File
42H	Move File Pointer
43H	Get/Set File Attributes
44H	IOCTL (all subfunctions)
45H	Duplicate File Handle
46H	Force Duplicate File Handle
47H	Get Current Directory
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
4BH	Load and Execute Program (EXEC)
4CH	Terminate Process with Return Code
4DH	Get Return Code of Child Process
4EH	Find First File
4FH	Find Next File
54H	Get Verify Flag
56H	Rename File
57H	Get/Set Date/Time of File
59H	Get Extended Error Information
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region

FCBs

FCBs are not supported in MS OS/2 protected mode. Use handle-based calls instead.

Interrupt calls

MS-DOS versions 2 and 3 use an interrupt-based interface; MS OS/2 protected mode uses a procedure-call interface. Write your code to accommodate this difference by encapsulating the interrupt-based interfaces into individual subroutines that can then easily be modified to use the MS OS/2 procedure-call interface.

System call register usage

The MS OS/2 procedure-call interface preserves all registers except AX and FLAGS. Write your program to assume that the contents of AX and the contents of any register modified by MS-DOS version 2 and 3 interrupt interfaces are destroyed at each system call, regardless of the success or failure of that call.

Flush/Commit calls

Your program should issue Flush/Commit calls where necessary — for example, after writing out the user's work file — but no more than necessary. Because MS OS/2 is multi-tasking, the floppy disk that contains the files to be flushed may not be in the drive. In such a case, MS OS/2 prompts the user to insert the proper floppy disk. As a result, too frequent flushes could generate a great many *Insert disk* messages and degrade the system's usability.

Seeks

Seeks to negative offsets and to devices also create compatibility issues.

To negative offsets

Your program should not attempt to seek to a negative file location. A negative seek offset is permissible as long as the sum of the seek offset and the current file position is positive. MS-DOS versions 2 and 3 allow seeking to a negative offset as long as you do not attempt to read or write the file at that offset. MS OS/2 and subsequent systems return an error code for negative net offsets.

On devices

Your program should not issue seeks to devices (such as AUX, COM, and so on). Doing so produces an error under MS OS/2.

Error codes

Because future releases of the operating system may return new error codes to system calls, you should write code that is open-ended about error codes — that is, write your program to deal with error codes beyond those currently defined. You can generally do this by including special handling for any codes that require special treatment, such as "File not found," and by taking a generic course of action for all other errors. The MS OS/2 protected mode API and the Family API have an interface that contains a message describing the error; this message can be displayed to the user. The interface also returns error classification information and a recommended action.

Multitasking concerns

Multitasking is a feature of MS OS/2 and will be a feature of all future versions of MS-DOS. The following guidelines apply to all programs, even to those written for MS-DOS version 3, because they may run in compatibility mode under MS OS/2.

Disabling interrupts

Do not disable interrupts, typically with the CLI instruction. The consequences of doing so depend on the environment.

In real mode programs under MS OS/2, disabling interrupts works normally but has a negative impact on the system's ability to maintain proper system throughput. Communications programs or networking applications might lose data. In a future version of real mode MS OS/2-80386, the operating system will disregard attempts to disable interrupts.

Protected mode programs under MS OS/2 can disable interrupts only in special Ring 2 segments. Disabling interrupts for longer than 100 microseconds might cause communications programs or networking applications to lose data or break connection. A future 80386-specific version of MS OS/2 will ignore attempts to disable interrupts in protected mode programs.

Measuring system resources

Do not attempt to measure system resources by exhausting them, and do not assume that because a resource is available at one time it will be available later. Remember: System resources are being shared with other programs.

For example, it is common for an MS-DOS version 3 application to request 1 MB of memory. The system cannot fulfill this request, so it returns the largest amount of memory available. The application then requests that amount of memory. Typically, applications do not even check for an error code from the second request. They routinely request all available memory because their creators knew that no other application could be in the system at the same time. This practice will work in real mode MS OS/2, although it is inefficient because MS OS/2 must allocate memory to a program that has no effective use for it. However, this practice will *not* work under MS OS/2 protected mode or under the Family API.

Another typical resource-exhaustion technique is opening files until an open is refused and then closing unneeded file handles. All applications, even those that run only in an MS OS/2 real mode environment, must use only the resources they need and not waste system resources; in a multitasking environment, other programs in the system usually need those resources.

Sharing rules

Because multiple programs can run under MS OS/2 simultaneously and because the system can be networked, conflicts can occur when two programs try to access the same file. MS OS/2 handles this situation with special file-sharing support. Although programs

ignorant of file-sharing rules can run in real mode, you should explicitly specify file-sharing rules in your program. This will reduce the number of file-access conflicts the user will encounter.

Miscellaneous guidelines

Do not use undocumented features of MS-DOS or undocumented fields such as those in the Find First File buffer. Also, do not modify or store your own values in such areas.

Maintain at least 2048 free bytes on the stack at all times. Future releases of MS-DOS may require extra stack space at system call and at interrupt time.

Print using conventional handle writes to the LPT device(s). For example:

```
fd = open("LPT1");  
write(fd, data, datalen);
```

Do not use Interrupt 17H (the IBM ROM BIOS printer services), writes to the *stdprn* handle (handle 3), or special-purpose Interrupt 21H functions such as 05H (Printer Output). These methods are not supported under MS OS/2 protected mode or in the Family API.

Do not use the MS-DOS standard handles *stdaux* and *stdprn* (handles 3 and 4); these handles are not supported in MS OS/2 protected mode. Use only *stdin* (handle 0), *stdout* (handle 1), and *stderr* (handle 2). Do use these latter handles where appropriate and avoid opening the CON device directly. Avoid Interrupt 21H Functions 03H (Auxiliary Input) and 04H (Auxiliary Output), which are polling operations on *stdaux*.

Summary

A tenet of MS OS/2 design was flexibility: Each component was constructed in anticipation of massive changes in a future release and with an eye toward existing versions of MS-DOS. Writing applications that are upwardly and backwardly compatible in such an environment is essential — and challenging. Following the guidelines in this article will ensure that your programs function appropriately in the MS-DOS/OS/2 operating-system family.

Gordon Letwin

Article 17

Windows

Microsoft Windows is an operating environment that runs under MS-DOS versions 2.0 and later. The current version of Windows, version 2.0, requires either a fixed disk or two double-sided floppy-disk drives, at least 320 KB of memory, and a video display board and monitor capable of graphics and a screen resolution of at least 640 (horizontal) by 200 (vertical) pixels. A fixed disk and 640 KB of memory provide the best environment for running Windows; a mouse or other pointing device is optional but recommended.

For the user, Windows provides a multitasking, graphics-based windowing environment for running programs. In this environment, users can easily switch among several programs and transfer data between them. Because programs specially designed to run under Windows usually have a consistent user interface, the time spent learning a new program is greatly diminished. Furthermore, the user can carry out command functions using only the keyboard, only the mouse, or some combination of the two. In some cases, Windows (and Windows applications) provides several different ways to execute the same command.

For the program developer, Windows provides a wealth of high-level routines that make it easy to incorporate menus, scroll bars, and dialog boxes (which contain controls, such as push buttons and list boxes) into programs. Windows' graphics interface is device independent, so programs developed for Windows work with every video display adapter and printer that has a Windows driver (usually supplied by the hardware manufacturer). Windows also includes features that facilitate the translation of programs into foreign languages for international markets.

When Windows is running, it shares responsibility for managing system resources with MS-DOS. Thus, programs that run under Windows continue to use MS-DOS function calls for all file input and output and for executing other programs, but they do not use MS-DOS for display or printer output, keyboard or mouse input, or memory management. Instead, they use functions provided by Windows.

Program Categories

Programs that run under Windows can be divided into three categories:

1. Programs specially designed for the Windows environment. Examples of such programs include Clock and Calculator, which come with Windows. Microsoft Excel is also specially designed for Windows. Other programs of this type (such as Aldus's Pagemaker) are available from software vendors other than Microsoft. Programs in this category cannot run under MS-DOS without Windows.
2. Programs designed to run under MS-DOS but that can usually be run in a window along with programs designed specially for Windows. These programs do not require

large amounts of memory, do not write directly to the display, do not use graphics, and do not alter the operation of the keyboard interrupt. They cannot use the mouse, the Windows application-program interface (such as menus and dialog boxes), or the graphics services that Windows provides. MS-DOS utilities, such as EDLIN and CHKDSK, are examples of programs in this category.

3. Programs designed to run under MS-DOS but that require large amounts of memory, write directly to the display, use graphics, or alter the operation of the keyboard interrupt. When Windows runs such a program, it must suspend operation of all other programs running in Windows and allow the program to use the full screen. In some cases, Windows cannot switch back to its normal display until the program terminates. Microsoft Word and Lotus 1-2-3 are examples of programs in this category.

The programs in categories 2 and 3 are sometimes called standard applications. To run one of these programs in Windows, the user must create a PIF file (Program Information File) that describes how much memory the program requires and how it uses the computer's hardware.

Although the ability to run existing MS-DOS programs under Windows benefits the user, the primary purpose of Windows is to provide an environment for specially designed programs that take full advantage of the Windows interface. This discussion therefore concentrates almost exclusively on programs written for the Windows 2.0 environment.

The Windows Display

Figure 17-1 shows a typical Windows display running several programs that are included with the retail version of Windows 2.0.

The display is organized as a desktop, with each program occupying one or more rectangular windows that, unlike the tiled (juxtaposed) windows typical of earlier versions, can be overlapped. Only one program is active at any time — usually the program that is currently receiving keyboard input. Windows displays the currently active program on top of (overlying) the others. Programs such as CLOCK and TERMINAL that are not active continue to run normally, but do not receive keyboard input.

The user can make another program active by pressing and releasing (clicking) the mouse button when the mouse cursor is positioned in the new program's window or by pressing either the Alt-Tab or Alt-Esc key combination. Windows then brings the new active program to the top.

Most Windows programs allow their windows to be moved to another part of the display or to be resized to occupy smaller or larger areas. Most of these programs can also be maximized to fill the entire screen or minimized — generally as a small icon displayed at the bottom of the screen — to occupy a small amount of display space.

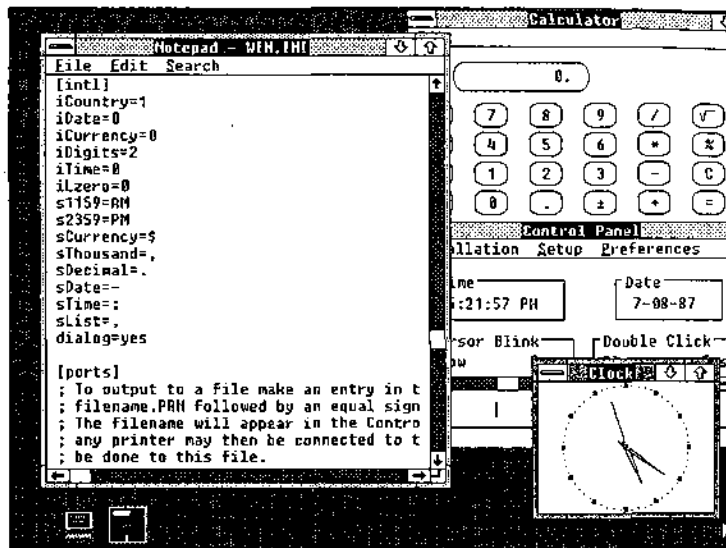


Figure 17-1. A typical Windows display.

Parts of the window

Figure 17-2 shows the Windows NOTEPAD program, with the different parts of the window identified. NOTEPAD is a small ASCII text editor limited to files of 16 KB. The various parts of the NOTEPAD window (similar to all Windows programs) are described in this section.

Title bar (or caption bar). The title bar identifies the program and, if applicable, the data file currently loaded into the program. For example, the NOTEPAD window shown in Figure 17-2 on the next page has the file WIN.INI loaded into memory. Windows uses different title-bar colors to distinguish the active window from inactive windows. The user can move a window to another part of the display by pressing the mouse button when the mouse pointer is positioned anywhere on the title bar and dragging (moving) the mouse while the button is pressed.

System-menu icon. When the user clicks a system-menu icon with the mouse (or presses Alt-Spacebar), Windows displays a system menu like that shown in Figure 17-3. (Most Windows programs have identical system menus.) The user selects a menu item in one of several ways: clicking on the item; moving the highlight bar to the item with the cursor-movement keys and then pressing Enter; or pressing the letter that is underlined in the menu item (for example, *n* for *Minimize*).

The keyboard combinations (Alt plus function key) at the right of the system menu are keyboard accelerators. Using a keyboard accelerator, the user can select system-menu options without first displaying the system menu.

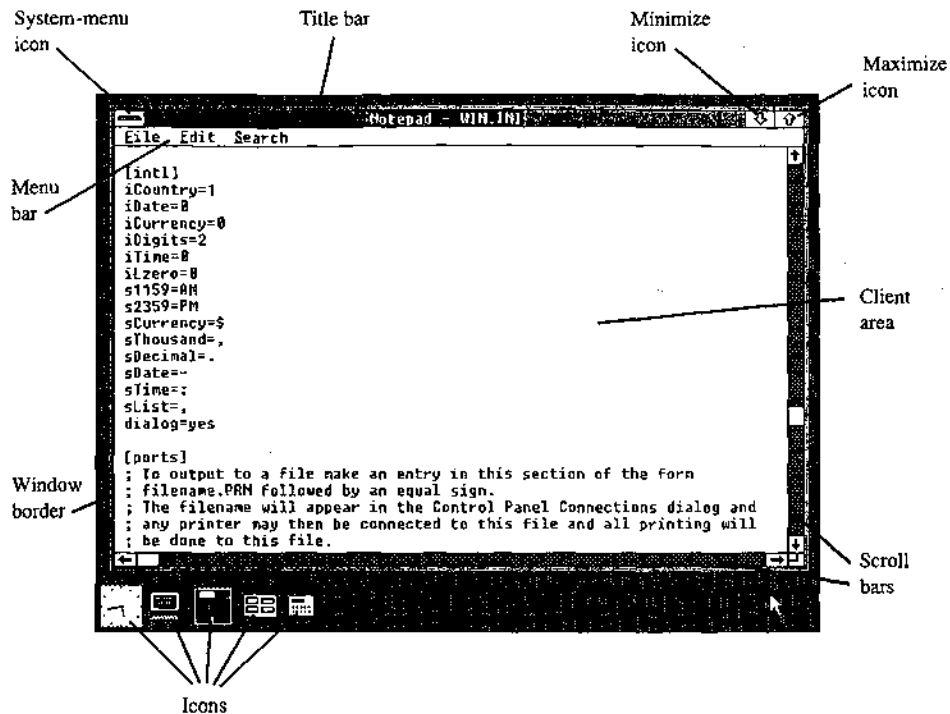


Figure 17-2. The Windows NOTEPAD program, with different parts of the display labeled.

The six options on the standard system menu are

- *Restore*: Return the window to its previous position and size after it has been minimized or maximized.
- *Move*: Allow the window to be moved with the cursor-movement keys.
- *Size*: Allow the window to be resized with the cursor-movement keys.
- *Minimize*: Display the window in its iconic form.
- *Maximize*: Allow the window to occupy the full screen.
- *Close*: End the program.

Windows displays an option on the system menu in grayed text to indicate that the option is not currently valid. In the system menu shown in Figure 17-3, for example, the *Restore* option is grayed because the window is not in a minimized or maximized form.

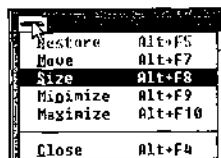


Figure 17-3. A system menu, displayed either when the user clicks the system-menu icon (top left corner) or presses Alt-Spacebar.


 — Restore icon

Figure 17-4. The restore icon, which replaces the maximize icon when a window is expanded to fill the entire screen.

Minimize icon. When the user clicks on the minimize icon with the mouse, Windows displays the program in its iconic form.

Maximize icon. Clicking on the maximize icon expands the window to fill the full screen. Windows then replaces the maximize icon with a restore icon (shown in Figure 17-4). Clicking on the restore icon restores the window to its previous size and position.

Programs that use a window of a fixed size (such as the CALC.EXE calculator program included with Windows) do not have a maximize icon.

Menu bar. The menu bar, sometimes called the program's main or top-level menu, displays keywords for several sets of commands that differ from program to program.

When the user clicks on a main-menu item with the mouse or presses the Alt key and the underlined letter in the menu text, Windows displays a pop-up menu for that item. The pop-up menu for NOTEPAD's keyword *File* is shown in Figure 17-5. Items are selected from a pop-up menu in the same way they are selected from the system menu.

A Windows program can display options on the menu in grayed text to indicate that they are not currently valid. The program can also display checkmarks to the left of pop-up menu items to indicate which of several options have been selected by the user.

In addition, items on a pop-up menu can be followed by an ellipsis (...) to indicate that selecting the item invokes a dialog box that prompts the user for additional information — more than can be provided by the menu.

Client area. The client area of the window is where the program displays data. In the case of the NOTEPAD program shown in Figure 17-2, the client area displays the file currently being edited. A program's handling of keyboard and mouse input within the client area depends on the type of work it does.

Scroll bars. Programs that cannot display all the data in a file within the client area of the window often have a horizontal scroll bar across the bottom and a vertical scroll bar down the right edge. Both types of scroll bars have a small, boxed arrow at each end to indicate the direction in which to scroll. In the NOTEPAD window in Figure 17-2, for example, clicking on the up arrow of the vertical scroll bar moves the data within the window down

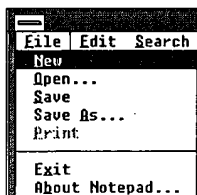


Figure 17-5. The NOTEPAD program's pop-up file menu.

one line. Clicking on the shaded part of the vertical scroll bar above the thumb (the box near the middle) moves the data within the client area of the window down one screen; clicking below the thumb moves the data up one screen. The user can also drag the thumb with the mouse to move to a relative position within the file.

Windows programs often include a keyboard interface (generally relying on the cursor-movement keys) to duplicate the mouse-based scroll-bar commands.

Window border. The window border is a thick frame surrounding the entire window. It is segmented into eight sections that represent the four sides and four corners of the window. The user can change the size of a window by dragging the window border with the mouse. Dragging a corner section moves two adjacent sides of the border.

When a program is maximized to fill the full screen, Windows does not draw the window border. Programs that use a window of a fixed size do not have a window border either.

Dialog boxes

When a pop-up menu is not adequate for all the command options a program requires, the program can display a dialog box. A dialog box is a pop-up window that contains various controls in the form of push buttons, check boxes, radio buttons, list boxes, and text and edit fields. Programmers can also design their own controls for use in dialog boxes. A user fills in a dialog box and then clicks on a button, such as *OK*, or presses Enter to indicate that the information can be processed by the program.

Most Windows programs use a dialog box to open an existing data file and load it into the program. The program displays the dialog box when the user selects the *Open* option on the *File* pop-up menu. The sample dialog box shown in Figure 17-6 is from the NOTEPAD program.

The list box displays a list of all valid disk drives, the subdirectories of the current directory, and all the filenames in the current directory, including the filename extension used by the program. (NOTEPAD uses the extension .TXT for its data files.) The user can scroll through this list box and change the current drive or subdirectory or select a filename with the keyboard or the mouse. The user can also perform these actions by typing the name directly into the edit field.

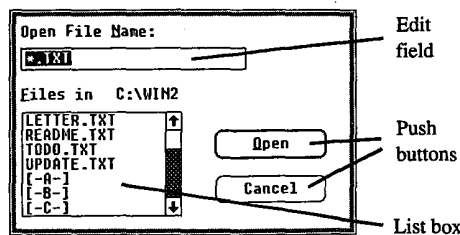


Figure 17-6. A dialog box from the NOTEPAD program, with parts labeled.

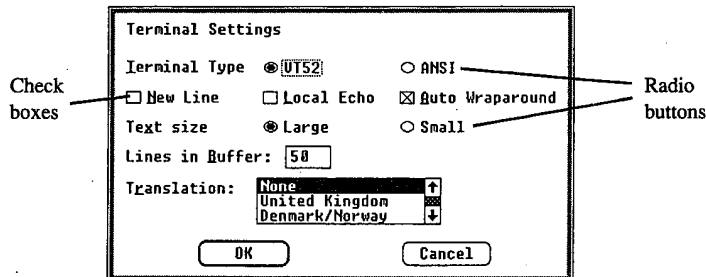


Figure 17-7. A dialog box from the *TERMINAL* program, with parts labeled.

Clicking the *Open* button (or pressing Enter) indicates to NOTEPAD that a file has been selected or that a new drive or subdirectory has been chosen (in this case, the program displays the files on the new drive or subdirectory). Clicking the *Cancel* button (or pressing Esc) tells NOTEPAD to close the dialog box without loading a new file.

Figure 17-7 shows a different dialog box — this one from the Windows *TERMINAL* communications program. The check boxes turn options on (indicated by an X) and off. The circular radio buttons allow the user to select from a set of mutually exclusive options.

Another, simple form of a dialog box is called a message box. This box displays one or more lines of text, an optional icon such as an exclamation point or an asterisk, and one or more buttons containing the words *OK*, *Yes*, *No*, or *Cancel*. Programs sometimes use message boxes for warnings or error messages.

The MS-DOS Executive

Within Windows, the MS-DOS Executive program (shown in Figure 17-8) serves much the same function as the *COMMAND.COM* program in the MS-DOS environment.

The top of the MS-DOS Executive client area displays all valid disk drives. The current disk drive is highlighted. Below or to the right of the disk drives is a display of the full path of the current directory. Below this is an alphabetic listing of all subdirectories in the current directory, followed by an alphabetic listing of all files in the current directory. Subdirectory names are displayed in boldface to distinguish them from filenames.

The user can change the current drive by clicking on the disk drive with the mouse or by pressing Ctrl and the key corresponding to the disk drive letter.

To change to one of the parent directories, the user double-clicks (clicks the mouse button twice in succession) on the part of the text string corresponding to the directory name. Pressing the Backspace key moves up one directory level toward the root directory. The user can also change the current directory to a child subdirectory by double-clicking on the subdirectory name in the list or by pressing the Enter key when the cursor highlight is on the subdirectory name. In addition, the menu also contains an option for changing the current directory.

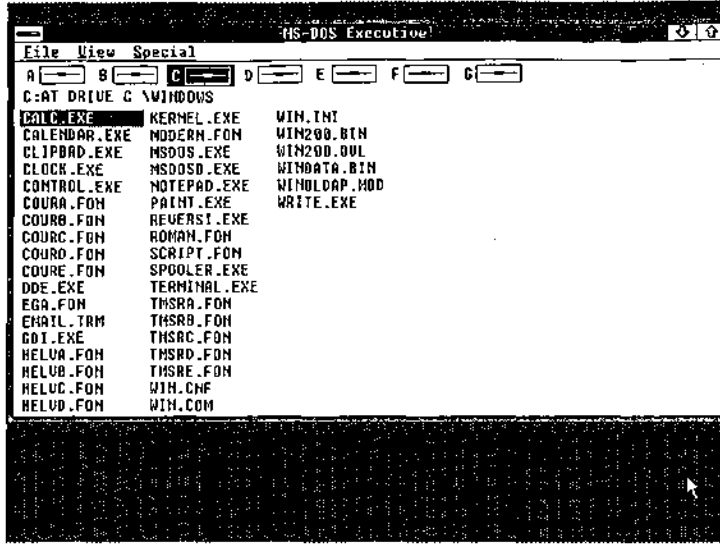


Figure 17-8. The MS-DOS Executive.

The user can run a program by double-clicking on the program filename, by pressing the Enter key when the highlight is on the program name, or by selecting it from a menu.

Other menu options allow the user to display the file and subdirectory lists in a variety of ways. A long format includes the same information displayed by the MS-DOS DIR command, or the user can choose to display a select group of files. Menu options also enable the user to specify whether the files should be listed in alphabetic order by filename, by filename extension, or by date or size.

The remaining options on the MS-DOS Executive menu allow the user to run programs; copy, rename, and delete files; format a floppy disk; change a volume name; make a system disk; create a subdirectory; and print a text file.

Other Windows Programs

Windows 2.0 also includes a number of application and utility programs. The application programs are CALC (a calculator), CALENDAR, CARDFILE (a database arranged as a series of index cards), CLOCK, NOTEPAD, PAINT (a drawing and painting program), REVERSI (a game), TERMINAL, and WRITE (a word processor).

The utility programs include

CLIPBRD. This program displays the current contents of the Clipboard, which is a storage facility that allows users to transfer data from one program to another.

CONTROL. The Control Panel utility allows the user to add or delete font files and printer drivers and to change the following: current printer, printer output port, communications parameters, date and time, cursor blink rate, screen colors, border width, mouse double-click time and options, and country-specific information, such as time and date formats. The Control Panel stores much of this information in the file named WIN.INI (Windows Initialization), so the information is available to other Windows programs.

PIFEDIT. The PIF editor allows the user to create or modify the PIFs that contain information about standard applications that have not been specially designed to run under Windows. This information allows Windows to adjust the environment in which the program runs.

SPOOLER. Windows uses the print-spooler utility to print files without suspending the operation of other programs. Most printer-directed output from Windows programs goes to the print spooler, which then prints the files while other programs run. SPOOLER enables the user to change the priority of print jobs or to cancel them.

The Structure of Windows

When programs run under MS-DOS, they make requests of the operating system through MS-DOS software interrupts (such as Interrupt 21H), through BIOS software interrupts, or by directly accessing the machine hardware.

When programs run under Windows, they use MS-DOS function calls only for file input and output and (more rarely) for executing other programs. Windows programs do not use MS-DOS function calls for memory management, keyboard input, display or printer output, or RS232 communications. Nor do Windows programs use BIOS routines or direct access to the hardware.

Instead, Windows provides application programs with access to more than 450 functions that allow programs to create and manipulate windows on the display; use menus, dialog boxes, and scroll bars; display text and graphics within the client area of a window; use the printer and RS232 communications port; and allocate memory.

The Windows modules

The functions provided by Windows are largely handled by three main modules named KERNEL, GDI, and USER. The KERNEL module is responsible for scheduling and multi-tasking, and it provides functions for memory management and some file I/O. The GDI module provides Windows' Graphics Device Interface functions, and the USER module does everything else.

The USER and GDI modules, in turn, call functions in various driver modules that are also included with Windows. Drivers control the display, printer, keyboard, mouse, sound, RS232 port, and timer. In most cases, these driver modules access the hardware of the computer directly. Windows includes different driver files for various hardware configurations. Hardware manufacturers can also develop Windows drivers specifically for their products.

A block diagram showing the relationships of an application program, the KERNEL, USER, and GDI modules, and the driver modules is shown in Figure 17-9. The figure shows each of these modules as a separate file — KERNEL, USER, and GDI have the extension .EXE; the driver files have the extension .DRV. Some program developers install Windows with these modules in separate files, as in Figure 17-9, but most users install Windows by running the SETUP program included with Windows.

SETUP combines most of these modules into two larger files called WIN200.BIN and WIN200.OVL. Printer drivers are a little different from the other driver files, however, because the Windows SETUP program does not include them in WIN200.BIN and WIN200.OVL. The name of the driver file identifies the printer. For example, IBMGRX.DRV is a printer driver file for the IBM Personal Computer Graphics Printer.

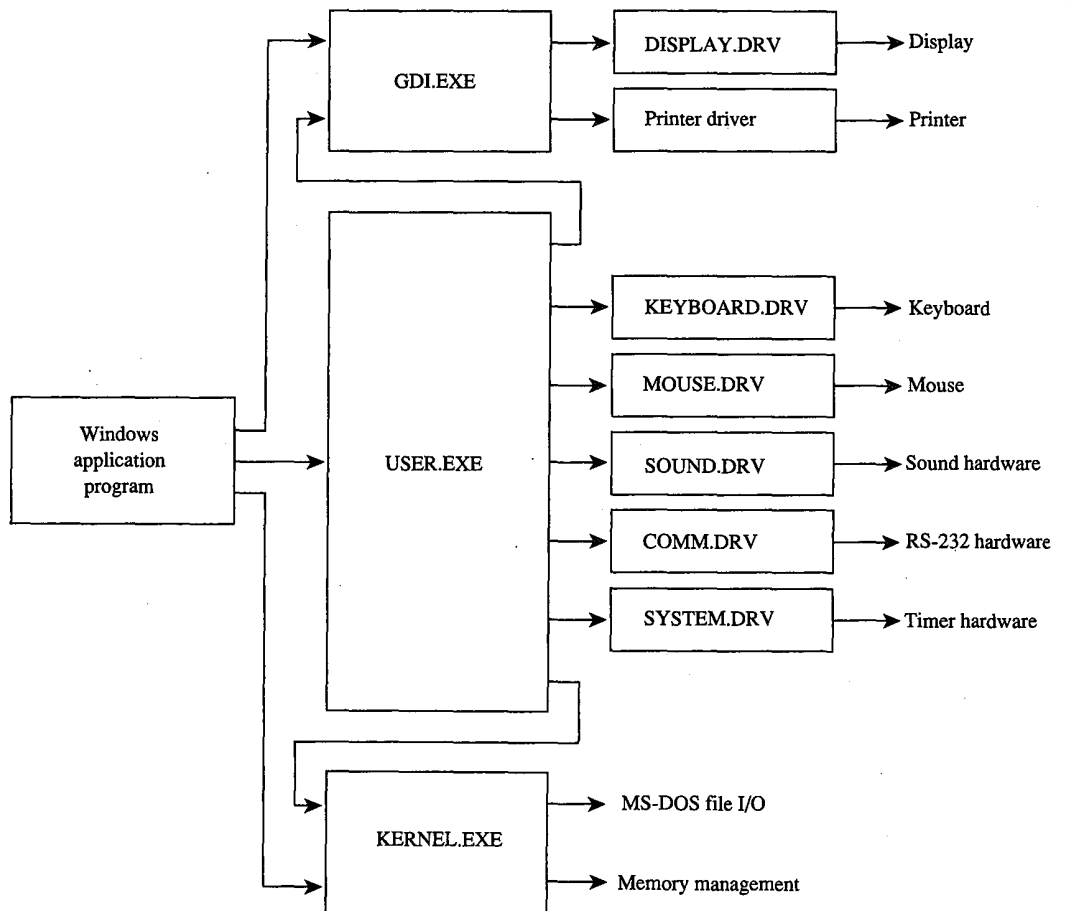


Figure 17-9. A simplified block diagram showing the relationships of an application program, Windows modules (GDI, USER, and KERNEL), driver modules, and system hardware.

The diagram in Figure 17-9 is somewhat simplified. In reality, a Windows application program can also make direct calls to the `KEYBOARD.DRV` and `SOUND.DRV` modules, and `USER.EXE` calls the `DISPLAY.DRV` and printer driver modules directly. The `GDI.EXE` module and driver modules can also call routines in `KERNEL.EXE`, and drivers sometimes call routines in `SYSTEM.DRV`.

Also, Figure 17-9 omits the various font files provided with Windows, the `WIN.INI` file that contains Windows initialization information and user preferences, and the files `WINOLDAP.MOD` and `WINOLDAP.GRB`, which Windows uses to run standard MS-DOS applications.

Libraries and programs

The `USER.EXE`, `GDI.EXE`, and `KERNEL.EXE` files, all driver files with the extension `.DRV`, and all font files with the extension `.FON` are called Windows libraries or, sometimes, dynamic link libraries to distinguish them from Windows programs. Programs and libraries both use a file format called the New Executable format.

From the user's perspective, a Windows program and a Windows library are very different. The user cannot run a Windows library directly. Windows loads a part of a library into memory only when a program needs to use a function that the library provides.

The user can also run multiple instances of the same Windows program. Windows uses the same code segments for the different instances but creates a unique data segment for each. Windows never runs multiple instances of a Windows library.

From the programmer's perspective, a Windows program is a task that creates and manages windows on the display. Libraries are modules that assist the task. A programmer can write additional library modules, which one or more programs can use. For the developer, one important distinction between programs and libraries is that a Windows library does not have its own stack; instead, the library uses the stack of the program that calls the routine in the library.

The New Executable format used for both programs and libraries gives Windows much more information about the module than is provided by the current MS-DOS `.EXE` format. In particular, the module contains information that allows Windows to make links between program modules and library modules when a program is run.

When a module (such as a library) contains functions that can be called from another module (such as a program), the functions are said to be exported from the module that contains them. Each exported function in a module is identified either by a name (generally the name of the function) or by an ordinal (positive) number. A list of all exported functions in a module is included in the New Executable format header section of the module.

Conversely, when a module (such as a program) contains code that calls a function in another module (such as a library), the function is said to be imported to the module that makes the call. This call appears in the `.EXE` file as an unresolved reference to an external function. The New Executable format identifies the module and the function name or ordinal number that the call references.

When Windows loads a program or a library into memory, it must resolve all calls the module makes to functions in other modules. Windows does this by inserting the addresses of the functions into the code—a process called dynamic linking.

For example, many Windows programs use the function `TextOut` to display text in the client area. In the code segment of the program's `.EXE` file, a call to `TextOut` appears as an unresolved far (intersegment) call. The code segment's relocation table shows that this call is to an imported function in the GDI module identified by the ordinal number 33. The header section of the GDI module lists `TextOut` as an exported function with the ordinal number 33. When Windows loads the program, it resolves all references to `TextOut` by inserting the address of the function into the program's code segment in each place where `TextOut` is called.

Although Windows programs reference many functions that are exported from the standard Windows libraries, Windows programs also often include at least one exported function, called a window function. While the program is running, Windows calls this function to pass messages to the program's window. *See* *The Structure of a Windows Program* below.

Memory Management

Windows' memory management is based on the segmented-memory architecture of the Intel 8086 family of microprocessors. The memory space controlled by Windows is divided into segments of various lengths. Windows uses separate segments for nearly everything kept in memory—such as the code and data segments of programs and libraries—and for resources, such as fonts and bitmaps.

Windows programs and libraries contain one or more code segments, which are usually both movable and discardable. Windows can move a code segment in memory in order to consolidate free memory space. It can also discard a code segment from memory and later reload the code segment from the program's or library's `.EXE` file when it is needed again. This capability is called demand loading.

Windows programs usually contain only one data segment; Windows libraries are limited to one data segment. In most cases, Windows can move data segments in memory. However, it cannot usually discard data segments, because they can contain data that changes after the program begins executing. When a user runs multiple copies of a program, the different instances share the same code segments but have separate data segments.

The use of movable and discardable segments allows Windows to run several large programs in a memory space that might be inadequate for even one of the programs if the entire program were kept in memory, as is typical under MS-DOS without Windows. The ability of Windows to use memory in this way is called memory overcommitment.

The moving and discarding of code segments requires Windows to make special provisions so that intersegment calls continue to reference the correct address when a code

segment is moved. These provisions are another part of dynamic linking. When Windows resolves a far call from one code segment to a function in another code segment that is movable and discardable, the call actually references a fixed area of memory. This fixed area of memory contains a small section of code called a thunk. If the code segment containing the function is currently in memory, the thunk simply jumps to the function. If the code segment with the function is not currently in memory, the thunk calls a loader that loads the segment into memory. This process is called dynamic loading. When Windows moves or discards a code segment, it must alter the thunks appropriately.

Windows and Windows programs generally reference data structures stored in Windows' memory space by using 16-bit unsigned integers known as handles. The data structure that a handle references can be movable and discardable, so when Windows or the Windows program needs to access the data directly, it must lock the handle to cause the data to become fixed in memory. The function that locks the segment returns a pointer to the program.

During the time the handle is locked, Windows cannot move or discard the data. The data can then be referenced directly with the pointer. When Windows (or the Windows program) finishes using the data, it unlocks the segment so that it can be moved (or in some cases discarded) to free up memory space if necessary.

Programmers can choose to allocate nonmovable data segments, but the practice is not recommended, because Windows cannot relocate the segments to make room for segments required by other programs.

The Structure of a Windows Program

During development, a Windows program includes several components that are combined later into a single executable file with the extension .EXE for execution under Windows. Although the Windows executable file has the same .EXE filename extension as MS-DOS executable files, the format is different. Among other things, the New Executable format includes Windows-specific information required for dynamic linking and the discarding and reloading of the program's code segments.

Programmers generally use C, Pascal, or assembly language to create applications specially designed to run under Windows. Also required are several header files and development tools, which are included in the Microsoft Windows Software Development Kit.

The Microsoft Windows Software Development Kit

The Windows Software Development Kit contains reference material, a special linker (LINK4), the Windows Resource Compiler (RC), special versions of the SYMDEB and CodeView debuggers, header files, and several programs that aid development and testing. These programs include

- DIALOG: Used for creating dialog boxes.
- ICONEDIT: Used for creating a program's icon, customized cursors, and bitmap images.

- FONTEDIT: Used for creating customized fonts derived from an existing font file with the extension .FNT.
- HEAPWALK: Used for displaying the organization of code and data segments in Windows' memory space and for testing programs under low memory conditions.
- SHAKER: Used for randomly allocating memory to force segment movement and discarding. SHAKER tests a program's response to movement in memory and is useful for exposing program bugs involving pointers to unlocked segments.

The Windows Software Development Kit also provides several *include* and header files that contain declarations of all Windows functions, definitions of many macro identifiers that the programmer can use, and structure definitions. Import libraries included in the kit allow LINK4 to resolve calls to Windows functions and to prepare the program's .EXE file for dynamic linking.

Work with the Windows Software Development Kit requires one of the following compilers or assemblers:

- Microsoft C Compiler version 4.0 or later
- Microsoft Pascal Compiler version 3.31 or later
- Microsoft Macro Assembler version 4.0 or later

Other software manufacturers also provide compilers that are suitable for compiling Windows programs.

Components of a Windows program

The discussion in this section is illustrated by a program called SAMPLE, which displays the word *Windows* in its client area. In response to a menu selection, the program

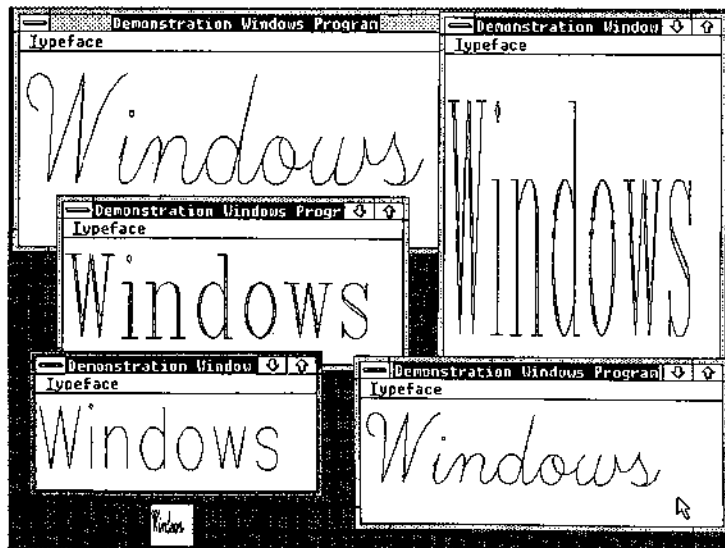


Figure 17-10. A display produced by the example program SAMPLE.

displays this text in any of the three vector fonts — Script, Modern, and Roman — that are included with Windows. Sometimes also called stroke or graphics fonts, these vector fonts are defined by a series of line segments, rather than by the pixel patterns that make up the more common raster fonts. The SAMPLE program picks a font size that fits the client area.

Figure 17-10 shows several instances of this program running under Windows.

Five separate files go into the making of this program:

1. Source-code file: This is the main part of the program, generally written in C, Pascal, or assembly language. The SAMPLE program was written in C, which is the most popular language for Windows programs because of its flexibility in using pointers and structures. The SAMPLE.C source-code file is shown in Figure 17-11.

```

/* SAMPLE.C -- Demonstration Windows Program */

#include <windows.h>
#include "sample.h"

long FAR PASCAL WndProc (HWND, unsigned, WORD, LONG) ;

int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
HANDLE      hInstance, hPrevInstance ;
LPSTR      lpszCmdLine ;
int        nCmdShow ;
{
WNDCLASS   wndclass ;
HWND       hWnd ;
MSG        msg ;
static char szAppName [] = "Sample" ;

        /*-----*/
        /* Register the Window Class */
        /*-----*/

if (!hPrevInstance)
{
    wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc     = WndProc ;
    wndclass.cbClsExtra     = 0 ;
    wndclass.cbWndExtra     = 0 ;
    wndclass.hInstance      = hInstance ;
    wndclass.hIcon          = NULL ;
    wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground  = GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName   = szAppName ;
    wndclass.lpszClassName  = szAppName ;

    RegisterClass (&wndclass) ;
}

```

Figure 17-11. The SAMPLE.C source code.

(more)

```

/*-----*/
/* Create the window and display it */
/*-----*/

hWnd = CreateWindow (szAppName, "Demonstration Windows Program",
                    WS_OVERLAPPEDWINDOW,
                    (int) CW_USEDEFAULT, 0,
                    (int) CW_USEDEFAULT, 0,
                    NULL, NULL, hInstance, NULL) ;

ShowWindow (hWnd, nCmdShow) ;
UpdateWindow (hWnd) ;

/*-----*/
/* Stay in message loop until a WM_QUIT message */
/*-----*/

while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
}

long FAR PASCAL WndProc (hWnd, iMessage, wParam, lParam)
HWND          hWnd ;
unsigned      iMessage ;
WORD          wParam ;
LONG          lParam ;
{
    PAINTSTRUCT ps ;
    HFONT        hFont ;
    HMENU        hMenu ;
    static short xClient, yClient, nCurrentFont = IDM_SCRIPT ;
    static BYTE  cFamily [] = { FF_SCRIPT, FF_MODERN, FF_ROMAN } ;
    static char  *szFace [] = { "Script", "Modern", "Roman" } ;

    switch (iMessage)
    {

        /*-----*/
        /* WM_COMMAND message: Change checkmarked font */
        /*-----*/

        case WM_COMMAND:
            hMenu = GetMenu (hWnd) ;
            CheckMenuItem (hMenu, nCurrentFont, MF_UNCHECKED) ;
            nCurrentFont = wParam ;
            CheckMenuItem (hMenu, nCurrentFont, MF_CHECKED) ;
            InvalidateRect (hWnd, NULL, TRUE) ;
            break ;
    }
}

```

Figure 17-11. Continued.

(more)

```

/*-----*/
/* WM_SIZE message: Save dimensions of window */
/*-----*/

case WM_SIZE:
    xClient = LOWORD (lParam) ;
    yClient = HIWORD (lParam) ;
    break ;

/*-----*/
/* WM_PAINT message: Display "Windows" in Script */
/*-----*/

case WM_PAINT:
    BeginPaint (hWnd, &ps) ;

    hFont = CreateFont (yClient, xClient / 8,
                        0, 0, 400, 0, 0, 0, OEM_CHARSET,
                        OUT_STROKE_PRECIS, OUT_STROKE_PRECIS,
                        DRAFT_QUALITY, (BYTE) VARIABLE_PITCH,
                        cFamily [nCurrentFont - IDM_SCRIPT],
                        szFace [nCurrentFont - IDM_SCRIPT]) ;

    hFont = SelectObject (ps.hdc, hFont) ;
    TextOut (ps.hdc, 0, 0, "Windows", 7) ;

    DeleteObject (SelectObject (ps.hdc, hFont)) ;
    EndPaint (hWnd, &ps) ;
    break ;

/*-----*/
/* WM_DESTROY message: Post Quit message */
/*-----*/

case WM_DESTROY:
    PostQuitMessage (0) ;
    break ;

/*-----*/
/* Other messages: Do default processing */
/*-----*/

default:
    return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
}
return 0L ;
}

```

Figure 17-11. Continued.