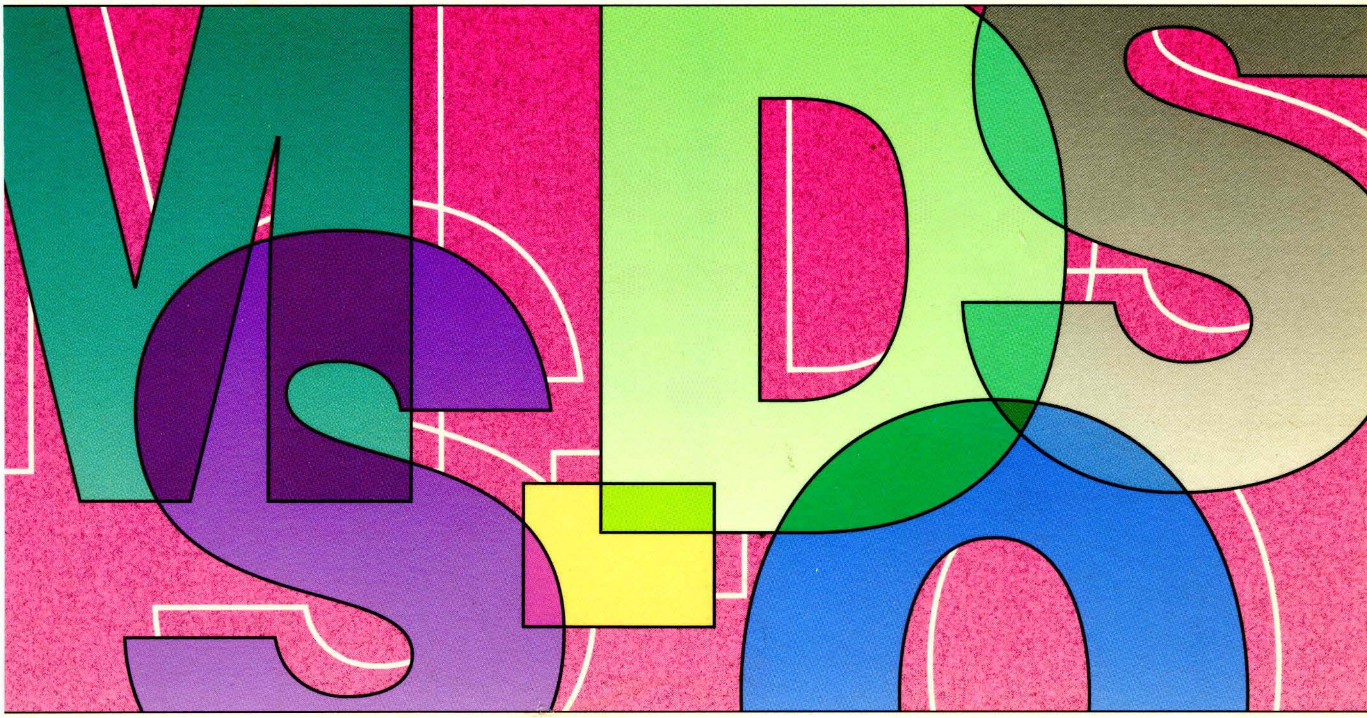
Complete and Unabridged

The MS-DOS® Encyclopedia



Microsoft PRESS®

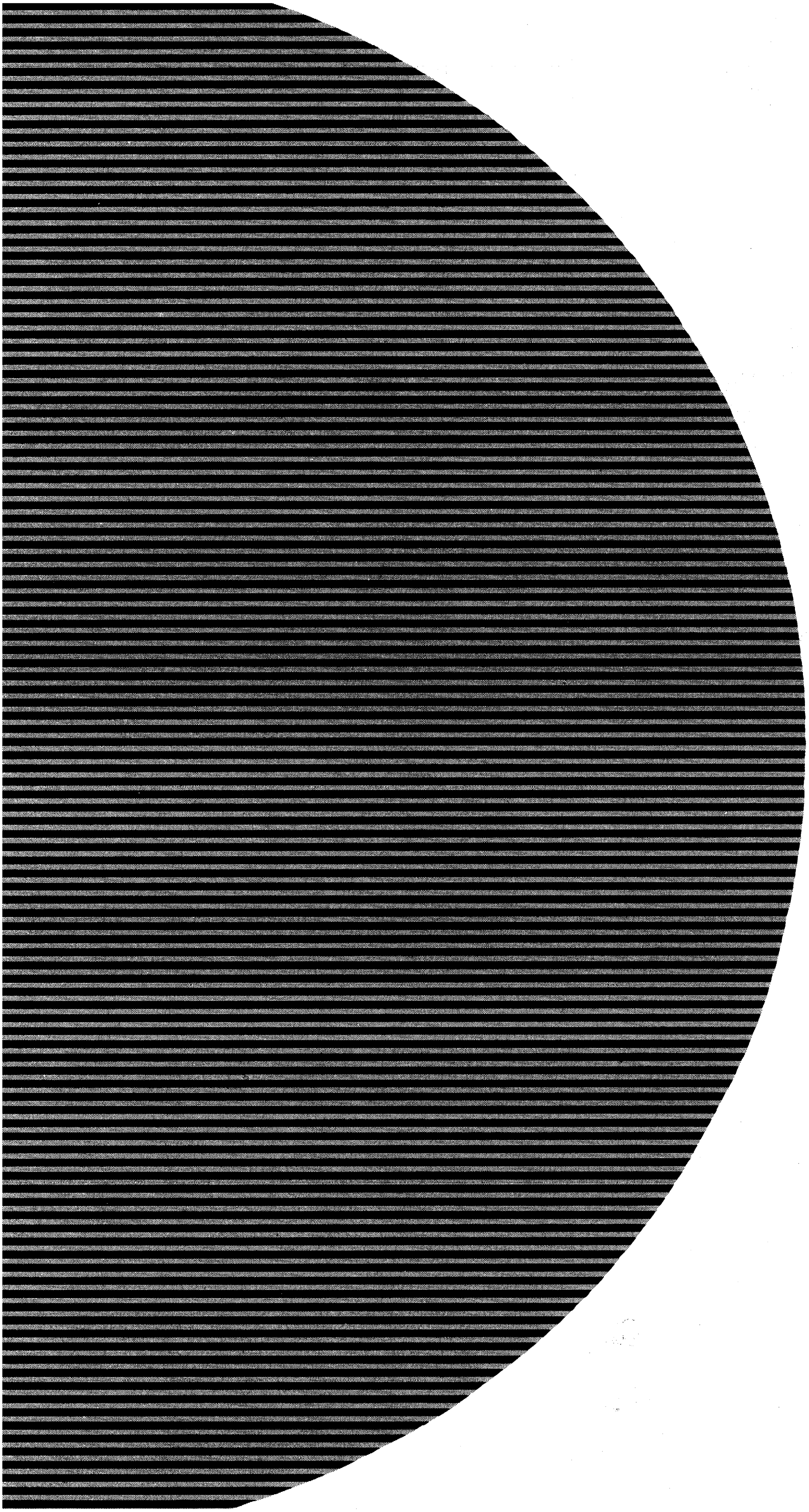
Foreword, Bill Gates

General Editor, Ray Duncan

The

MS-DOS®

Encyclopedia



The

MS-DOS®

Encyclopedia

Microsoft® PRESS

Microsoft Press
Redmond, Washington
1988

Ray Duncan, General Editor

Foreword by Bill Gates

Published by
Microsoft Press
A Division of Microsoft Corporation
16011 NE 36th Way, Box 97017, Redmond, Washington 98073-9717

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.

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

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

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. Brunsman

Editorial Coordinator: Sarah Hersack

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

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

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

Index: Shane-Armstrong Information Services

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

Marketing and Sales Director: James Brown

Director of Production: Christopher D. Banks

Publisher: Min S. Yee

Contributors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Special Technical Advisor

Mark Zbikowski

Technical Advisors

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

Contents

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 multitasking 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

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%.

Introduction

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

Organization of the Book

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

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

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

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

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

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.

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

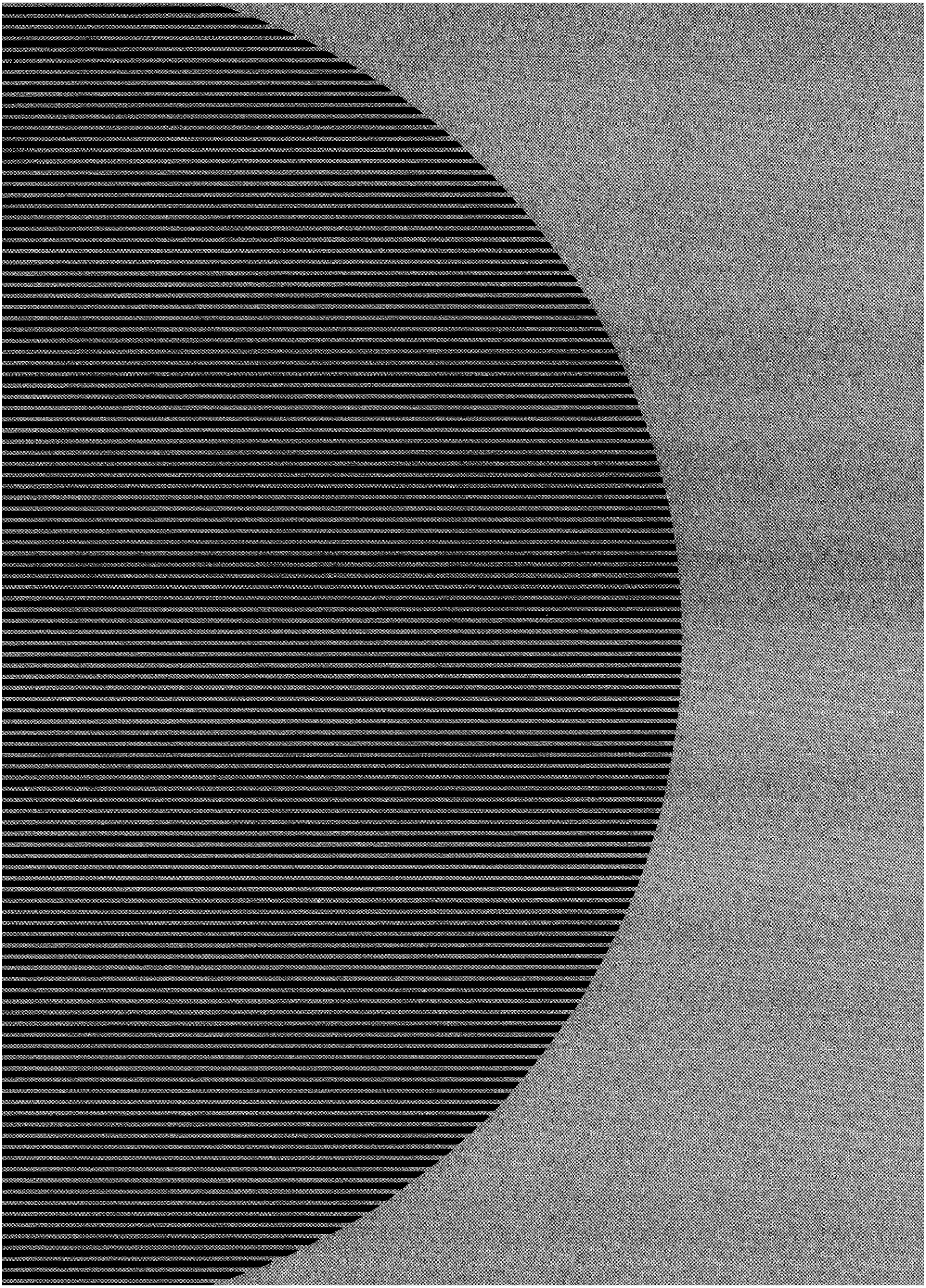
Although not an official IBM name, the term *PC-DOS* in this book means the IBM implementation of MS-DOS. If PC-DOS is referenced and the information differs from that for the related MS-DOS version, the PC-DOS version number is included. To avoid confusion, the term *DOS* is never used without a modifier.

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 I
The Development of MS-DOS

The Development of MS-DOS

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

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

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

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

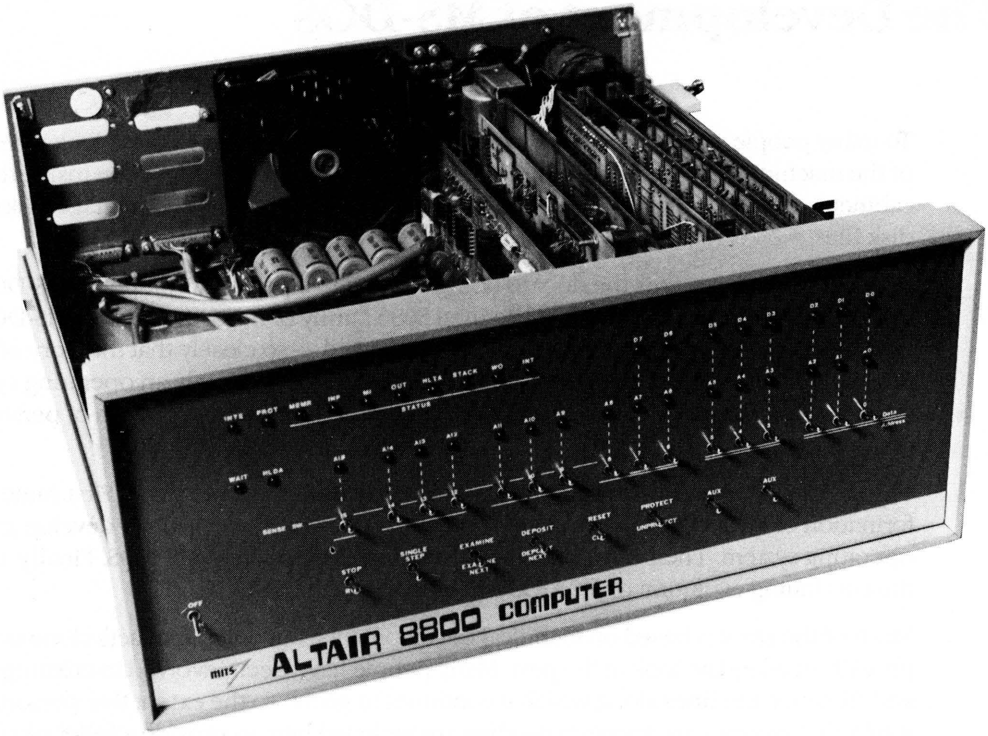
Before MS-DOS

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

Microsoft and BASIC

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

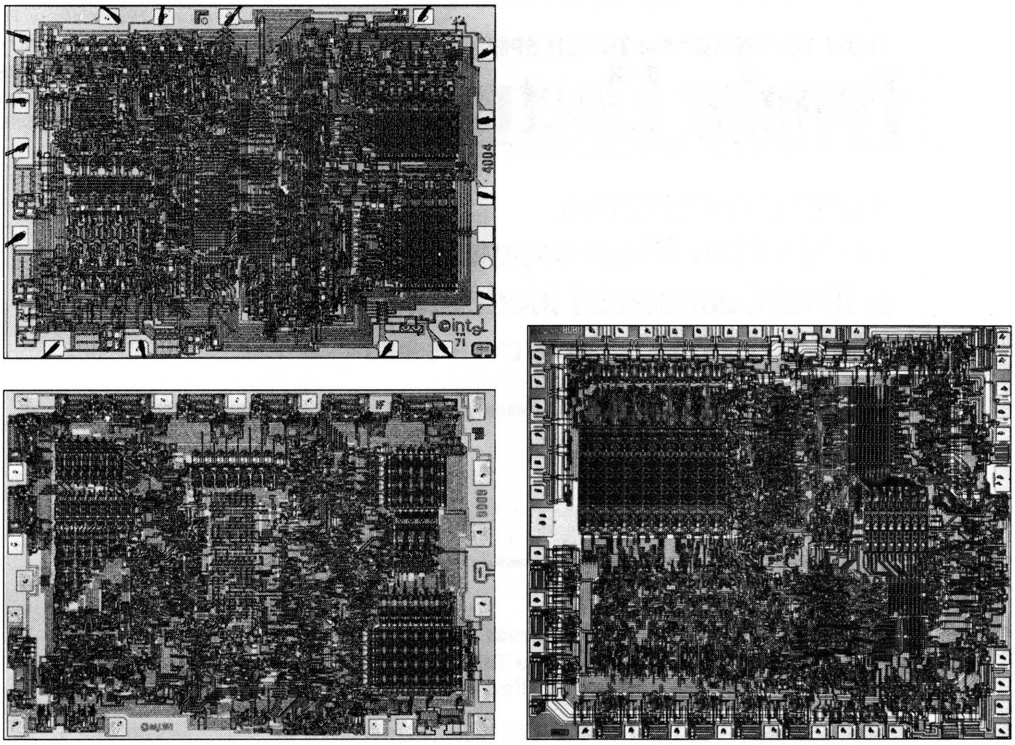
Before Microsoft even became a company, its founders, Paul Allen and Bill Gates, developed a version of BASIC for a revolutionary small computer named the Altair, which was introduced in January 1975 by Micro Instrumentation Telemetry Systems (MITS) of



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

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

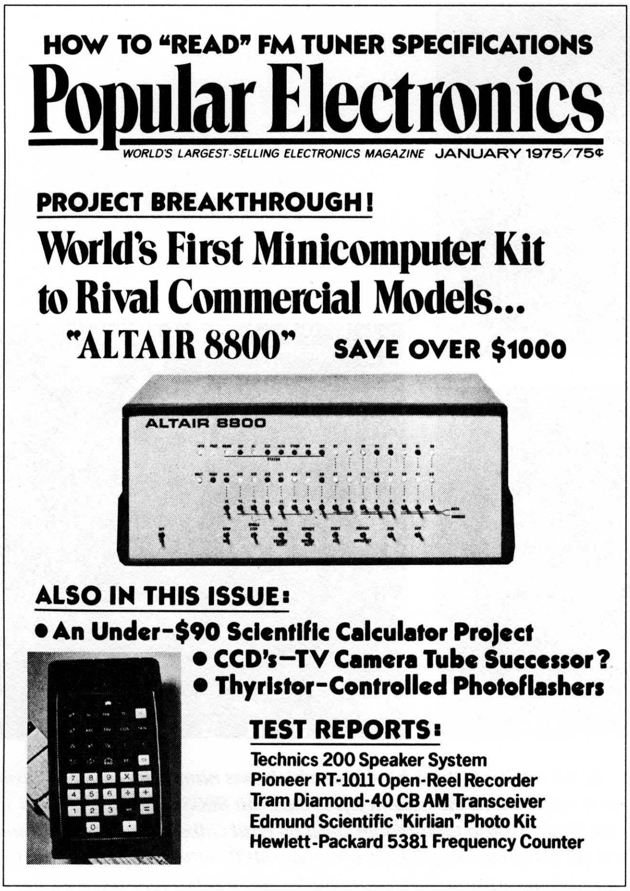
Now, given the hindsight of a little more than a decade of microcomputing history, it is easy to see that the Altair's combination of small size and affordability was the thin edge of a wedge that, in just a few years, would move everyday computing power away from impersonal monoliths in climate-controlled rooms and onto the desks of millions of people. In 1975, however, the computing environment was still primarily a matter of data processing for specialists rather than personal computing for everyone. Thus when 4 KB



Intel's 4004, 8008, and 8080 chips. At the top left is the 4-bit 4004, which was named for the approximate number of old-fashioned transistors it replaced. At the bottom left is the 8-bit 8008, which addressed 16 KB of memory; this was the chip used in the Traf-O-Data tape-reader built by Paul Gilbert. At the right is the 8080, a faster 8-bit chip that could address 64 KB of memory. The brain of the MITS Altair, the 8080 was, in many respects, the chip on which the personal computing industry was built. The 4004 and 8008 chips were developed early in the 1970s; the 8080 appeared in 1974.

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

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



HOW TO "READ" FM TUNER SPECIFICATIONS

Popular Electronics

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

PROJECT BREAKTHROUGH!

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

"ALTAIR 8800" SAVE OVER $1000

ALSO IN THIS ISSUE:

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

TEST REPORTS:

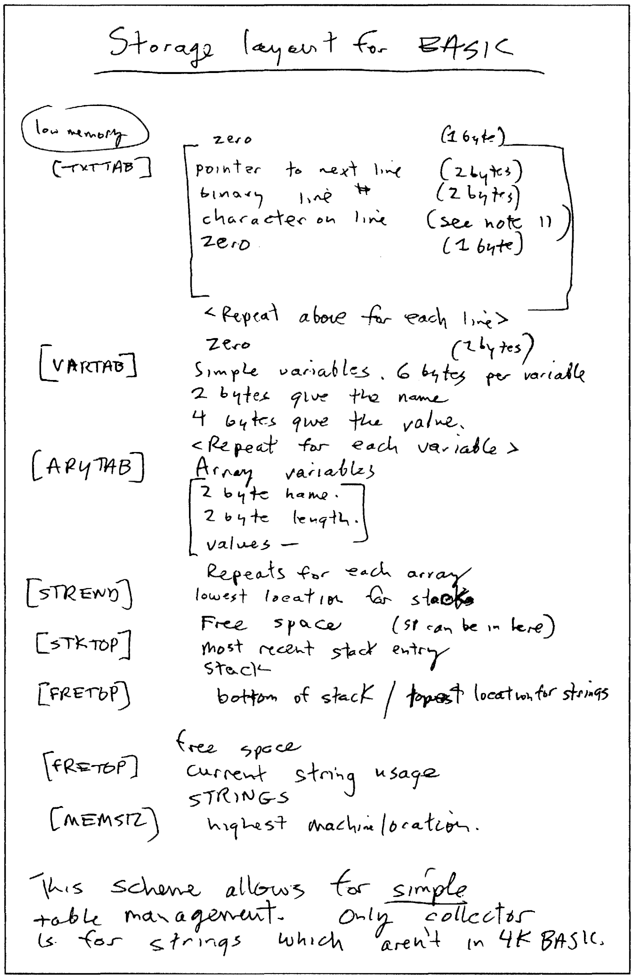
Technics 200 Speaker System
Pioneer RT-1011 Open-Reel Recorder
Tram Diamond-40 CB AM Transceiver
Edmund Scientific "Kirlian" Photo Kit
Hewlett-Packard 5381 Frequency Counter

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

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

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

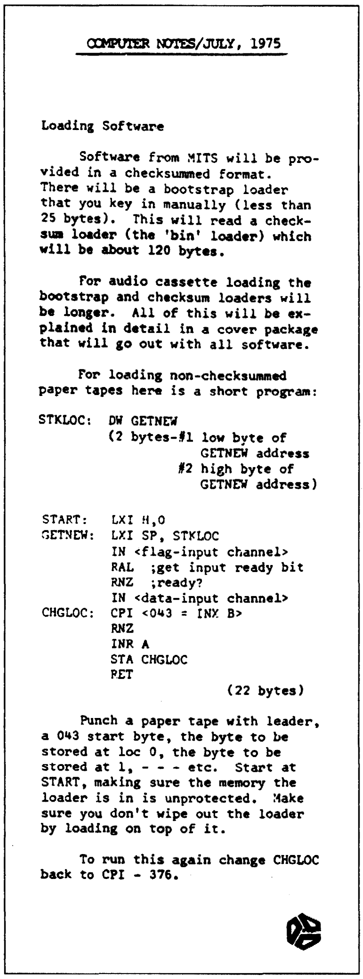
Though not a direct forerunner of MS-DOS, Altair BASIC, like the machine for which it was developed, was a landmark product in the history of personal computing. On another level, Altair BASIC was also the first link in a chain that led, somewhat circuitously, to Tim Paterson and the disk operating system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



Storage layout for BASIC

```
low memory
[TXTTAB]     zero                              (1 byte)
             pointer to next line              (2 bytes)
             binary line #                     (2 bytes)
             character on line                 (see note 1)
             zero                              (1 byte)
             <Repeat above for each line>
             zero                              (2 bytes)
[VARTAB]     Simple variables. 6 bytes per variable
             2 bytes give the name
             4 bytes give the value.
             <Repeat for each variable>
[ARYTAB]     Array variables
             2 byte name.
             2 byte length.
             values —
             Repeats for each array
[STREND]     lowest location for stack
             Free space       (SP can be in here)
[STKTOP]     most recent stack entry
             Stack
[FRETOP]     bottom of stack / ~~topest~~ location for strings
             free space
[FRETOP]     current string usage
             STRINGS
[MEMSIZ]     highest machine location.
```

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



COMPUTER NOTES/JULY, 1975

Loading Software

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

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

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

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

START:   LXI H,0
GETNEW:  LXI SP, STKLOC
         IN <flag-input channel>
         RAL  ;get input ready bit
         RNZ  ;ready?
         IN <data-input channel>
CHGLOC:  CPI <043 = INX B>
         RNZ
         INR A
         STA CHGLOC
         RET
                          (22 bytes)
```

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

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

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

From paper tape to disk

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

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

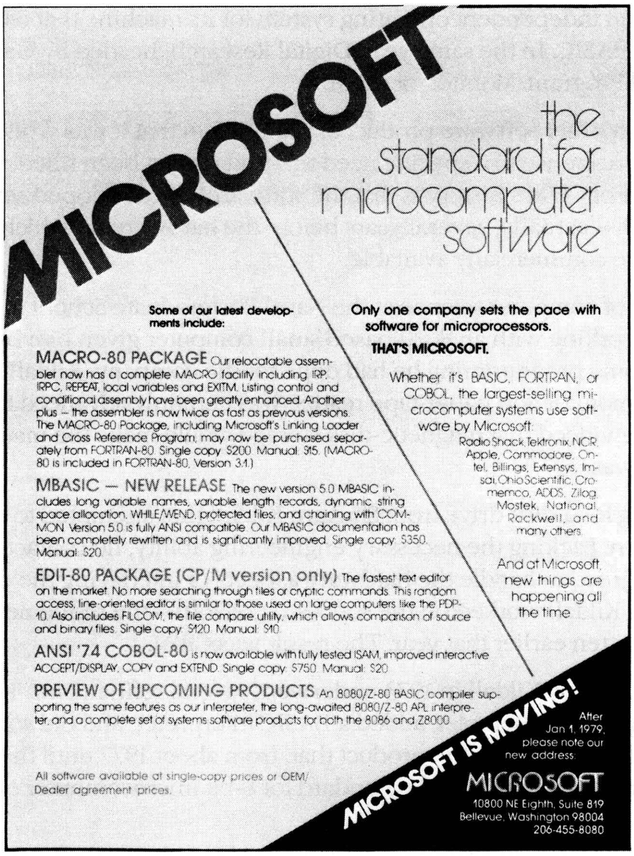
Gates, still at Harvard University, agreed to write this version of BASIC for MITS. He went to Albuquerque and, as has often been recounted, checked into the Hilton Hotel with a stack of yellow legal pads. Five days later he emerged, yellow pads filled with the code for the new version of BASIC. Arriving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

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

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

M-DOS

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



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 IRP, IRPC, REPEAT, local variables and EXITM. 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: $15. (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/WEND, protected files, and chaining with COMMON. Version 5.0 is fully ANSI compatible. Our MBASIC documentation has been completely rewritten and is significantly improved. Single copy: $350. Manual: $20.

EDIT-80 PACKAGE (CP/M version only) The fastest text editor on the market. No more searching through files or cryptic commands. This random access, line-oriented editor is similar to those used on large computers like the PDP-10. Also includes FILCOM, the file compare utility, which allows comparison of source and binary files. Single copy: $120. Manual: $10.

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

PREVIEW OF UPCOMING PRODUCTS An 8080/Z-80 BASIC compiler supporting the same features as our interpreter, the long-awaited 8080/Z-80 APL interpreter, and a complete set of systems software products for both the 8086 and Z8000.

All software available at single-copy prices or OEM/Dealer agreement prices.

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, Tektronix, NCR, Apple, Commodore, Ontel, Billings, Extensys, Imsai, Ohio Scientific, Cromemco, ADDS, Zilog, Mostek, National, Rockwell, and many others.

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

MICROSOFT IS MOVING!

After Jan 1, 1979, please note our new address:

MICROSOFT
10800 NE Eighth, Suite 819
Bellevue, Washington 98004
206-455-8080

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.

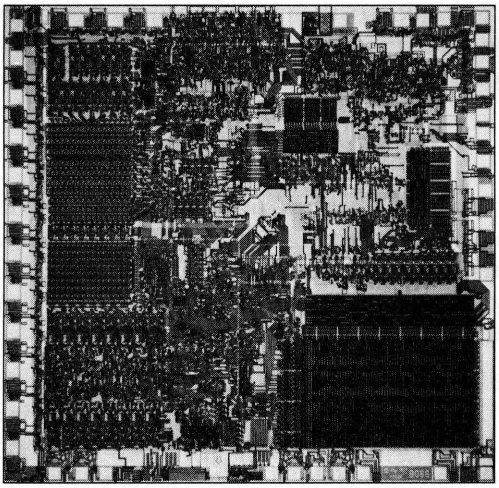
Trading some programming for a disk drive from Shugart, Kildall first attempted to build a drive controller on his own. Lacking the necessary engineering ability, he contacted a friend, John Torode, who agreed to handle the hardware aspects of interfacing the computer and the disk drive while Kildall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

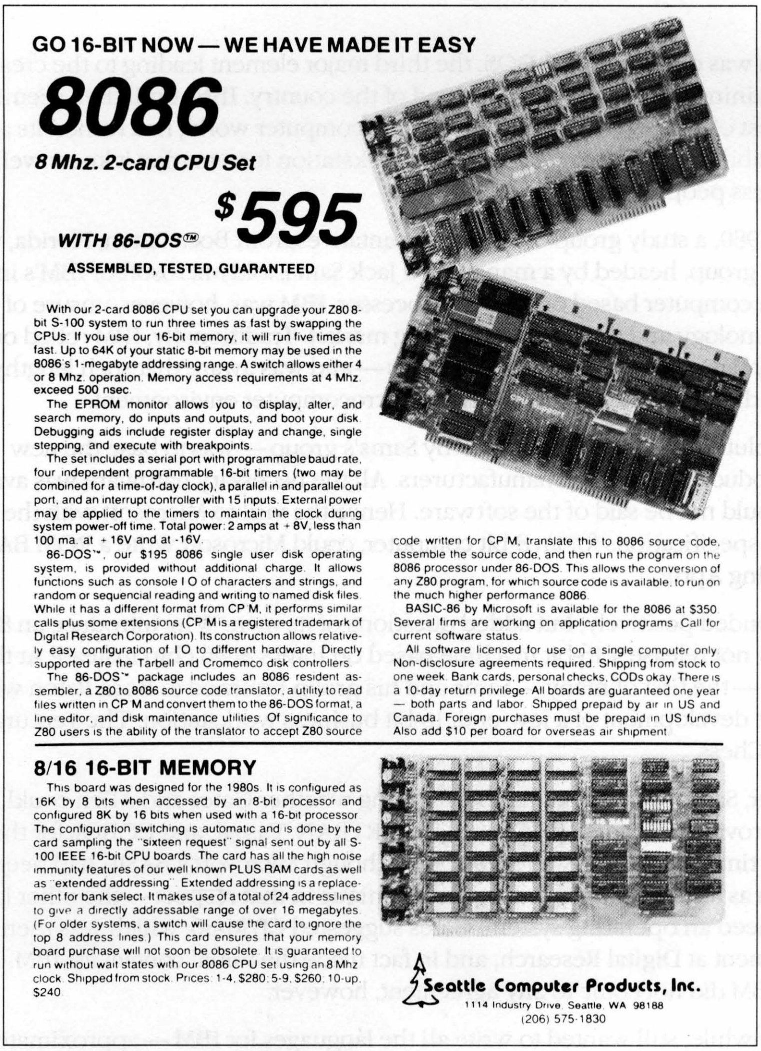
During this period, Paterson also received a call from Digital Research asking whether they could borrow the new board for developing CP/M-86. Though Seattle Computer did not have a board to loan, Paterson asked when CP/M-86 would be ready. Digital's representative said December 1979, which meant, according 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

WITH 86-DOS™ **$595**

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. exceed 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 sequencial 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 Tarbell and Cromemco disk controllers.

The 86-DOS™ package includes an 8086 resident assembler, a Z80 to 8086 source code translator, a utility to read 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 "sixteen 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 98188
(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.

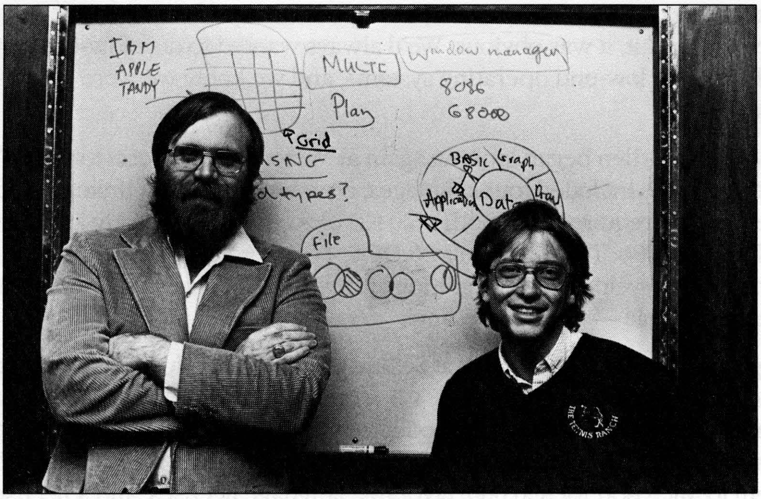
In October 1980, with 86-DOS in hand, Microsoft submitted another proposal to IBM. This time the plan included both an operating system and the languages for the new computer. Time was short and the boundaries between the languages and the operating system were unclear, so Microsoft explained that it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



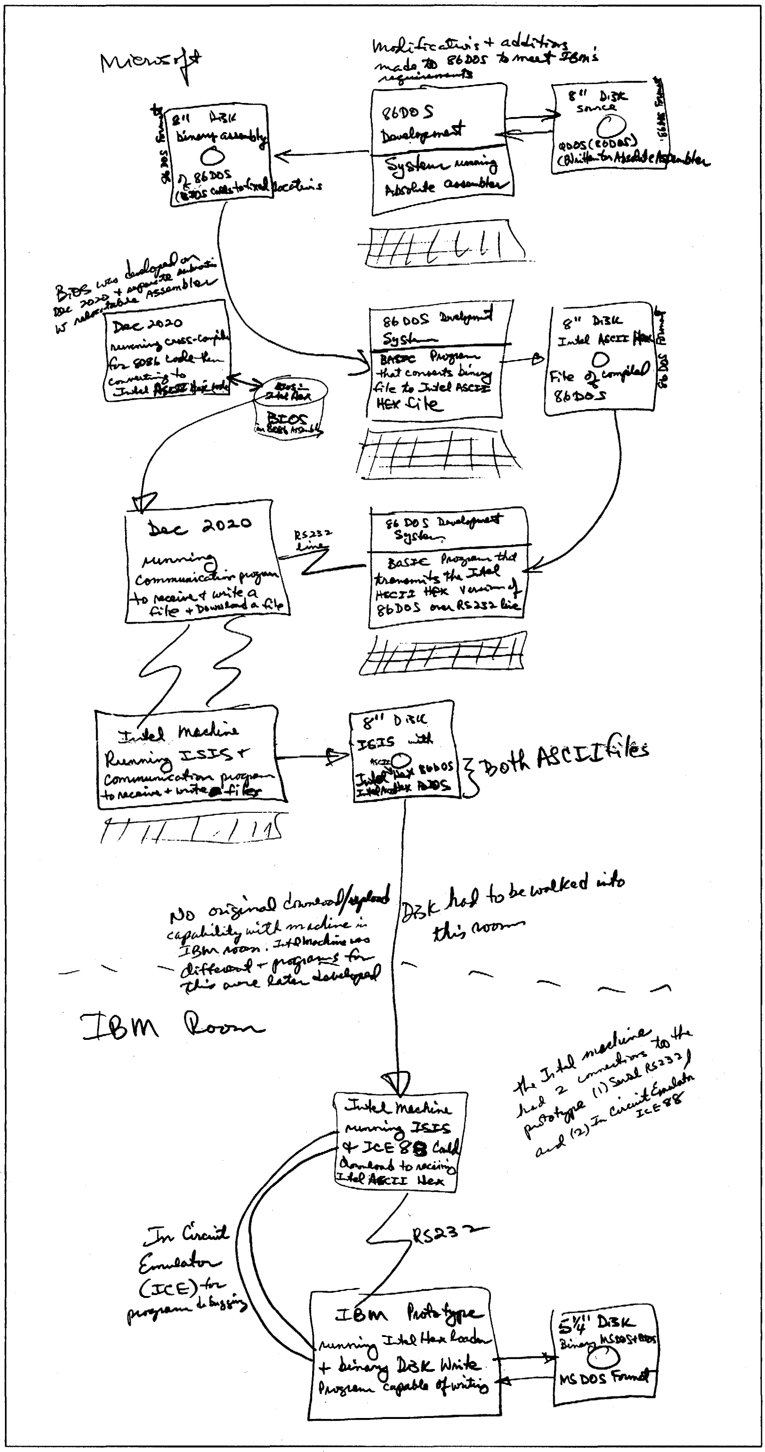
Paul Allen and Bill Gates (1982).

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.

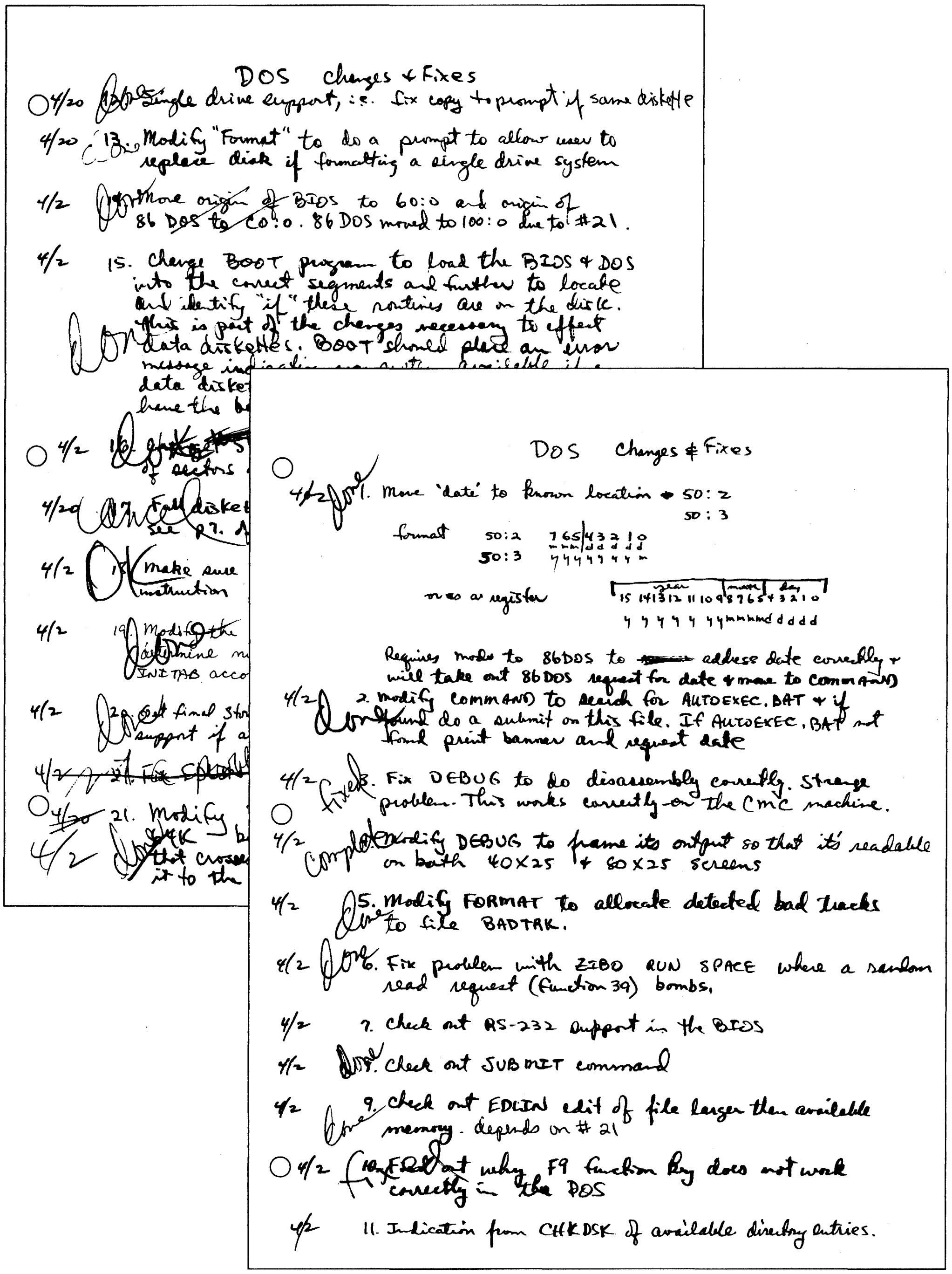
Defining and implementing the MS-DOS disk format — different from Paterson's 8-inch format — was an added challenge. Paterson's ultimate goal for 86-DOS was logical device independence, but during this first stage of development, the operating system simply had to be converted to handle logical records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


Microsoft
Modifications + additions made to 86DOS to meet IBM's requirements
8" DISK binary assembly of 86DOS (BIOS calls to fixed locations)
86DOS Format
86DOS Development System running Absolute assembler
8" DISK source 86DOS (86DOS) (Written for Absolute Assembler
86DOS Format
BIOS was developed on Dec 2020 + separate autrotic w relocatable Assembler
Dec 2020 running cross-compiler for 8086 code then converting to Intel ASCII Hex code
BIOS in Intel Hex
BIOS in 8086 Assembly
86DOS Development System
BASIC Program that converts binary file to Intel ASCII HEX file
8" DISK Intel ASCII Hex File of compiled 86DOS
86 DOS Format
Dec 2020 running communication program to receive + write a file + Download a file
RS232 line
86DOS Development System
BASIC Program that transmits the Intel ASCII HEX version of 86DOS over RS232 line
Intel Machine Running ISIS + Communication program to receive + write files
8" DISK ISIS with ASCII Intel Hex 86DOS Intel ASCII Hex BIOS
Both ASCII files
No original download/upload capability with machine in IBM room. Intel Machine was different + programs for this were later developed
DISK had to be walked into this room
IBM Room
The Intel machine had 2 connections to the prototype (1) Serial RS232 and (2) In Circuit Emulator ICE88
Intel Machine running ISIS + ICE 88 Could download to receiving Intel ASCII Hex
RS232
In Circuit Emulator (ICE) for program debugging
IBM Prototype running Intel Hex loader + binary DISK Write Program capable of writing
5¼" DISK Binary MSDOS+BIOS MS DOS Format

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



DOS changes & Fixes

○ 4/20 Bob 12. Single drive support, i.e. fix copy to prompt if same diskette

4/20 13. Modify "Format" to do a prompt to allow user to replace disk if formatting a single drive system

4/2 Done 14. ~~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. This is part of the changes necessary to effect data diskettes. BOOT should place an error
Done message in
data disket
have the b

○ 4/2 Done 16.
of sectors

4/20 Done 17. Fall disket
see #7.

4/2 OK 18. make sure
instruction

4/2 19. Modify the
Done determine n
INITAB acco

4/2 Done 20. Get final Stor
support if a

~~4/2 21. Fix Sp~~

○ ~~4/20~~ 4/2 21. Modify
Done 64K — b
that crosse
it to the

DOS Changes & Fixes

○

4/2 Done 1. Move 'date' to known location • 50:2
50:3

```
format     50:2    7 6 5|4 3 2 1 0
                   m m m|d d d d d
           50:3    y y y y y y y m

                     year      month  day
or as a register   15 14 13 12 11 10 9|8 7 6 5|4 3 2 1 0
                    y  y  y  y  y  y y m m m m d d d d
```

Requires mods to 86DOS to ~~[illegible]~~ address date correctly & will take out 86DOS request for date & move to COMMAND

4/2 Done 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 fixed 3. Fix DEBUG to do disassembly correctly. Strange problem. This works correctly on the CMC machine.

○

4/2 Complete 4. Modify DEBUG to frame its output so that it's readable on both 40X25 & 80X25 screens

4/2 Done 5. Modify FORMAT to allocate detected bad tracks to file BADTRK.

4/2 Done 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

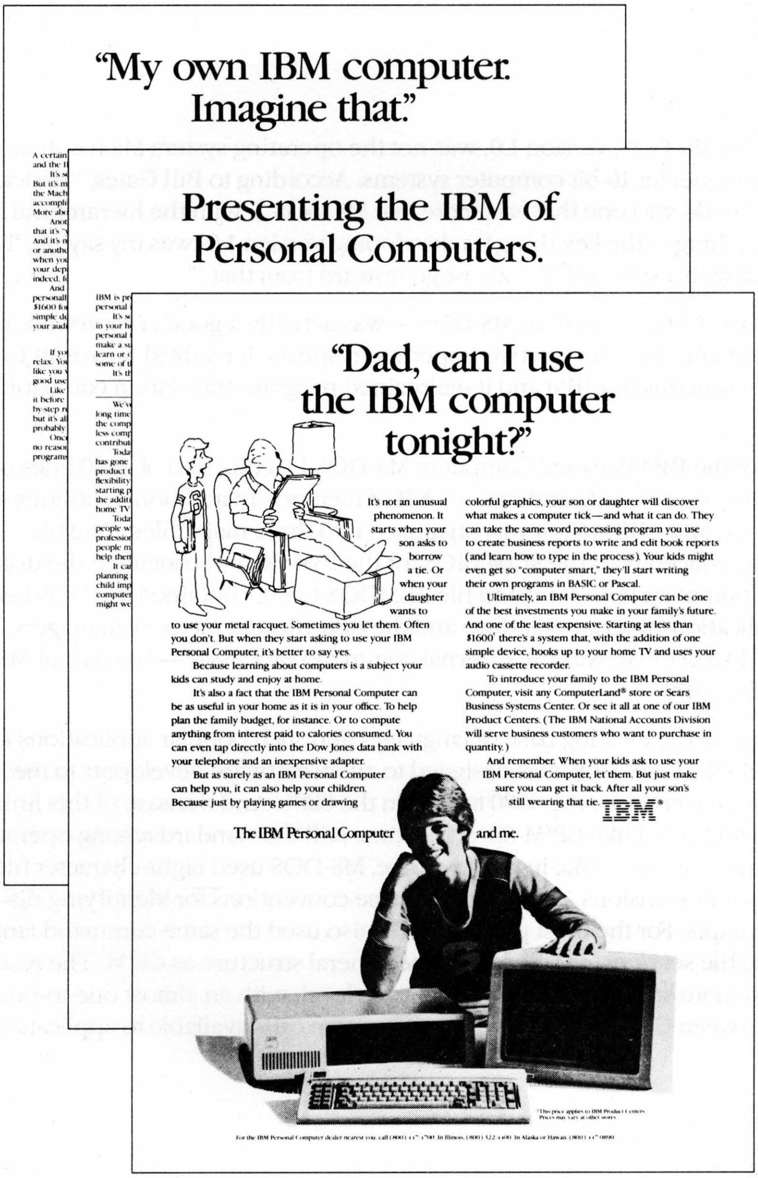
4/2 Done 8. Check out SUBMIT command

4/2 Done 9. check out EDLIN edit of file larger then available memory. depends on #21

○ 4/2 fix 10. ~~Find out~~ why F9 function key does not work correctly in the DOS

4/2 11. Indication from CHKDSK of available directory 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 tie. Or when your daughter wants to to use your metal racquet. Sometimes you let them. Often you don't. But when they start asking to use your IBM Personal Computer, it's better to say yes.

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

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

But as surely as an IBM Personal Computer can help you, it can also help your children. Because just by playing games or drawing colorful graphics, your son or daughter will discover what makes a computer tick—and what it can do. They can take the same word processing program you use to create business reports to write and edit book reports (and learn how to type in the process). Your kids might even get so "computer smart," they'll start writing their own programs in BASIC or Pascal.

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

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

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

The IBM Personal Computer and me.

[†] This price applies to IBM Product Centers. Prices may vary at other stores.

For the IBM Personal Computer dealer nearest you, call (800) 447-4700. In Illinois, (800) 322-4400. In Alaska or Hawaii, (800) 447-0890.

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.'"

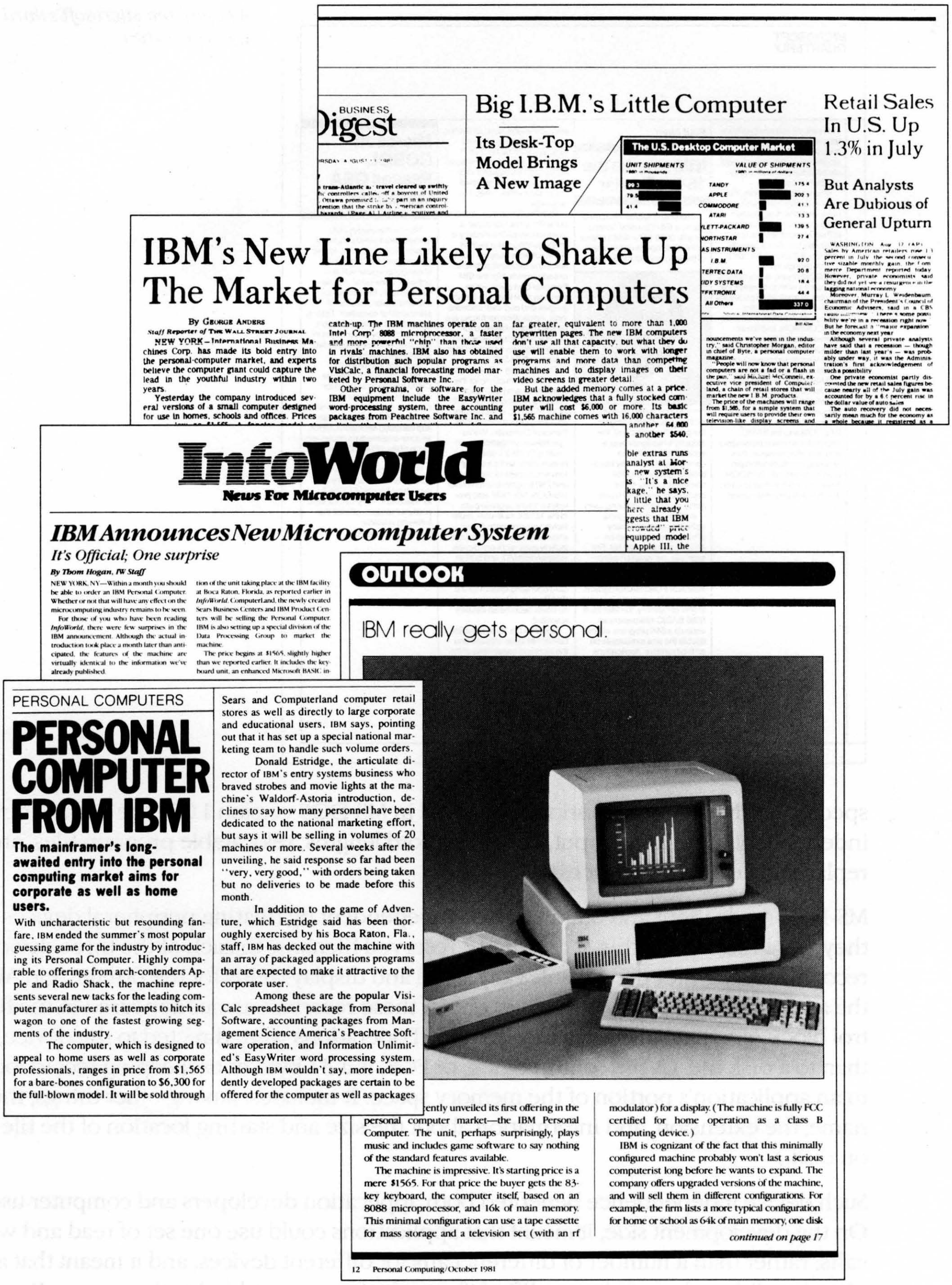
This first version — Gates's subset of MS-DOS — was actually a good compromise between the present and the future in two important respects: It enabled Microsoft to meet the development schedule for IBM and it maintained program-translation compatibility with CP/M.

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

RSDAY, AUGUST 13, 1981

trans-Atlantic air travel cleared up swiftly
he controllers called off a boycott of United
Ottawa promised to take part in an inquiry
tention that the strike by American control-
hazards. (Page A1.) Airline executives and

Big I.B.M.'s Little Computer

Its Desk-Top Model Brings A New Image

The U.S. Desktop Computer Market

UNIT SHIPMENTS 1980 in thousands		VALUE OF SHIPMENTS 1980 in millions of dollars
99.3	TANDY	175.4
79.5	APPLE	202.3
41.4	COMMODORE	41.1
	ATARI	13.3
	...LETT-PACKARD	139.5
	NORTHSTAR	27.4
	...AS INSTRUMENTS	
	I.B.M.	92.0
	...TERTEC DATA	20.8
	...IDY SYSTEMS	18.4
	...EKTRONIX	44.4
	All Others	337.0

Source: International Data Corporation

Bill Allen

nouncements we've seen in the industry," said Christopher Morgan, editor in chief of Byte, a personal computer magazine.

"People will now know that personal computers are not a fad or a flash in the pan," said Michael McConnell, executive vice president of Computerland, a chain of retail stores that will market the new I.B.M. products.

The price of the machines will range from $1,565, for a simple system that will require users to provide their own television-like display screens and

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

But Analysts Are Dubious of General Upturn

WASHINGTON, Aug. 12 (AP) Sales by American retailers rose 1.3 percent in July, the second consecutive sizable monthly gain, the Commerce Department reported today. However, private economists said they did not yet see a resurgence in the lagging national economy.

Moreover, Murray L. Weidenbaum, chairman of the President's Council of Economic Advisers, said in a CBS radio interview: "There's some possibility we're in a recession right now." But he forecast a "major expansion" in the economy next year.

Although several private analysts have said that a recession — though milder than last year's — was probably under way, it was the Administration's first acknowledgement of such a possibility.

One private economist partly discounted the new retail sales figures because nearly all of the July gain was accounted for by a 6.6 percent rise in the dollar value of auto sales.

The auto recovery did not necessarily mean much for the economy as a whole because it registered as a

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

By George Anders
Staff Reporter of The Wall Street Journal

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

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

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

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

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

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

another 64,000
s another $540.

ble extras runs
analyst at Mor-
new system's
s. "It's a nice
kage," he says.
y little that you
here already."
ggests that IBM
"crowded" price
equipped model
Apple III, the

InfoWorld
News For Microcomputer Users

IBM Announces New Microcomputer System

It's Official; One surprise

By Thom Hogan, IW Staff

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

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

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

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

PERSONAL COMPUTERS

PERSONAL COMPUTER FROM IBM

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

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

The computer, which is designed to appeal to home users as well as corporate professionals, ranges in price from $1,565 for a bare-bones configuration to $6,300 for the full-blown model. It will be sold through Sears and Computerland computer retail stores as well as directly to large corporate and educational users, IBM says, pointing out that it has set up a special national marketing team to handle such volume orders.

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

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

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

OUTLOOK

IBM really gets personal.

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

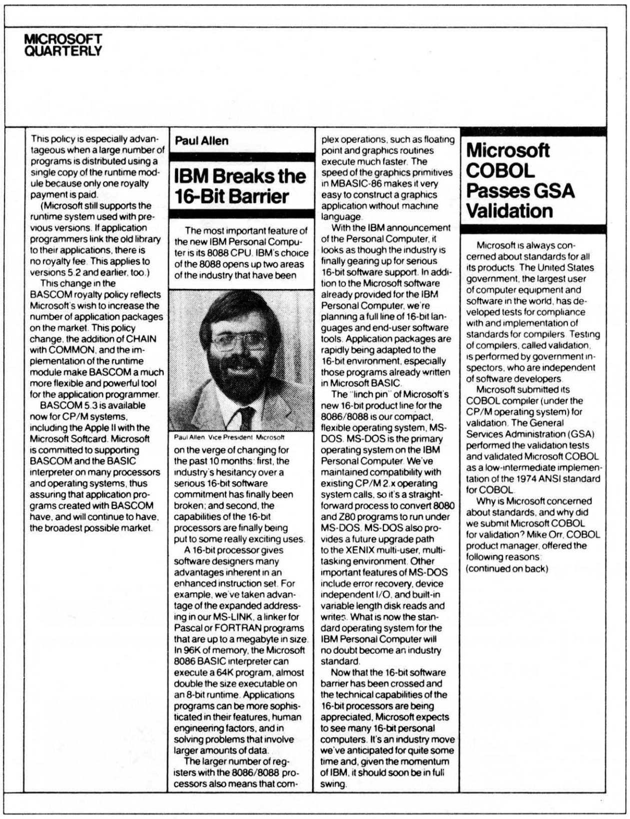
The machine is impressive. It's starting price is a mere $1565. For that price the buyer gets the 83-key keyboard, the computer itself, based on an 8088 microprocessor, and 16k of main memory. This minimal configuration can use a tape cassette for mass storage and a television set (with an rf modulator) for a display. (The machine is fully FCC certified for home operation as a class B computing device.)

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

continued on page 17

12 Personal Computing/October 1981

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



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

MICROSOFT QUARTERLY

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

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

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

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

Paul Allen

IBM Breaks the 16-Bit Barrier

The most important feature of the new IBM Personal Computer is its 8088 CPU. IBM's choice of the 8088 opens up two areas of the industry that have been on the verge of changing for the past 10 months: first, the industry's hesitancy over a serious 16-bit software commitment has finally been broken; and second, the capabilities of the 16-bit processors are finally being put to some really exciting uses.

Paul Allen Vice President Microsoft

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

The larger number of registers with the 8086/8088 processors also means that complex operations, such as floating point and graphics routines execute much faster. The speed of the graphics primitives in MBASIC-86 makes it very easy to construct a graphics application without machine language.

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

The "linch pin" of Microsoft's new 16-bit product line for the 8086/8088 is our compact, flexible operating system, MS-DOS. MS-DOS is the primary operating system on the IBM Personal Computer. We've maintained compatibility with existing CP/M 2.x operating system calls, so it's a straightforward process to convert 8080 and Z80 programs to run under MS-DOS. MS-DOS also provides a future upgrade path to the XENIX multi-user, 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 all its products. The United States government, the largest user of computer equipment and software in the world, has developed tests for compliance with and implementation of standards for compilers. Testing of compilers, called validation, is performed by government inspectors, who are independent of software developers.

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

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

(continued on back)

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

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

Such device independence benefited both application developers and computer users. On the development side, it meant that applications could use one set of read and write calls, rather than a number of different calls for different devices, and it meant that an application did not have to be modified if new devices were added to the system. From the user's point of view, device independence meant greater flexibility. For example, even if a program had been designed for disk I/O only, the user could still use a file for input or direct output to the printer.

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

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

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

Performance

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

Like CP/M, MS-DOS shared out disk space in allocation units. Unlike CP/M, however, MS-DOS mapped the use of these allocation units in a central file allocation table—the FAT—that was always in memory. Both operating systems used a directory entry for recording information about each file, but whereas a CP/M directory entry included an allocation map—a list of sixteen 1 KB allocation units where successive parts of the file were stored—an MS-DOS directory entry pointed only to the first allocation unit in the FAT and each entry in the table then pointed to the next unit associated with the file. Thus, CP/M might require several directory entries (and more than one disk access) to load a file larger than 16 KB, but MS-DOS retained a complete in-memory list of all file components and all available disk space without having to access the disk at all. As a result, MS-DOS's ability to find and load even very long files was extremely rapid compared with CP/M's.

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

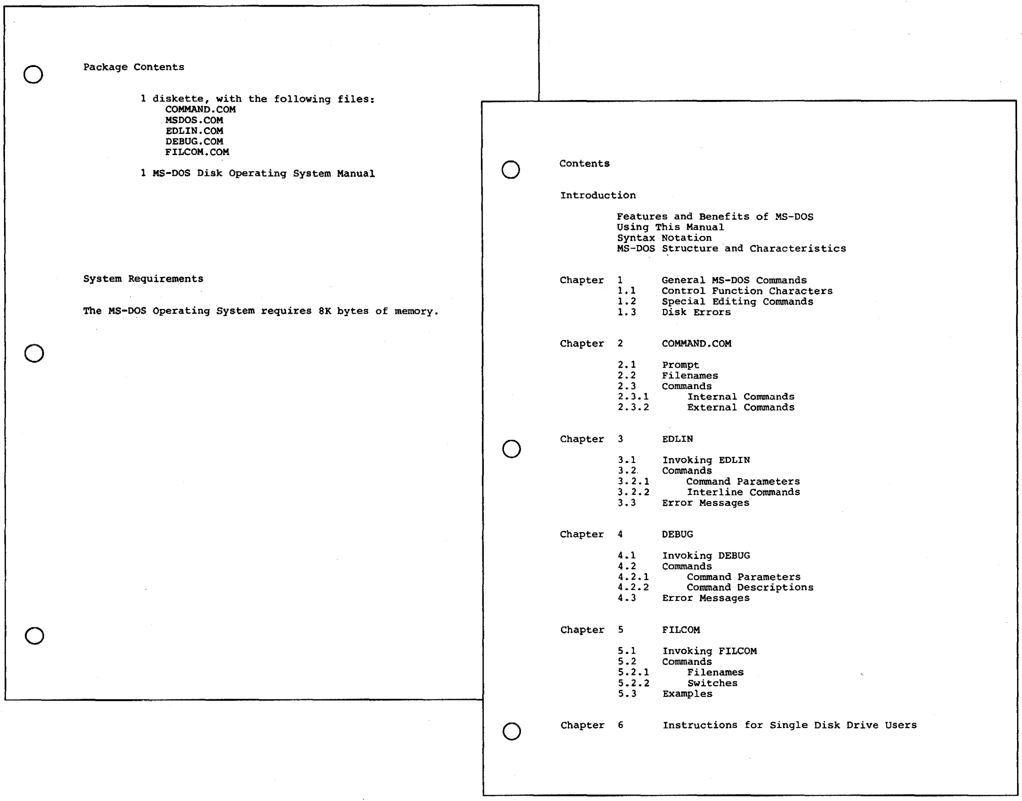
The independence of the logical record from the physical sector laid the foundation for the ability to read and write multiple sectors. When reading multiple records in CP/M, an application had to issue a read function call for each sector, one at a time. With MS-DOS, the application could issue one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

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

Ease of Use

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

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



```
Package Contents

        1 diskette, with the following files:
            COMMAND.COM
            MSDOS.COM
            EDLIN.COM
            DEBUG.COM
            FILCOM.COM

        1 MS-DOS Disk Operating System Manual



System Requirements

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

```
Contents

Introduction

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

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

Chapter  2       COMMAND.COM

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

Chapter  3       EDLIN

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

Chapter  4       DEBUG

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

Chapter  5       FILCOM

         5.1     Invoking FILCOM
         5.2     Commands
         5.2.1      Filenames
         5.2.2      Switches
         5.3     Examples

Chapter  6       Instructions for Single Disk Drive Users
```

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

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

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

Another new feature — one visible with the DIR command — was date and time stamping of disk files. Even in its earliest forms, MS-DOS tracked the system date and displayed it at every startup, and now, when it turned out that only the first 16 bytes of a directory entry were needed for file-header information, the MS-DOS programmers decided to use some of the remaining 16 bytes to record the date and time of creation or update (and the size of the file) as well.

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

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

The Marketplace

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

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

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

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

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

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

An operating system by any other name...

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

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

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

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

Developers and MS-DOS

Early in its career, MS-DOS represented just a small fraction of Microsoft's business — much larger revenues were generated by BASIC and other languages. In addition, in the first two years after the introduction of the IBM PC, the growth of CP/M-86 and other environments nearly paralleled that of MS-DOS. So Microsoft found itself in the unenviable position of giving its support to MS-DOS while also selling languages to run on CP/M-86, thereby contributing to the growth of software for MS-DOS's biggest competitor.

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

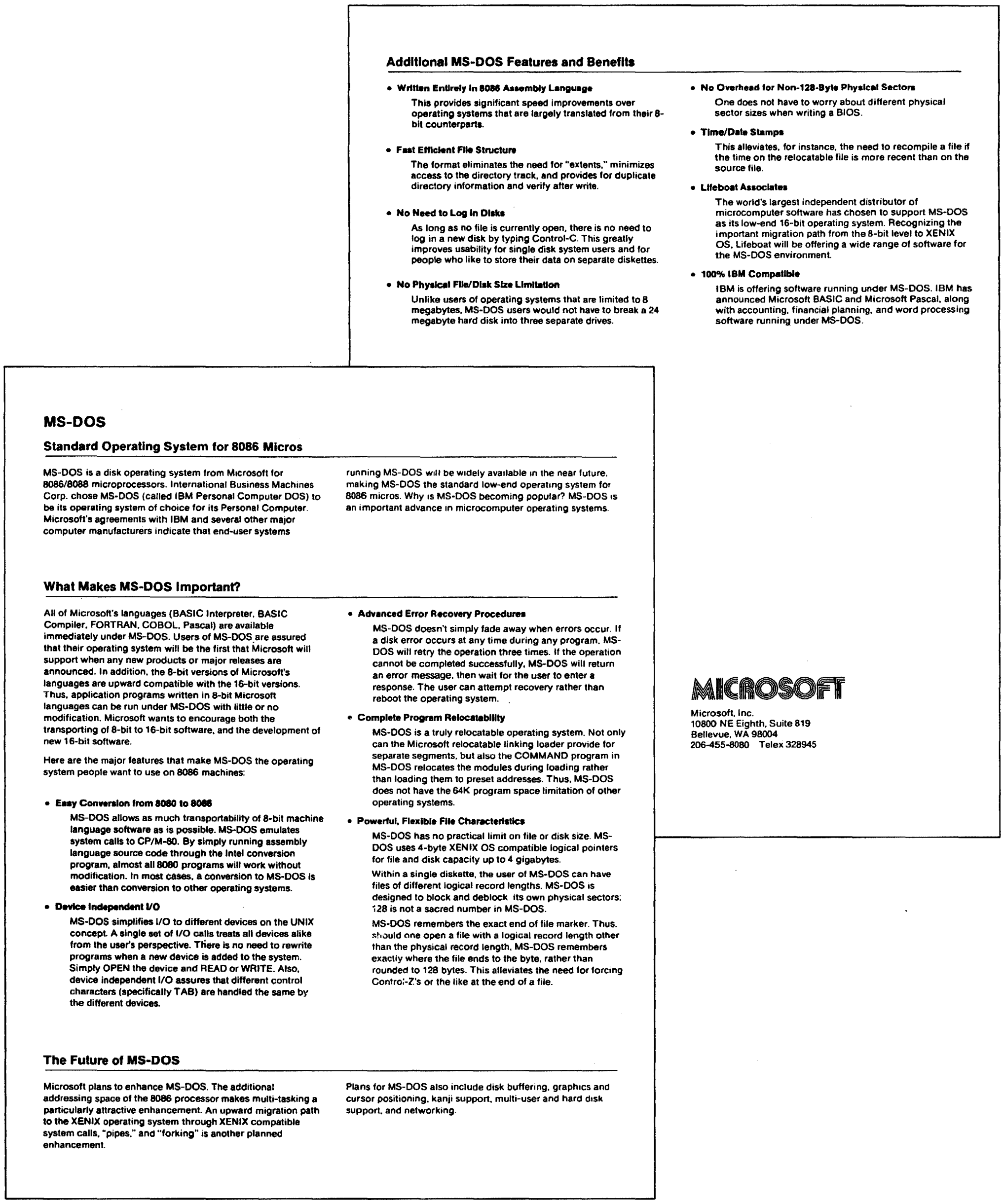
Microsoft approached the problem by emphasizing four related points in its discussions with hardware manufacturers:

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

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

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

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



Additional MS-DOS Features and Benefits

- **Written Entirely In 8086 Assembly Language**

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

- **Fast Efficient File Structure**

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

- **No Need to Log In Disks**

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

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

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

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

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

- **Time/Date Stamps**

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

- **Lifeboat Associates**

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

- **100% IBM Compatible**

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

MS-DOS

Standard Operating System for 8086 Micros

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

What Makes MS-DOS Important?

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

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

- **Easy Conversion from 8080 to 8086**

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

- **Device Independent I/O**

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

- **Advanced Error Recovery Procedures**

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

- **Complete Program Relocatability**

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

- **Powerful, Flexible File Characteristics**

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

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

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

The Future of MS-DOS

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

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

MICROSOFT

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

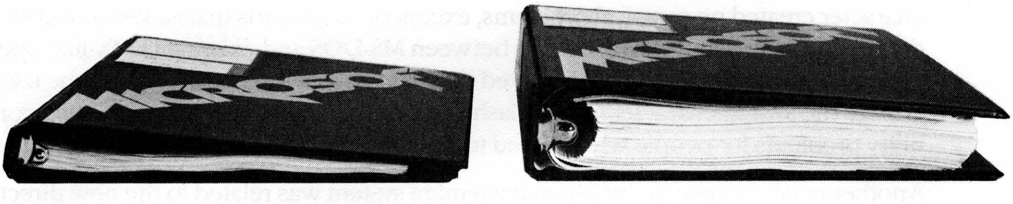
There were three particular areas that interested Microsoft: a new, hierarchical file system, installable device drivers, and some type of multitasking. Each of these features contributed to version 2.0, and together they represented a major change in MS-DOS while still maintaining compatibility with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

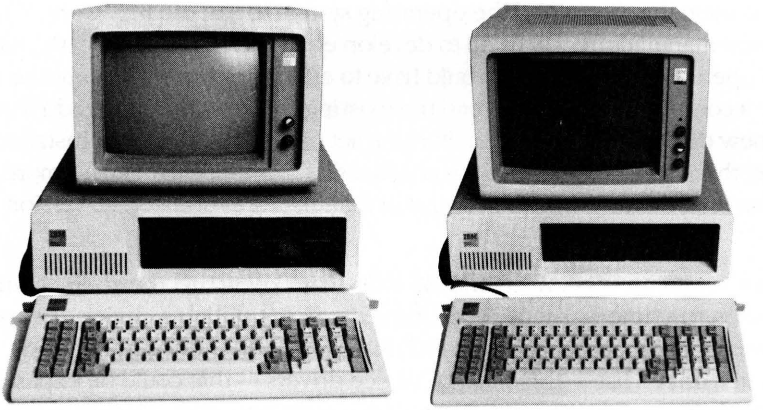
At IBM's request, version 2.0 of MS-DOS also possessed the undocumented ability to perform rudimentary background processing — an interim solution to a growing awareness of the potentials of multitasking.

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

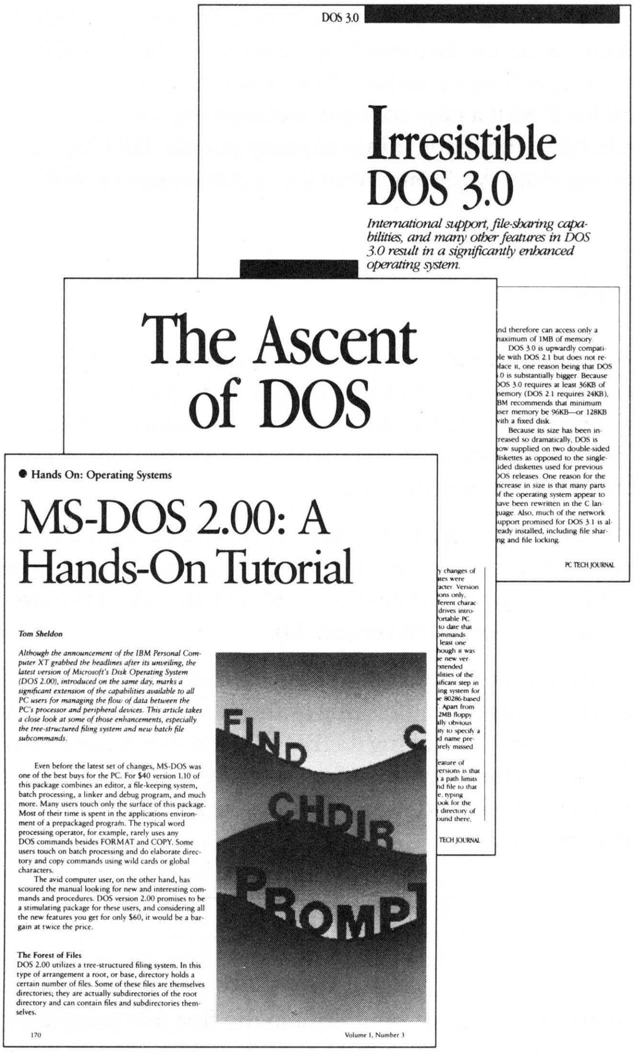
When version 2.0 was released, IBM was already planning to introduce its PCjr. The PCjr would have the ability to run programs from ROM cartridges and, in addition to using half-height 5¼-inch drives, would employ a slightly different disk-controller architecture. Because of these differences from the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


FIND
C
CHDIR
PROMPT

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

DOS 3.0

Irresistible DOS 3.0

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

nd therefore can access only a
naximum of 1MB of memory.
DOS 3.0 is upwardly compati-
ble with DOS 2.1 but does not re-
place it, one reason being that DOS
.0 is substantially bigger. Because
DOS 3.0 requires at least 36KB of
nemory (DOS 2.1 requires 24KB),
BM recommends that minimum
user memory be 96KB—or 128KB
vith a fixed disk.
Because its size has been in-
reased so dramatically, DOS is
now supplied on two double-sided
diskettes as opposed to the single-
ided diskettes used for previous
DOS releases. One reason for the
ncrease in size is that many parts
of the operating system appear to
have been rewritten in the C lan-
guage. Also, much of the network
upport promised for DOS 3.1 is al-
ready installed, including file shar-
ng and file locking.

PC TECH JOURNAL

The Ascent of DOS

y changes of
ates were
acter. Version
ons only,
ferent charac-
drives intro-
Portable PC
to date that
ommands
least one
hough it was
e new ver-
xtended
ilities of the
ificant step in
ing system for
e 80286-based
. Apart from
2MB floppy
lly obvious
ity to specify a
d name pre-
rely missed

eature of
versions is that
a path limits
nd file to that
e, typing
ook for the
directory of
ound there.

TECH JOURNAL

● Hands On: Operating Systems

MS-DOS 2.00: A Hands-On Tutorial

Tom Sheldon

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

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

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

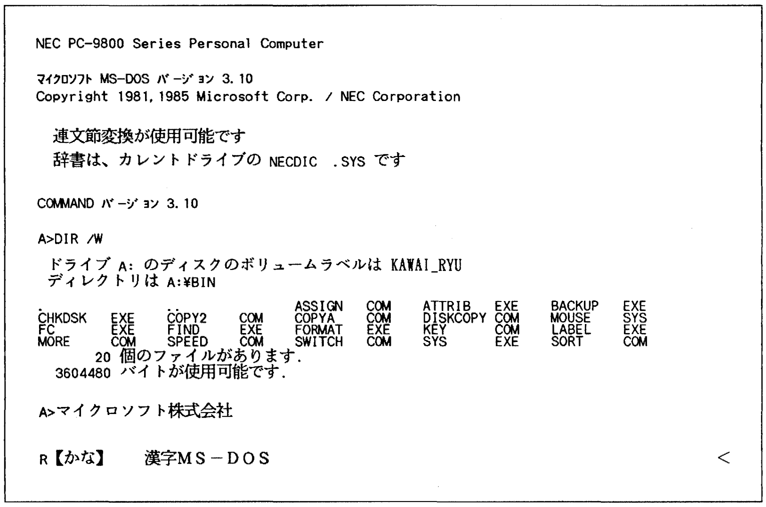
The Forest of Files

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

170 Volume 1, Number 3

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 ﾊﾞｰｼﾞｮﾝ 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 ﾊﾞｰｼﾞｮﾝ 3.10

A>DIR /W

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

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

A>マイクロソフト株式会社

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

A Kanji screen with the MS-DOS copyright message.

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

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

Different customers, different versions

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

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

Software Concerns

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

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

For example, because the information about the internals of the BIOS and the ROM 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."

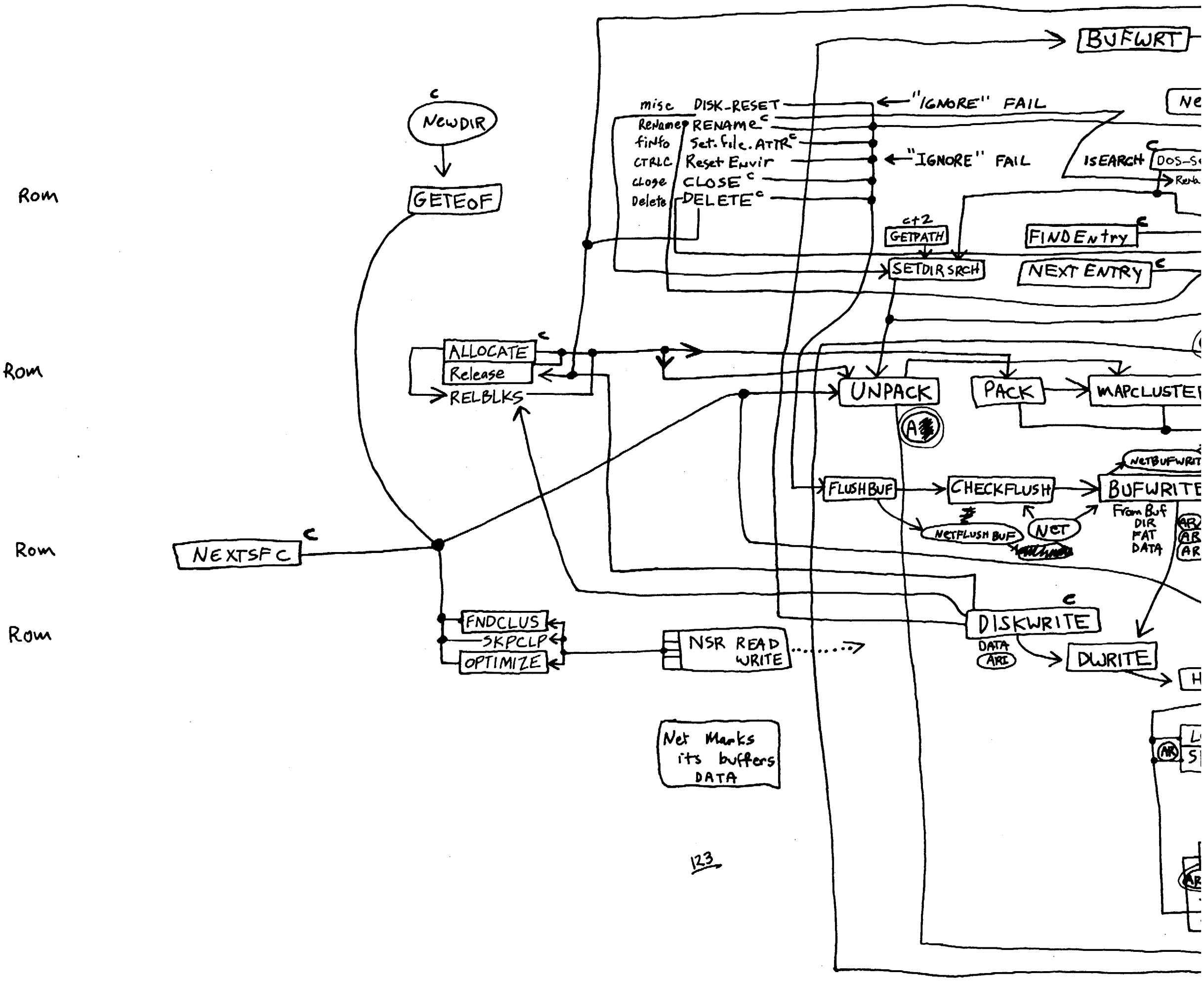
In fact, the great variability in programs and programming approaches that MS-DOS supported eventually proved to be one of the biggest obstacles to the development of a sophisticated networking system and, in the longer term, to the addition of true multitasking.

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.

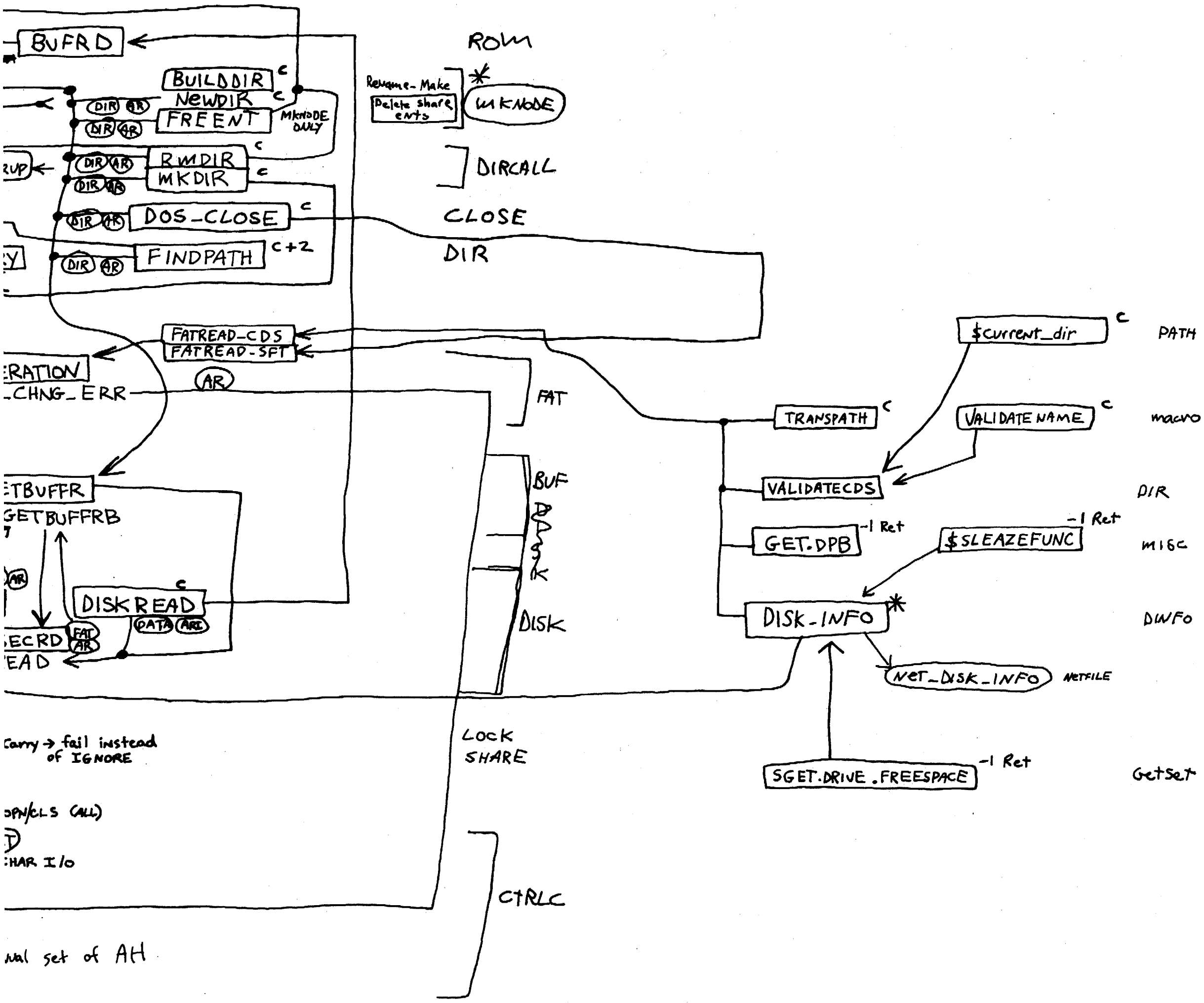
Version 3.0

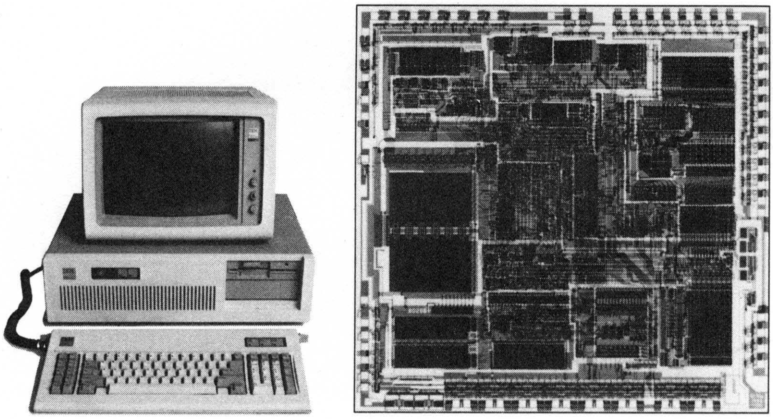
All told, the work on version 3.0 of MS-DOS proved to be long and difficult. For a year and a half, Microsoft grappled with problems of software incompatibility, remote file management, and logical device independence at the network level. Even so, when IBM was ready to announce its new Personal Computer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


BUFRD
ROM
BUILDDIR c
NewDIR c
FREENT
MKNODE ONLY
Rename-Make
Delete share ents
*
MKNODE
RMDIR c
MKDIR c
DIRCALL
DOS_CLOSE c
CLOSE
FINDPATH c+2
DIR
DIR AR
FATREAD-CDS
FATREAD-SFT
AR
RATION
CHNG_ERR
FAT
TBUFFR
GETBUFFRB
BUF
DISKREAD c
DATA AR
FAT AR
ECRD
EAD
DISK
$current_dir c
PATH
TRANSPATH c
VALIDATE NAME c
macro
VALIDATECDS
DIR
GET.DPB -1 Ret
$SLEAZEFUNC -1 Ret
MISC
DISK_INFO *
DINFO
net_DISK_INFO
NETFILE
SGET.DRIVE.FREESPACE -1 Ret
GetSet
Lock
SHARE
CTRLC
Carry → fail instead of IGNORE
OPN/CLS (ALL)
CHAR I/O
al set of AH



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, memory, and pipes. Of these, MS-DOS formally supports only pipes. (A pipe unidirectional, sequential stream of data that is written by one program a another.) The data in a pipe resides in memory or in a disk file, deper mentation; MS-DOS uses disk files for intermediate storage of data is a single-tasking operating system.

Memory management

Because the amount of memory a program needs var traditional operating system ordinarily provides m 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 ta. The directory contains a list of the files stored on the device, their sizes, and informa- about where the data for each file is located.

different approaches to file allocation and directory entries exist. MS-DOS uses a 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 tc 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 ir e THE DEVELOPMENT OF MS-DOS. The major MS-DOS and PC-DOS implem 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 ver-
ns switched to a hierarchical file system like that found in UNIX/XENIX and to file-
le access instead of FCBs. (A file handle is a 16-bit number used to reference an inter-
e that MS-DOS uses to keep track of currently open files; an application program
cess to this internal table.) The UNIX/XENIX-style file functions allow files to be
byte stream instead of as a collection of records. Applications can read or write
s in a single operation, starting at any byte offset within the file. Filenames

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

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

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

Basic MS-DOS Requirements

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

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

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

Specific hardware requirements

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

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

The microprocessor

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

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

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

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

Memory requirements

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

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

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

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

Peripheral devices

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

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

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

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

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

The ROM BIOS

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

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

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

Summary

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

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

William Wong

Article 2
The Components of MS-DOS

MS-DOS is a modular operating system consisting of multiple components with 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 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.

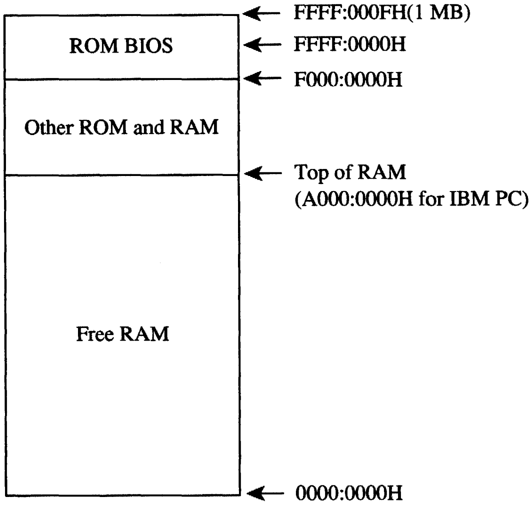


Figure 2-1. Memory layout at startup.

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

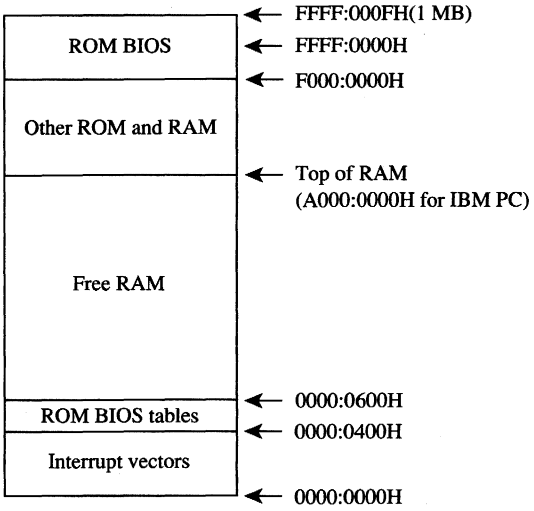


Figure 2-2. The interrupt vector table and the ROM BIOS table.

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

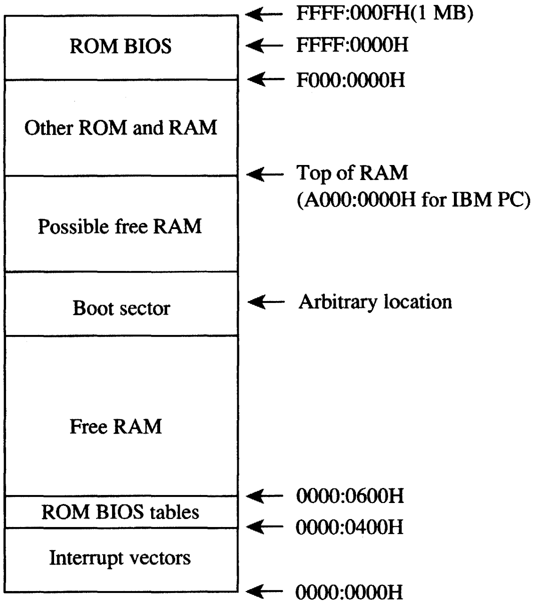


Figure 2-3. A loaded boot sector.

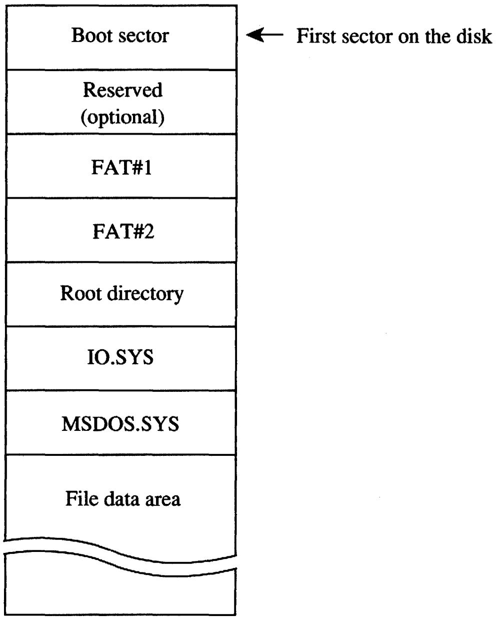


Figure 2-4. Boot-disk configuration.

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.

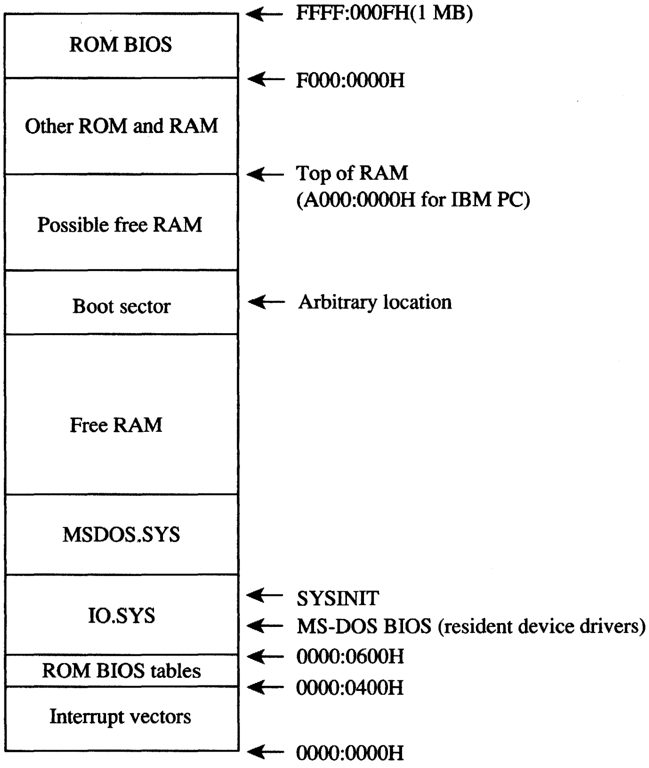


Figure 2-5. IO.SYS and MSDOS.SYS loaded.

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.

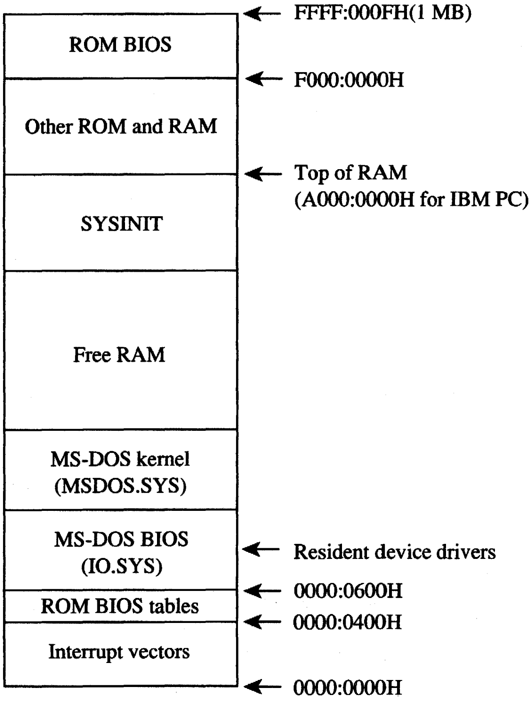


Figure 2-6. SYSINIT and MSDOS.SYS relocated.

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.

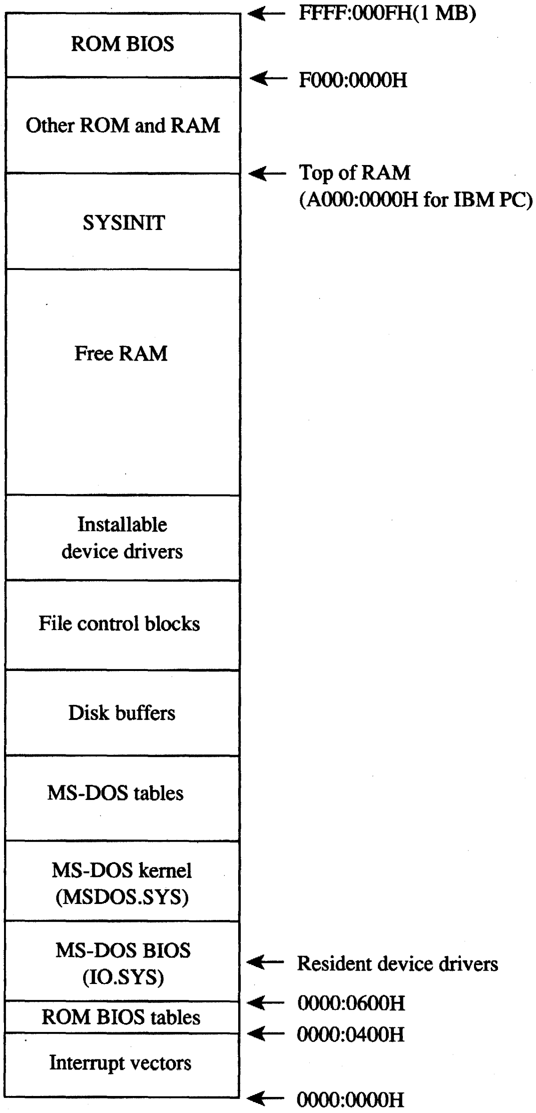


Figure 2-7. Tables allocated and installable device drivers loaded.

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 reloading 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.

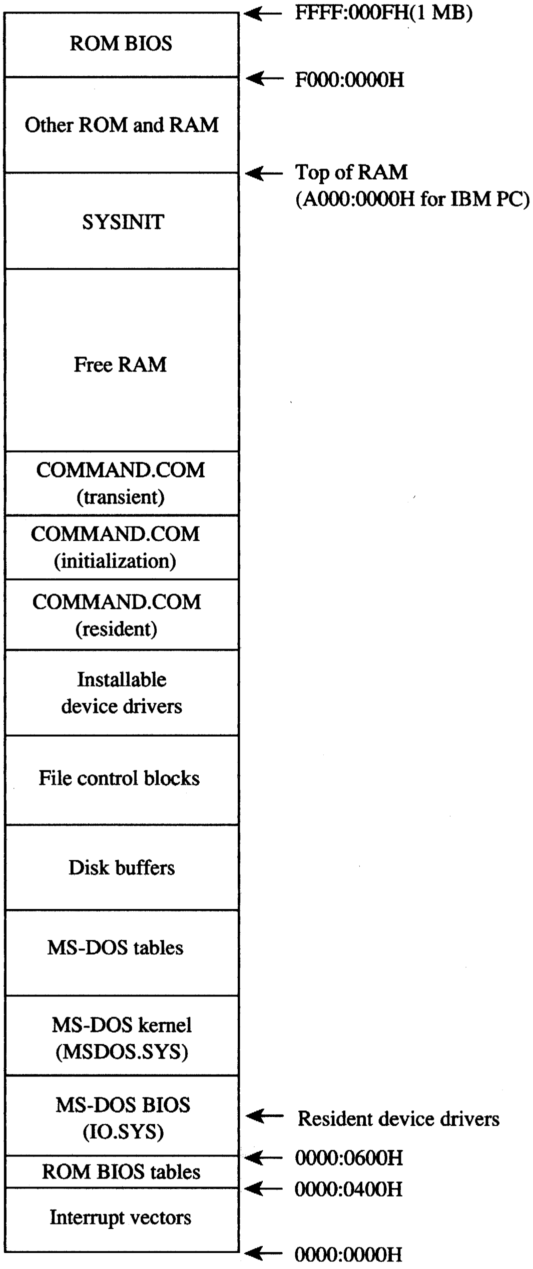


Figure 2-8. COMMAND.COM loaded.

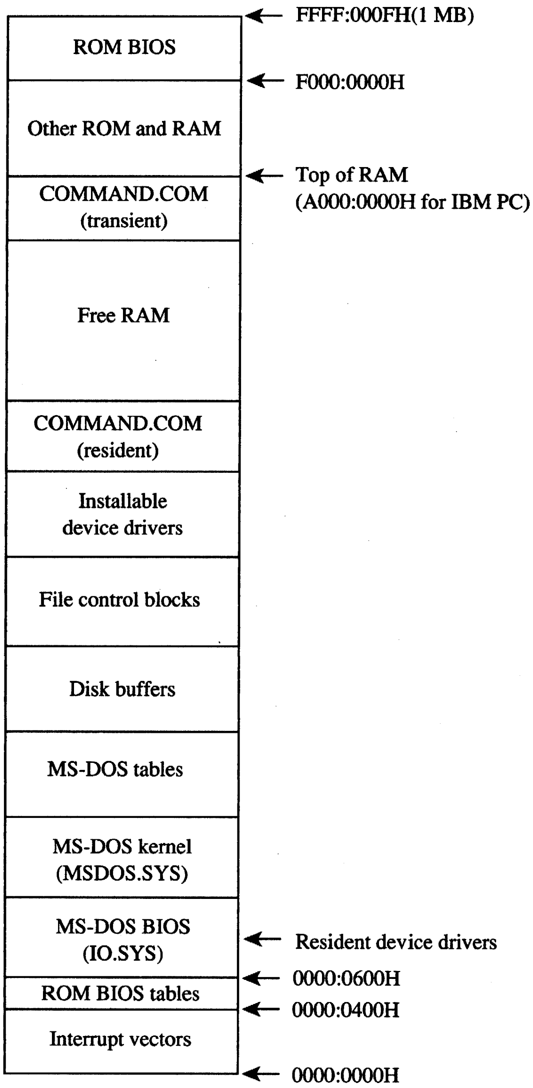


Figure 2-9. COMMAND.COM after relocation.

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.

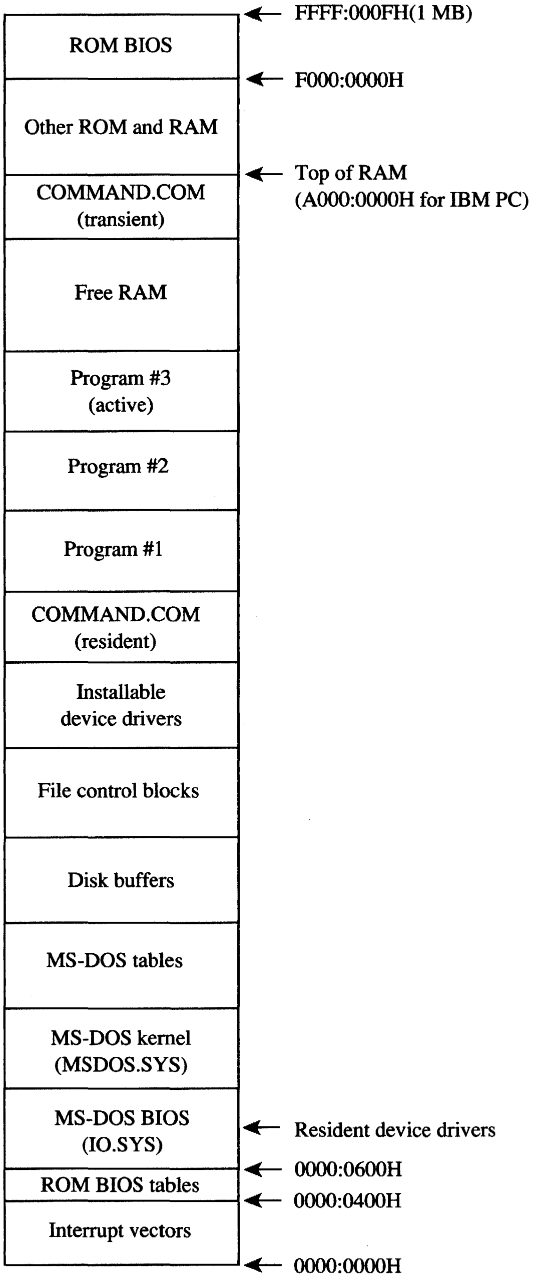


Figure 2-10. Multiple programs loaded.

```
; 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).

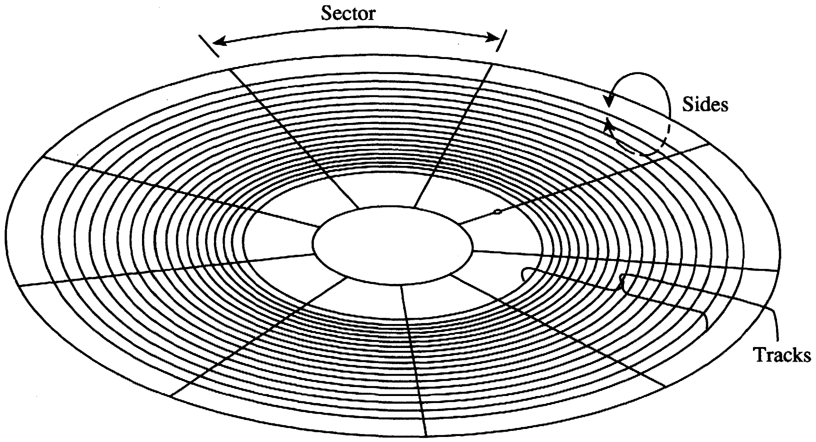


Figure 3-1. The physical layout of a CAV 9-sector, 5.25-inch floppy disk.

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).

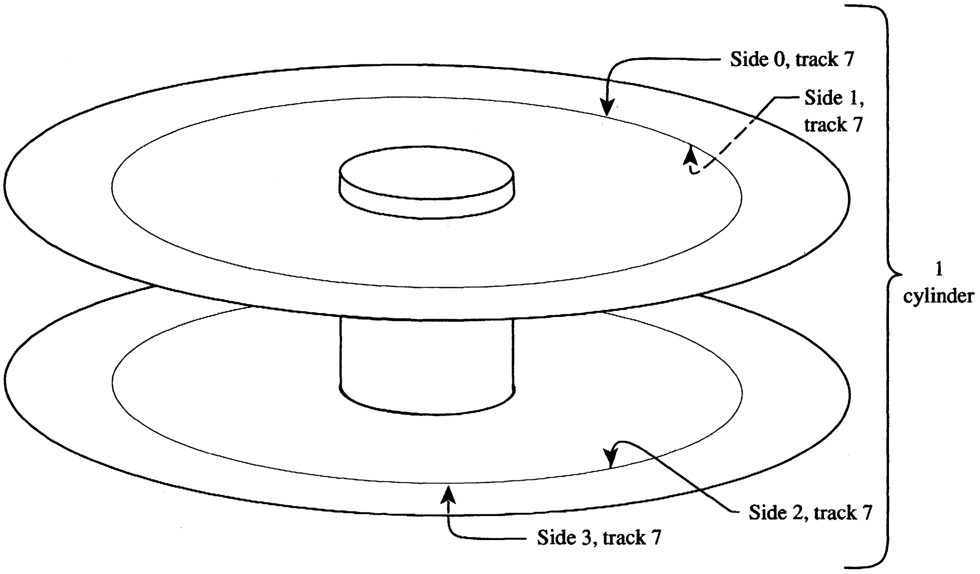


Figure 3-2. Tracks and cylinders on a fixed-disk system.

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.

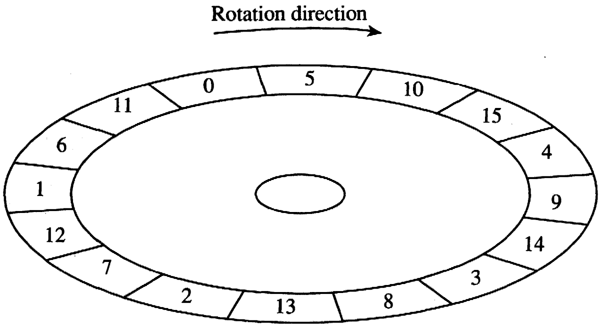


Figure 3-3. A 3:1 interleave.

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.)

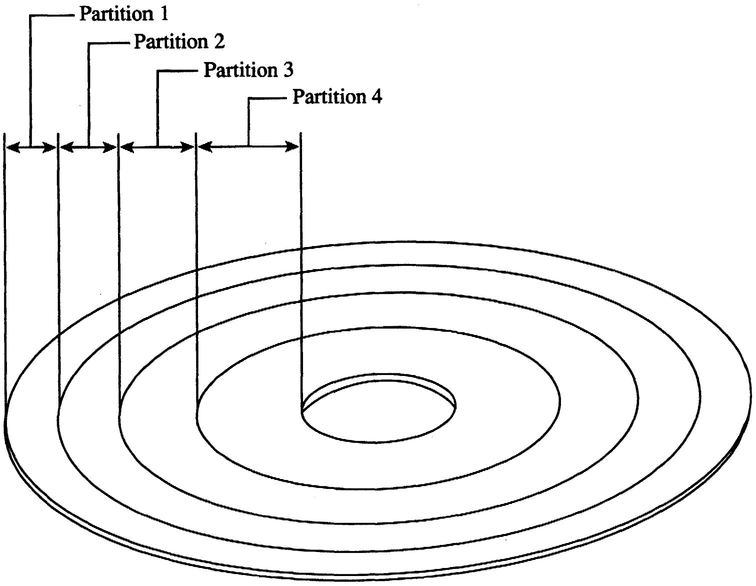


Figure 3-4. A partitioned disk.

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.

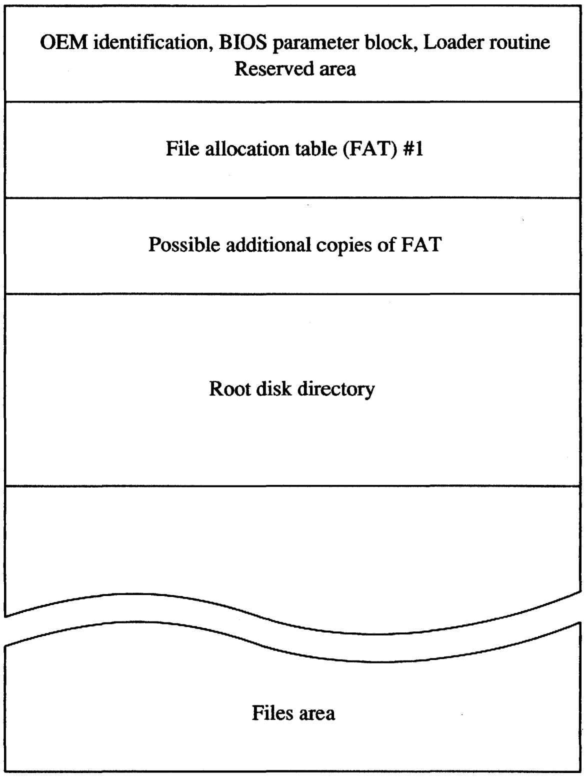

OEM identification, BIOS parameter block, Loader routine
Reserved area
File allocation table (FAT) #1
Possible additional copies of FAT
Root disk directory
Files area

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

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

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

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

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

*Assumes 512 bytes per sector.

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

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

The boot sector

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

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

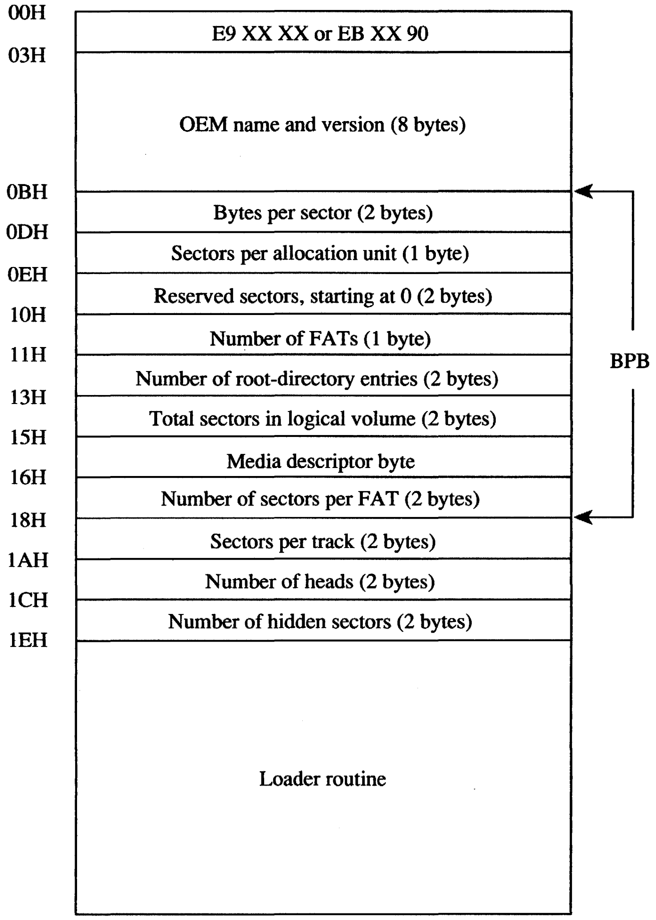

00H
E9 XX XX or EB XX 90
03H
OEM name and version (8 bytes)
0BH
Bytes per sector (2 bytes)
0DH
Sectors per allocation unit (1 byte)
0EH
Reserved sectors, starting at 0 (2 bytes)
10H
Number of FATs (1 byte)
11H
Number of root-directory entries (2 bytes)
13H
Total sectors in logical volume (2 bytes)
15H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

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

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

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

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   EB 2D 90 20 20 20 20 20-20 20 20 00 02 02 01 00   k-.        .....
0010   02 70 00 A0 05 F9 03 00-09 00 02 00 00 00 00 00   .p. .y..........
0020   00 0A 00 00 DF 02 25 02-09 2A FF 50 F6 0A 02 FA   ...._.%..*.Pv..z
0030   B8 C0 07 8E D8 BC 00 7C-33 C0 8E D0 8E C0 FB FC   8@..X<.¦3@.P.@{¦
  .
  .
  .
0180   0A 44 69 73 6B 20 42 6F-6F 74 20 46 61 69 6C 75   .Disk Boot Failu
0190   72 65 0D 0A 0D 0A 4E 6F-6E 2D 53 79 73 74 65 6D   re....Non-System
01A0   20 64 69 73 6B 20 6F 72-20 64 69 73 6B 20 65 72    disk or disk er
01B0   72 6F 72 0D 0A 52 65 70-6C 61 63 65 20 61 6E 64   ror..Replace and
01C0   20 70 72 65 73 73 20 61-6E 79 20 6B 65 79 20 77    press any key w
01D0   68 65 6E 20 72 65 61 64-79 0D 0A 00 00 00 00 00   hen ready.......
01E0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
01F0   00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA   ...............*
```

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

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

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

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

The file allocation table

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

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

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

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

12-bit FAT:

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

16-bit FAT:

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

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

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

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

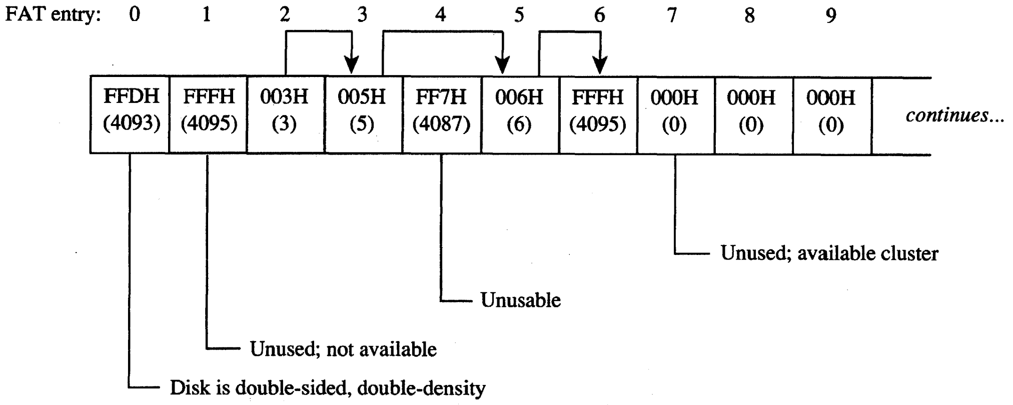


Figure 3-8. Space allocation in the FAT for a typical MS-DOS disk.

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

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

Special considerations

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

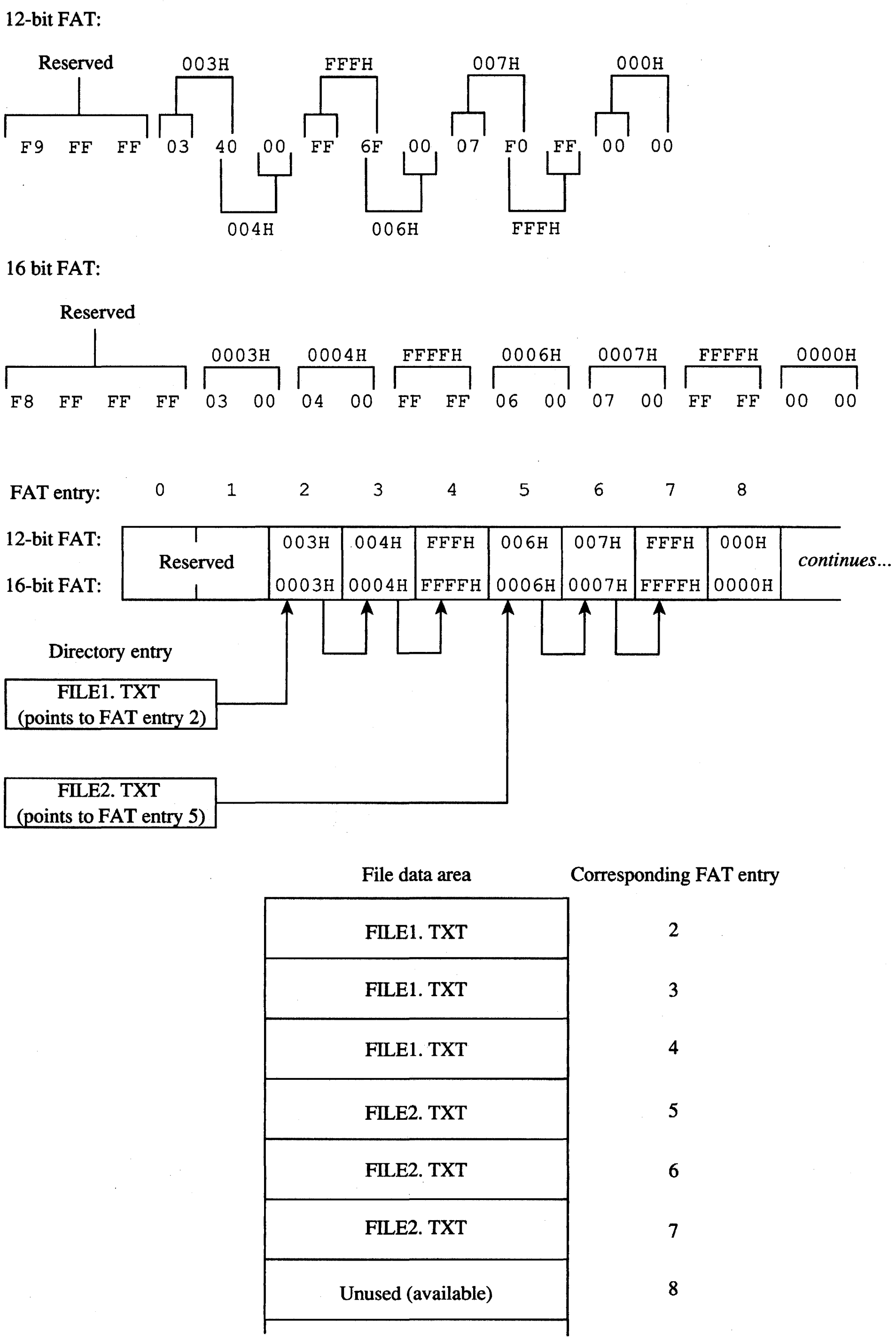


Figure 3-9. Correspondence between the FAT and the file data area.

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

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

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

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

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

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

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

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

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

The root directory

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

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

Figure 3-12 shows a dump of a 512-byte directory sector containing 16 directory entries. (Each entry occupies two lines in this example.) The byte at offset 0ABH, containing a 10H, signifies that the entry starting at 0A0H is for a subdirectory. The byte at offset 160H, containing 0E5H, means that the file has been deleted. The byte at offset 8BH, containing the value 08H, indicates that the directory entry beginning at offset 80H is a volume label. Finally the zero byte at offset 1E0H marks the end of the directory, indicating that the subsequent entries in the directory have never been used and therefore need not be searched (versions 2.0 and later).

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

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

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

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

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

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

The file area

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

Other MS-DOS Storage Devices

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

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

William Wong

Part B
Programming for MS-DOS

Article 4
Structure of an Application Program

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

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

The .EXE Program

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

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

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

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

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

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.

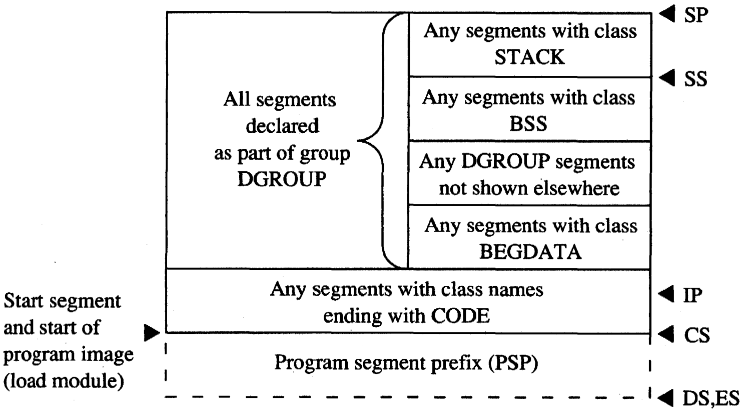

SP
SS
All segments declared as part of group DGROUP
Any segments with class STACK
Any segments with class BSS
Any DGROUP segments not shown elsewhere
Any segments with class BEGDATA
Any segments with class names ending with CODE
IP
CS
Start segment and start of program image (load module)
Program segment prefix (PSP)
DS,ES

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

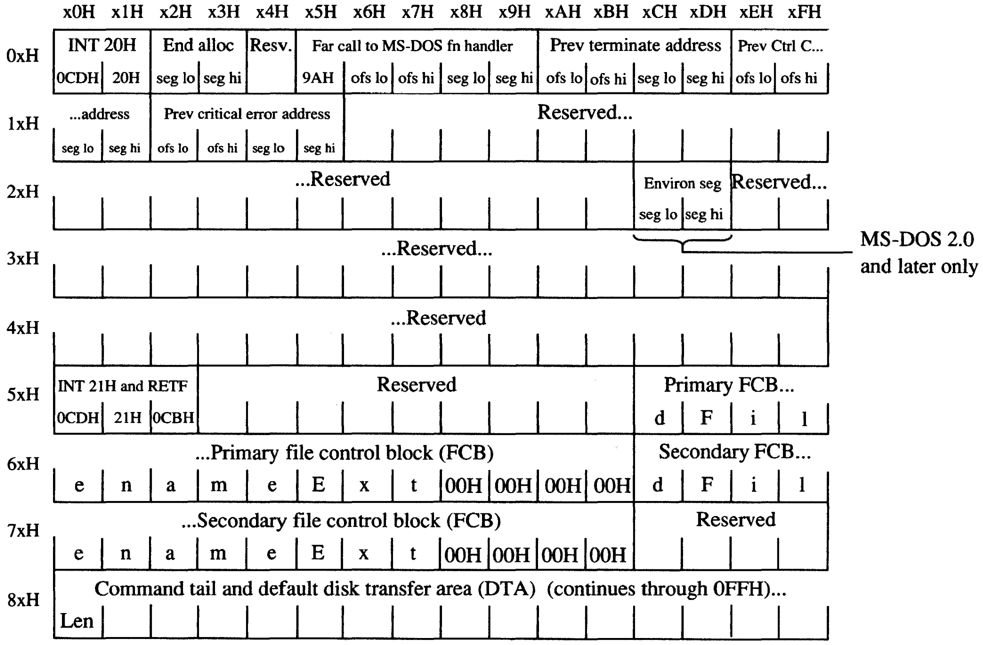
Before transferring control to the .EXE program, MS-DOS initializes various areas of memory and several of the microprocessor's registers. The following discussion explains what to expect from MS-DOS before it gives the .EXE program control.

The program segment prefix

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

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

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



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

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

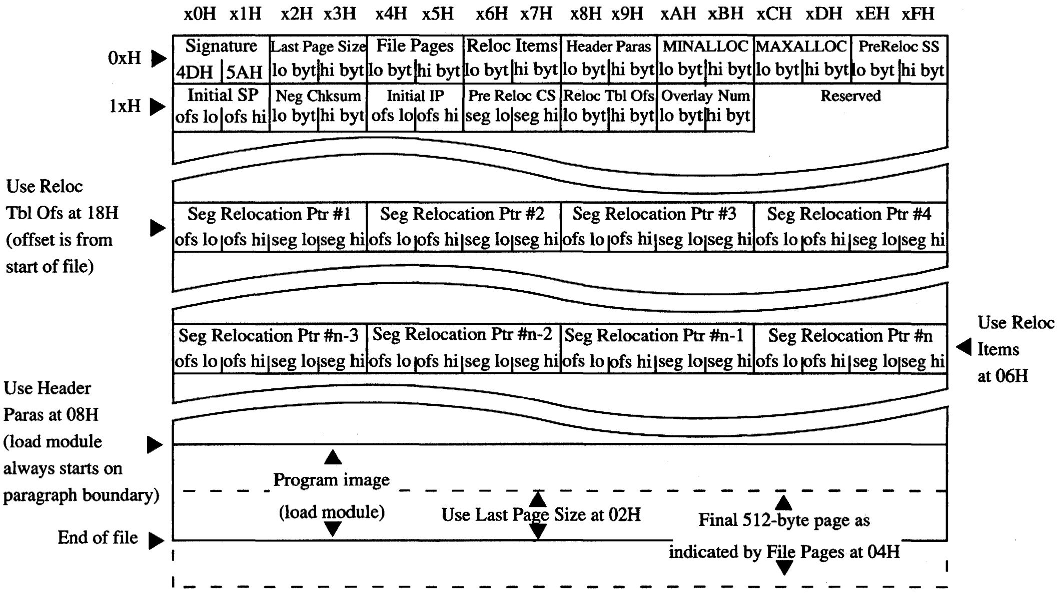

x0H x1H x2H x3H x4H x5H x6H x7H x8H x9H xAH xBH xCH xDH xEH xFH
0xH
Signature 4DH 5AH
Last Page Size lo byt hi byt
File Pages lo byt hi byt
Reloc Items lo byt hi byt
Header Paras lo byt hi byt
MINALLOC lo byt hi byt
MAXALLOC lo byt hi byt
PreReloc SS lo byt hi byt
1xH
Initial SP ofs lo ofs hi
Neg Chksum lo byt hi byt
Initial IP ofs lo ofs hi
Pre Reloc CS seg lo seg hi
Reloc Tbl Ofs lo byt hi byt
Overlay Num lo byt hi byt
Reserved
Use Reloc Tbl Ofs at 18H (offset is from start of file)
Seg Relocation Ptr #1 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #2 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #3 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #4 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #n-3 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #n-2 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #n-1 ofs lo ofs hi seg lo seg hi
Seg Relocation Ptr #n ofs lo ofs hi seg lo seg hi
Use Reloc Items at 06H
Use Header Paras at 08H (load module always starts on paragraph boundary)
Program image (load module)
Use Last Page Size at 02H
Final 512-byte page as indicated by File Pages at 04H
End of file

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1A–1BH (Overlay Number) This word is normally set to 0000H, indicating that the .EXE file consists of the resident, or primary, part of the program. This number changes only in files containing programs that use overlays, which are sections of a program that remain on disk until the program actually requires them. These program sections are loaded into memory by special overlay managing routines included in the run-time libraries supplied with some Microsoft high-level-language compilers.

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

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

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

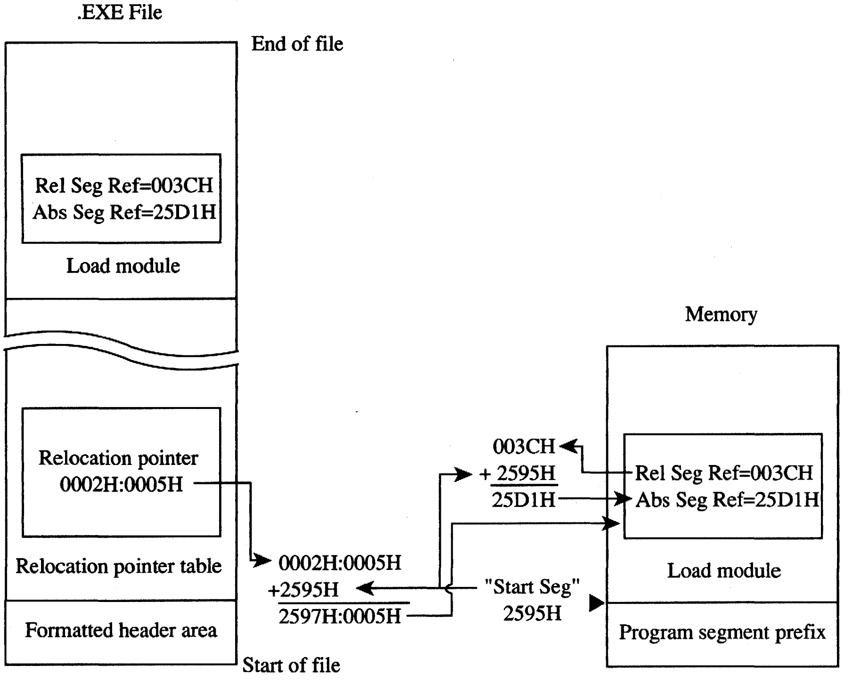


Figure 4-6. The .EXE file relocation procedure.

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

The load module

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

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

Loading the .EXE program

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

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

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

Controlling the .EXE program's structure

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

The MASM SEGMENT directive

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

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

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

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

The *align* type parameter

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

The following alignment types are permitted:

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

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

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

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

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

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

PAGE This *align* type instructs the linker to start the segment on the next page boundary. Page boundaries occur every 256 bytes and consist of all addresses in which the low address byte equals zero (0000H, 0100H, 0200H, and so forth). PAGE alignment ensures only that the linker positions the segment on a page boundary relative to the start of the load module. Unfortunately, this does not also ensure alignment of the segment on an absolute page within memory, because MS-DOS only guarantees alignment of the entire load module on a paragraph boundary.

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

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

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

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

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

The *combine* type parameter

The optional *combine* parameter allows the programmer to send directions to the linker on how to combine segments with the same *segname* occurring in different object modules. If no *combine* type is specified, the linker treats such segments as if each had a different *segname.* The *combine* type has no effect on the relationship of segments with different *segnames*. MASM and LINK both support the following *combine* types:

PUBLIC This *combine* type instructs the linker to concatenate multiple segments having the same *segname* into a single contiguous segment. The linker adjusts any address references to labels within the concatenated segments to reflect the new position of those labels relative to the start of the combined segment. This *combine* type is useful for accessing code or data in different source modules using a common segment register value.

STACK This *combine* type operates similarly to the PUBLIC *combine* type, except for two additional effects: The STACK type tells the linker that this segment comprises part of the program's stack and initialization data contained within STACK segments is handled differently than in PUBLIC segments. Declaring segments with the STACK *combine* type permits the linker to determine the initial SS and SP register values it places in the .EXE file header. Normally, a programmer would declare only one STACK segment in one of the source modules. If pieces of the stack are declared in different source modules, the linker will concatenate them in the same fashion as PUBLIC segments. However, initialization data declared within any STACK segment is placed at the high end of the combined STACK segments on a module-by-module basis. Thus, each successive module's initialization data overlays the previous module's data. At least one segment must be declared with the STACK *combine* type; otherwise, the linker will issue a warning message because it cannot determine the program's initial SS and SP values. (The warning can be ignored if the program itself initializes SS and SP.)

COMMON This *combine* type instructs the linker to overlap multiple segments having the same *segname*. The length of the resulting segment reflects the length of the longest segment declared. If any code or data is declared in the overlapping segments, the data contained in the final segments linked replaces any data in previously loaded segments. This *combine* type is useful when a data area is to be shared by code in different source modules.

MEMORY Microsoft's LINK treats this *combine* type the same as it treats the PUBLIC type. MASM, however, supports the MEMORY type for compatibility with other linkers that use Intel's definition of a MEMORY *combine* type.

AT address This *combine* type instructs LINK to pretend that the segment will reside at the absolute segment *address*. LINK then adjusts all address references to the segment in accordance with the masquerade. LINK will *not* create an image of the segment in the load module, and it will ignore any data defined within the segment. This behavior is consistent with the fact that MS-DOS does not support the loading of program segments into absolute memory segments. All programs must be able to execute from any segment address at which MS-DOS can find available memory. The SEGMENT AT address *combine* type is useful for creating templates of various areas in memory outside the program. For instance, *SEGMENT AT 0000H* could be used to create a template of the 8086-family interrupt vectors. Because data contained within SEGMENT AT address segments is suppressed by LINK and not by MASM (which places the data in the object module), it is possible to use .OBJ files generated by MASM with another linker that supports ROM or other absolute code generation should the programmer require this specialized capability.

The *class* type parameter

The *class* parameter provides the means to organize different segments into classifications. For instance, here are three source modules, each with its own separate code and data segments:

```
;Module "A"
A_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "A" data fields
A_DATA  ENDS
A_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "A" code
A_CODE  ENDS
        END
```

(more)

```
;Module "B"
B_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "B" data fields
B_DATA  ENDS
B_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "B" code
B_CODE  ENDS
        END

;Module "C"
C_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "C" data fields
C_DATA  ENDS
C_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "C" code
C_CODE  ENDS
        END
```

If the 'CODE' and 'DATA' *class* types are removed from the SEGMENT directives shown above, the linker organizes the segments as it encounters them. If the programmer specifies the modules to the linker in alphabetic order, the linker produces the following segment ordering:

```
A_DATA
A_CODE
B_DATA
B_CODE
C_DATA
C_CODE
```

However, if the programmer specifies the *class* types shown in the sample source modules, the linker organizes the segments by classification as follows:

```
'DATA' class:   A_DATA
                B_DATA
                C_DATA

'CODE' class:   A_CODE
                B_CODE
                C_CODE
```

Notice that the linker still organizes the classifications in the order in which it encounters the segments belonging to the various classifications. To completely control the order in which the linker organizes the segments, the programmer must use one of three basic approaches. The preferred method involves using the /DOSSEG switch with the linker. This produces the segment ordering shown in Figure 4-1. The second method involves creating a special source module that contains empty SEGMENT–ENDS blocks for all the segments declared in the various other source modules. The programmer creates the list in the order the segments are to be arranged in memory and then specifies the .OBJ file for this module as the first file for the linker to process. This procedure establishes the order of all the segments before LINK begins processing the other program modules, so the programmer can declare segments in these other modules in any convenient order. For instance, the following source module rearranges the result of the previous example so that the linker places the 'CODE' class before the 'DATA' class:

```
A_CODE   SEGMENT PARA PUBLIC 'CODE'
A_CODE   ENDS
B_CODE   SEGMENT PARA PUBLIC 'CODE'
B_CODE   ENDS
C_CODE   SEGMENT PARA PUBLIC 'CODE'
C_CODE   ENDS

A_DATA   SEGMENT PARA PUBLIC 'DATA'
A_DATA   ENDS
B_DATA   SEGMENT PARA PUBLIC 'DATA'
B_DATA   ENDS
C_DATA   SEGMENT PARA PUBLIC 'DATA'
C_DATA   ENDS

         END
```

Rather than creating a new module, the third method places the same segment ordering list shown above at the start of the first module containing actual code or data that the programmer will be specifying for the linker. This duplicates the approach used by Microsoft's newer compilers, such as C version 4.0.

The ordering of segments within the load module has no direct effect on the linker's adjustment of address references to locations within the various segments. Only the GROUP directive and the SEGMENT directive's *combine* parameter affect address adjustments performed by the linker. *See* The MASM GROUP Directive below.

Note: Certain older versions of the IBM Macro Assembler wrote segments to the object file in alphabetic order regardless of their order in the source file. These older versions can limit efforts to control segment ordering. Upgrading to a new version of the assembler is the best solution to this problem.

Ordering segments to shrink the .EXE file

Correct segment ordering can significantly decrease the size of a .EXE program as it resides on disk. This size-reduction ordering is achieved by placing all uninitialized data fields in their own segments and then controlling the linker's ordering of the program's segments so that the uninitialized data field segments all reside at the end of the program. When the program modules are assembled, MASM places information in the object modules to tell the linker about initialized and uninitialized areas of all segments. The linker then uses this information to prevent the writing of uninitialized data areas that occur at the end of the program image as part of the resulting .EXE file. To account for the memory space required by these fields, the linker also sets the MINALLOC field in the .EXE file header to represent the data area not written to the file. MS-DOS then uses the MINALLOC field to reallocate this missing space when loading the program.

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:

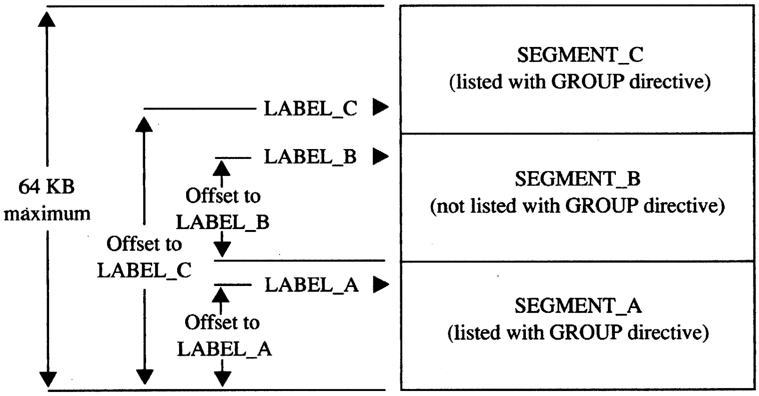


Figure 4-7. Noncontiguous segments in the same GROUP.

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.

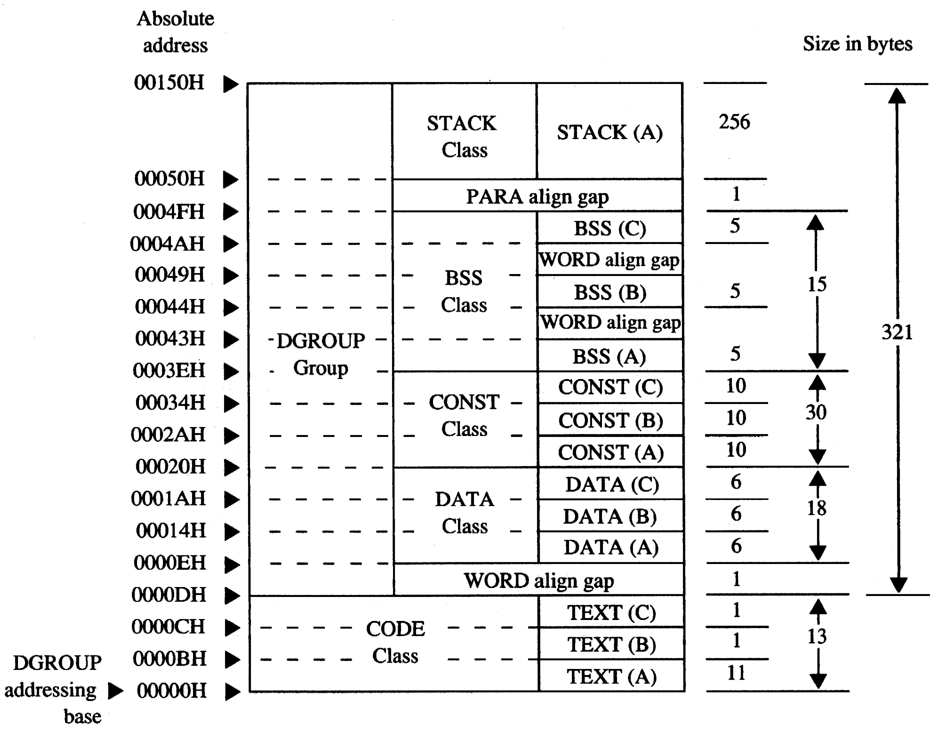


Figure 4-12. Structure of the sample .EXE program.

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.

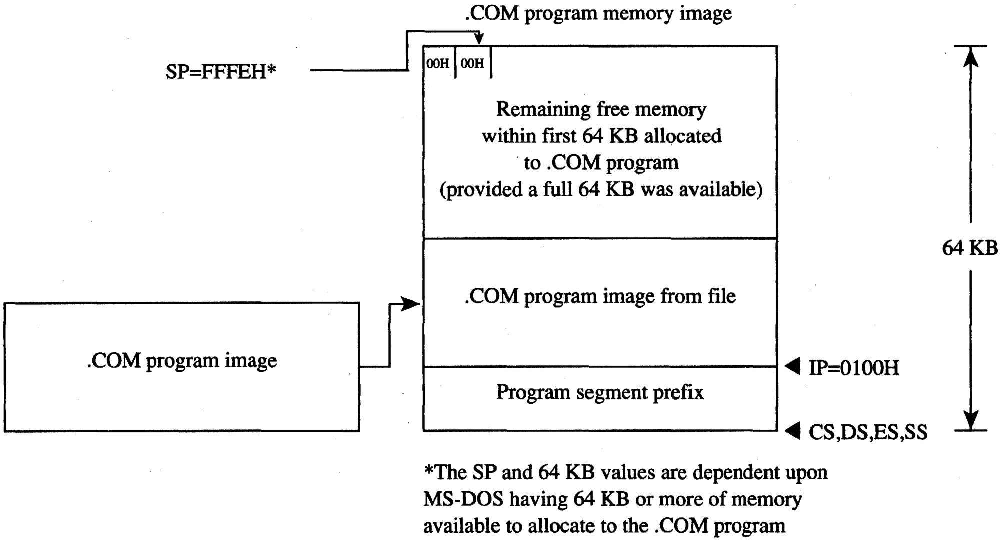
Giving control to the .COM program

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.


.COM program memory image
SP=FFFEH*
00H
00H
Remaining free memory within first 64 KB allocated to .COM program (provided a full 64 KB was available)
.COM program image from file
Program segment prefix
.COM program image
IP=0100H
CS,DS,ES,SS
64 KB*

*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 (*stdin*)	0	CON
Standard output (*stdout*)	1	CON
Standard error (*stderr*)	2	CON
Standard auxiliary (*stdaux*)	3	AUX
Standard printer (*stdprn*)	4	PRN

The standard input and standard output handles are especially important because they are subject to I/O redirection. Although these handles are associated by default with the CON device so that read and write operations are implemented using the keyboard and video display, the user can associate the handles with other character devices or with files by using redirection parameters in a program's command line:

Redirection	Result
< *file*	Causes read operations from standard input to obtain data from *file*.
> *file*	Causes data written to standard output to be placed in *file*.
>> *file*	Causes data written to standard output to be appended to *file*.
p1 ¦ *p2*	Causes data written to standard output by program *p1* to appear as the standard input of program *p2*.

This ability to redirect I/O adds great flexibility and power to the system. For example, programs ordinarily controlled by keyboard entries can be run with "scripts" from files, the output of a program can be captured in a file or on a printer for later inspection, and general-purpose programs (filters) can be written that process text streams without regard to the text's origin or destination. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Writing MS-DOS Filters.

Ordinarily, an application program is not aware that its input or output has been redirected, although a write operation to standard output will fail unexpectedly if standard output was redirected to a disk file and the disk is full. An application can check for the existence of I/O redirection with an IOCTL (Interrupt 21H Function 44H) call, but it cannot obtain any information about the destination of the redirected handle except whether it is associated with a character device or with a file.

Raw versus cooked mode

MS-DOS associates each handle for a character device with a mode that determines how I/O requests directed to that handle are treated. When a handle is in raw mode, characters are passed between the application program and the device driver without any filtering or buffering by MS-DOS. When a handle is in cooked mode, MS-DOS buffers any data that is read from or written to the device and takes special actions when certain characters are detected.

During cooked mode input, MS-DOS obtains characters from the device driver one at a time, checking each character for a Control-C. The characters are assembled into a string within an internal MS-DOS buffer. The input operation is terminated when a carriage return (0DH) or an end-of-file mark (1AH) is received or when the number of characters requested by the application have been accumulated. If the source is standard input, lone linefeed characters are translated to carriage-return/linefeed pairs. The string is then copied from the internal MS-DOS buffer to the application program's buffer, and control returns to the application program.

During cooked mode output, MS-DOS transfers the characters in the application program's output buffer to the device driver one at a time, checking after each character for a Control-C pending at the keyboard. If the destination is standard output and standard output has not been redirected, tabs are expanded to spaces using eight-column tab stops. Output is terminated when the requested number of characters have been written or when an end-of-file mark (1AH) is encountered in the output string.

In contrast, during raw mode input or output, data is transferred directly between the application program's buffer and the device driver. Special characters such as carriage return and the end-of-file mark are ignored, and the exact number of characters in the application program's request are always read or written. MS-DOS does not break the strings into single-character calls to the device driver and does not check the keyboard buffer for Control-C entries during the I/O operation. Finally, characters read from standard input in raw mode are not echoed to standard output.

As might be expected from the preceding description, raw mode input or output is usually much faster than cooked mode input or output, because each character is not being individually processed by the MS-DOS kernel. Raw mode also allows programs to read characters from the keyboard buffer that would otherwise be trapped by MS-DOS (for example, Control-C, Control-P, and Control-S). (If BREAK is on, MS-DOS will still check for Control-C entries during other function calls, such as disk operations, and transfer control to the Control-C exception handler if a Control-C is detected.) A program can use the MS-DOS IOCTL Get and Set Device Data services (Interrupt 21H Function 44H Subfunctions 00H and 01H) to set the mode for a character-device handle. *See* IOCTL below.

Ordinarily, raw or cooked mode is strictly an attribute of a specific handle that was obtained from a previous open operation and affects only the I/O operations requested by the program that owns the handle. However, when a program uses IOCTL to select raw or cooked mode for one of the standard device handles, the selection has a global effect on the behavior of the system because those handles are never closed. Thus, some of the "traditional" keyboard input functions might behave in unexpected ways. Consequently, programs that change the mode on a standard device handle should save the handle's mode at entry and restore it before performing a final exit to MS-DOS, so that the operation of COMMAND.COM and other applications will not be disturbed. Such programs should also incorporate custom critical error and Control-C exception handlers so that the programs cannot be terminated unexpectedly. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

The keyboard

Among the MS-DOS Interrupt 21H functions are two methods of checking for and receiving input from the keyboard: the traditional method, which uses MS-DOS character input Functions 01H, 06H, 07H, 08H, 0AH, 0BH, and 0CH (Table 5-1); and the handle method, which uses Function 3FH. Each of these methods has its own advantages and disadvantages. *See* SYSTEM CALLS.

Table 5-1. Traditional MS-DOS Character Input Functions.

Function	Name	Read Multiple Characters	Echo	Ctrl-C Check
01H	Character Input with Echo	No	Yes	Yes
06H	Direct Console I/O	No	No	No
07H	Unfiltered Character Input Without Echo	No	No	No
08H	Character Input Without Echo	No	No	Yes
0AH	Buffered Keyboard Input	Yes	Yes	Yes
0BH	Check Keyboard Status	No	No	Yes
0CH	Flush Buffer, Read Keyboard	*	*	*

* Varies depending on function (from above) called in the AL register.

The first four traditional keyboard input calls are really very similar. They all return a character in the AL register; they differ mainly in whether they echo that character to the display and whether they are sensitive to interruption by the user's entry of a Control-C. Both Functions 06H and 0BH can be used to test keyboard status (that is, whether a key has been pressed and is waiting to be read by the program); Function 0BH is simpler to use, but Function 06H is immune to Control-C entries.

Function 0AH is used to read a "buffered line" from the user, meaning that an entire line is accepted by MS-DOS before control returns to the program. The line is terminated when the user presses the Enter key or when the maximum number of characters (to 255) specified by the program have been received. While entry of the line is in progress, the usual editing keys (such as the left and right arrow keys and the function keys on IBM PCs and compatibles) are active; only the final, edited line is delivered to the requesting program.

Function 0CH allows a program to flush the type-ahead buffer before accepting input. This capability is important for occasions when a prompt must be displayed unexpectedly (such as when a critical error occurs) and the user could not have typed ahead a valid response. This function should also be used when the user is being prompted for a critical decision (such as whether to erase a file), to prevent a character that was previously pressed by accident from triggering an irrecoverable operation. Function 0CH is unusual in that it is called with the number of one of the other keyboard input functions in register AL. After any pending input has been discarded, Function 0CH simply transfers to the other specified input function; thus, its other parameters (if any) depend on the function that ultimately will be executed.

The primary disadvantage of the traditional function calls is that they handle redirected input poorly. If standard input has been redirected to a file, no way exists for a program calling the traditional input functions to detect that the end of the file has been reached — the input function will simply wait forever, and the system will appear to hang.

A program that wishes to use handle-based I/O to get input from the keyboard must use the MS-DOS Read File or Device service, Interrupt 21H Function 3FH. Ordinarily, the program can employ the predefined handle for standard input (0), which does not need to be opened and which allows the program's input to be redirected by the user to another file or device. If the program needs to circumvent redirection and ensure that its input is from the keyboard, it can open the CON device with Interrupt 21H Function 3DH and use the handle obtained from that open operation instead of the standard input handle.

A program using the handle functions to read the keyboard can control the echoing of characters and sensitivity to Control-C entries by selecting raw or cooked mode with the IOCTL Get and Set Device Data services (default = cooked mode). To test the keyboard status, the program can either issue an IOCTL Check Input Status call (Interrupt 21H Function 44H Subfunction 06H) or use the traditional Check Keyboard Status call (Interrupt 21H Function 0BH).

The primary advantages of the handle method for keyboard input are its symmetry with file operations and its graceful handling of redirected input. The handle function also allows strings as long as 65535 bytes to be requested; the traditional Buffered Keyboard Input function allows a maximum of 255 characters to be read at a time. This consideration is important for programs that are frequently used with redirected input and output (such as filters), because reading and writing larger blocks of data from files results in more efficient operation. The only real disadvantage to the handle method is that it is limited to MS-DOS versions 2.0 and later (although this is no longer a significant restriction).

Role of the ROM BIOS

When a key is pressed on the keyboard of an IBM PC or compatible, it generates a hardware interrupt (09H) that is serviced by a routine in the ROM BIOS. The ROM BIOS interrupt handler reads I/O ports assigned to the keyboard controller and translates the key's scan code into an ASCII character code. The result of this translation depends on the current state of the NumLock and CapsLock toggles, as well as on whether the Shift, Control, or Alt key is being held down. (The ROM BIOS maintains a keyboard flags byte at address 0000:0417H that gives the current status of each of these modifier keys.)

After translation, both the scan code and the ASCII code are placed in the ROM BIOS's 32-byte (16-character) keyboard input buffer. In the case of "extended" keys such as the function keys or arrow keys, the ASCII code is a zero byte and the scan code carries all the information. The keyboard buffer is arranged as a circular, or ring, buffer and is managed as a first-in/first-out queue. Because of the method used to determine when the buffer is empty, one position in the buffer is always wasted; the maximum number of characters that can be held in the buffer is therefore 15. Keys pressed when the buffer is full are discarded and a warning beep is sounded.

The ROM BIOS provides an additional module, invoked by software Interrupt 16H, that allows programs to test keyboard status, determine whether characters are waiting in the type-ahead buffer, and remove characters from the buffer. *See* Appendix O: IBM PC BIOS Calls. Its use by application programs should ordinarily be avoided, however, to prevent introducing unnecessary hardware dependence.

On IBM PCs and compatibles, the keyboard input portion of the CON driver in the BIOS is a simple sequence of code that calls ROM BIOS Interrupt 16H to do the hardware-dependent work. Thus, calls to MS-DOS for keyboard input by an application program are subject to two layers of translation: The Interrupt 21H function call is converted by the MS-DOS kernel to calls to the CON driver, which in turn remaps the request onto a ROM BIOS call that obtains the character.

Keyboard programming examples

Example: Use the ROM BIOS keyboard driver to read a character from the keyboard. The character is not echoed to the display.

```
mov     ah,00h          ; subfunction 00H = read character
int     16h             ; transfer to ROM BIOS
                        ; now AH = scan code, AL = character
```

Example: Use the MS-DOS traditional keyboard input function to read a character from the keyboard. The character is not echoed to the display. The input can be interrupted with a Ctrl-C keystroke.

```
mov     ah,08h          ; function 08H = character input
                        ;   without echo
int     21h             ; transfer to MS-DOS
                        ; now AL = character
```

Example: Use the MS-DOS traditional Buffered Keyboard Input function to read an entire line from the keyboard, specifying a maximum line length of 80 characters. All editing keys are active during entry, and the input is echoed to the display.

```
kbuf    db      80              ; maximum length of read
        db      0               ; actual length of read
        db      80 dup (0)      ; keyboard input goes here
        .
        .
        .
        mov     dx,seg kbuf     ; set DS:DX = address of
        mov     ds,dx           ; keyboard input buffer
        mov     dx,offset kbuf
        mov     ah,0ah          ; function 0AH = read buffered line
        int     21h             ; transfer to MS-DOS
                                ; terminated by a carriage return,
                                ; and kbuf+1 = length of input,
                                ; not including the carriage return
```

Example: Use the MS-DOS handle-based Read File or Device function and the standard input handle to read an entire line from the keyboard, specifying a maximum line length of 80 characters. All editing keys are active during entry, and the input is echoed to the display. (The input will not terminate on a carriage return as expected if standard input is in raw mode.)

```
kbuf    db      80 dup (0)      ; buffer for keyboard input
        .
        .
        .
        mov     dx,seg kbuf     ; set DS:DX = address of
        mov     ds,dx           ; keyboard input buffer
        mov     dx,offset kbuf
        mov     cx,80           ; CX = maximum length of input
        mov     bx,0            ; standard input handle = 0
        mov     ah,3fh          ; function 3FH = read file/device
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if function failed
                                ; otherwise AX = actual
                                ; length of keyboard input,
                                ; including carriage-return and
                                ; linefeed, and the data is
                                ; in the buffer 'kbuf'
```

The display

The output half of the MS-DOS logical character device CON is the video display. On IBM PCs and compatibles, the video display is an "option" of sorts that comes in several forms. IBM has introduced five video subsystems that support different types of displays: the Monochrome Display Adapter (MDA), the Color/Graphics Adapter (CGA), the Enhanced Graphics Adapter (EGA), the Video Graphics Array (VGA), and the Multi-Color Graphics Array (MCGA). Other, non-IBM-compatible video subsystems in common use include the Hercules Graphics Card and its variants that support downloadable fonts.

Two portable techniques exist for writing text to the video display with MS-DOS function calls. The traditional method is supported by Interrupt 21H Functions 02H (Character Output), 06H (Direct Console I/O), and 09H (Display String). The handle method is supported by Function 40H (Write File or Device) and is available only in MS-DOS versions 2.0 and later. *See* SYSTEM CALLS: INTERRUPT 21H: Functions 02H, 06H, 09H, 40H. All these calls treat the display essentially as a "glass teletype" and do not support bit-mapped graphics.

Traditional Functions 02H and 06H are similar. Both are called with the character to be displayed in the DL register; they differ in that Function 02H is sensitive to interruption by the user's entry of a Control-C, whereas Function 06H is immune to Control-C but cannot be used to output the character 0FFH (ASCII rubout). Both calls check specifically for carriage return (0DH), linefeed (0AH), and backspace (08H) characters and take the appropriate action if these characters are detected.

Because making individual calls to MS-DOS for each character to be displayed is inefficient and slow, the traditional Display String function (09H) is generally used in preference to Functions 02H and 06H. Function 09H is called with the address of a string that is terminated with a dollar-sign character ($); it displays the entire string in one operation, regardless of its length. The string can contain embedded control characters such as carriage return and linefeed.

To use the handle method for screen display, programs must call the MS-DOS Write File or Device service, Interrupt 21H Function 40H. Ordinarily, a program should use the predefined handle for standard output (1) to send text to the screen, so that any redirection requested by the user on the program's command line will be honored. If the program needs to circumvent redirection and ensure that its output goes to the screen, it can either use the predefined handle for standard error (2) or explicitly open the CON device with Interrupt 21H Function 3DH and use the resulting handle for its write operations.

The handle technique for displaying text has several advantages over the traditional calls. First, the length of the string to be displayed is passed as an explicit parameter, so the string need not contain a special terminating character and the $ character can be displayed as part of the string. Second, the traditional calls are translated to handle calls inside MS-DOS, so the handle calls have less internal overhead and are generally faster. Finally, use of the handle Write File or Device function to display text is symmetric with the methods the program must use to access its files. In short, the traditional functions should be avoided unless the program must be capable of running under MS-DOS versions 1.x.

Controlling the screen

One of the deficiencies of the standard MS-DOS CON device driver is the lack of screen-control capabilities. The default CON driver has no built-in routines to support cursor placement, screen clearing, display mode selection, and so on.

In MS-DOS versions 2.0 and later, an optional replacement CON driver is supplied in the file ANSI.SYS. This driver contains most of the screen-control capabilities needed by text-oriented application programs. The driver is installed by adding a DEVICE directive to the

CONFIG.SYS file and restarting the system. When ANSI.SYS is active, a program can position the cursor, inquire about the current cursor position, select foreground and background colors, and clear the current line or the entire screen by sending an escape sequence consisting of the ASCII Esc character (1BH) followed by various function-specific parameters to the standard output device. *See* USER COMMANDS: ANSI.SYS.

Programs that use the ANSI.SYS capabilities for screen control are portable to any MS-DOS implementation that contains the ANSI.SYS driver. Programs that seek improved performance by calling the ROM BIOS video driver or by assuming direct control of the hardware are necessarily less portable and usually require modification when new PC models or video subsystems are released.

Role of the ROM BIOS

The video subsystems in IBM PCs and compatibles use a hybrid of memory-mapped and port-addressed I/O. A range of the machine's memory addresses is typically reserved for a video refresh buffer that holds the character codes and attributes to be displayed on the screen; the cursor position, display mode, palettes, and similar global display characteristics are governed by writing control values to specific I/O ports.

The ROM BIOS of IBM PCs and compatibles contains a primitive driver for the MDA, CGA, EGA, VGA, and MCGA video subsystems. This driver supports the following functions:

- Read or write characters with attributes at any screen position.
- Query or set the cursor position.
- Clear or scroll an arbitrary portion of the screen.
- Select palette, background, foreground, and border colors.
- Query or set the display mode (40-column text, 80-column text, all-points-addressable graphics, and so on).
- Read or write a pixel at any screen coordinate.

These functions are invoked by a program through software Interrupt 10H. *See* Appendix O: IBM PC BIOS Calls. In PC-DOS-compatible implementations of MS-DOS, the display portions of the MS-DOS CON and ANSI.SYS drivers use these ROM BIOS routines. Video subsystems that are not IBM compatible either must contain their own ROM BIOS or must be used with an installable device driver that captures Interrupt 10H and provides appropriate support functions.

Text-only application programs should avoid use of the ROM BIOS functions or direct access to the hardware whenever possible, to ensure maximum portability between MS-DOS systems. However, because the MS-DOS CON driver contains no support for bit-mapped graphics, graphically oriented applications usually must resort to direct control of the video adapter and its refresh buffer for speed and precision.

Display programming examples

Example: Use the ROM BIOS Interrupt 10H function to write an asterisk character to the display in text mode. (In graphics mode, BL must also be set to the desired foreground color.)

```
        mov     ah,0eh          ; subfunction 0EH = write character
                                ; in teletype mode
        mov     al,'*'          ; AL = character to display
        mov     bh,0            ; select display page 0
        int     10h             ; transfer to ROM BIOS video driver
```

Example: Use the MS-DOS traditional function to write an asterisk character to the display. If the user's entry of a Control-C is detected during the output and standard output is in cooked mode, MS-DOS calls the Control-C exception handler whose address is found in the vector for Interrupt 23H.

```
        mov     ah,02h          ; function 02H = display character
        mov     dl,'*'          ; DL = character to display
        int     21h             ; transfer to MS-DOS
```

Example: Use the MS-DOS traditional function to write a string to the display. The output is terminated by the $ character and can be interrupted when the user enters a Control-C if standard output is in cooked mode.

```
msg     db      'This is a test message','$'
        .
        .
        .
        mov     dx,seg msg      ; DS:DX = address of text
        mov     ds,dx           ; to display
        mov     dx,offset msg
        mov     ah,09h          ; function 09H = display string
        int     21h             ; transfer to MS-DOS
```

Example: Use the MS-DOS handle-based Write File or Device function and the predefined handle for standard output to write a string to the display. Output can be interrupted by the user's entry of a Control-C if standard output is in cooked mode.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     dx,seg msg      ; DS:DX = address of text
        mov     ds,dx           ; to display
        mov     dx,offset msg
        mov     cx,msg_len      ; CX = length of text
        mov     bx,1            ; BX = handle for standard output
        mov     ah,40h          ; function 40H = write file/device
        int     21h             ; transfer to MS-DOS
```

The serial communications ports

Through version 3.2, MS-DOS has built-in support for two serial communications ports, identified as COM1 and COM2, by means of three drivers named AUX, COM1, and COM2. (AUX is ordinarily an alias for COM1.)

The traditional MS-DOS method of reading from and writing to the serial ports is through Interrupt 21H Function 03H for AUX input and Function 04H for AUX output. In MS-DOS versions 2.0 and later, the handle-based Read File or Device and Write File or Device functions (Interrupt 21H Functions 3FH and 40H) can be used to read from or write to the auxiliary device. A program can use the predefined handle for the standard auxiliary device (3) with Functions 3FH and 40H, or it can explicitly open the COM1 or COM2 devices with Interrupt 21H Function 3DH and use the handle obtained from that open operation to perform read and write operations.

MS-DOS support for the serial communications port is inadequate in several respects for high-performance serial I/O applications. First, MS-DOS provides no portable way to test for the existence or the status of a particular serial port in a system; if a program "opens" COM2 and writes data to it and the physical COM2 adapter is not present in the system, the program may simply hang. Similarly, if the serial port exists but no character has been received and the program attempts to read a character, the program will hang until one is available; there is no traditional function call to check if a character is waiting as there is for the keyboard.

MS-DOS also provides no portable method to initialize the communications adapter to a particular baud rate, word length, and parity. An application must resort to ROM BIOS calls, manipulate the hardware directly, or rely on the user to configure the port properly with the MODE command before running the application that uses it. The default settings for the serial port on PC-DOS-compatible systems are 2400 baud, no parity, 1 stop bit, and 8 databits. *See* USER COMMANDS: MODE.

A more serious problem with the default MS-DOS auxiliary device driver in IBM PCs and compatibles, however, is that it is not interrupt driven. Accordingly, when baud rates above 1200 are selected, characters can be lost during time-consuming operations performed by the drivers for other devices, such as clearing the screen or reading or writing a floppy-disk sector. Because the MS-DOS AUX device driver typically relies on the ROM BIOS serial port driver (accessed through software Interrupt 14H) and because the ROM BIOS driver is not interrupt driven either, bypassing MS-DOS and calling the ROM BIOS functions does not usually improve matters.

Because of all the problems just described, telecommunications application programs commonly take over complete control of the serial port and supply their own interrupt handler and internal buffering for character read and write operations. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Interrupt-Driven Communications.

Serial port programming examples

Example: Use the ROM BIOS serial port driver to write a string to COM1.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     bx,seg msg      ; DS:BX = address of message
        mov     ds,bx
        mov     bx,offset msg
        mov     cx,msg_len      ; CX = length of message
        mov     dx,0            ; DX = 0 for COM1
L1:     mov     al,[bx]         ; get next character into AL
        mov     ah,01h          ; subfunction 01H = output
        int     14h             ; transfer to ROM BIOS
        inc     bx              ; bump pointer to output string
        loop    L1              ; and loop until all chars. sent
```

Example: Use the MS-DOS traditional function for auxiliary device output to write a string to COM1.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     bx,seg msg      ; set DS:BX = address of message
        mov     ds,bx
        mov     bx,offset msg
        mov     cx,msg_len      ; set CX = length of message
L1:     mov     dl,[bx]         ; get next character into DL
        mov     ah,04h          ; function 04H = auxiliary output
        int     21h             ; transfer to MS-DOS
        inc     bx              ; bump pointer to output string
        loop    L1              ; and loop until all chars. sent
```

Example: Use the MS-DOS handle-based Write File or Device function and the predefined handle for the standard auxiliary device to write a string to COM1.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     dx,seg msg      ; DS:DX = address of message
        mov     ds,dx
        mov     dx,offset msg
        mov     cx,msg_len      ; CX = length of message
        mov     bx,3            ; BX = handle for standard aux.
        mov     ah,40h          ; function 40H = write file/device
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if write operation failed
```

The parallel port and printer

Most MS-DOS implementations contain device drivers for four printer devices: LPT1, LPT2, LPT3, and PRN. PRN is ordinarily an alias for LPT1 and refers to the first parallel output port in the system. To provide for list devices that do not have a parallel interface, the LPT devices can be individually redirected with the MODE command to one of the serial communications ports. *See* USER COMMANDS: MODE.

As with the keyboard, the display, and the serial port, MS-DOS allows the printer to be accessed with either traditional or handle-based function calls. The traditional function call is Interrupt 21H Function 05H, which accepts a character in DL and sends it to the physical device currently assigned to logical device name LPT1.

A program can perform handle-based output to the printer with Interrupt 21H Function 40H (Write File or Device). The predefined handle for the standard printer (4) can be used to send strings to logical device LPT1. Alternatively, the program can issue an open operation for a specific printer device with Interrupt 21H Function 3DH and use the handle obtained from that open operation with Function 40H. This latter method also allows more than one printer to be used at a time from the same program.

Because the parallel ports are assumed to be output only, no traditional call exists for input from the parallel port. In addition, no portable method exists to test printer port status under MS-DOS; programs that wish to avoid sending a character to the printer adapter when it is not ready or not physically present in the system must test the adapter's status by making a call to the ROM BIOS printer driver (by means of software Interrupt 17H; *see* Appendix O: IBM PC BIOS Calls) or by accessing the hardware directly.

Parallel port programming examples

Example: Use the ROM BIOS printer driver to send a string to the first parallel printer port.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     bx,seg msg      ; DS:BX = address of message
        mov     ds,bx
        mov     bx,offset msg
        mov     cx,msg_len      ; CX = length of message
        mov     dx,0            ; DX = 0 for LPT1
L1:     mov     al,[bx]         ; get next character into AL
        mov     ah,00h          ; subfunction 00H = output
        int     17h             ; transfer to ROM BIOS
        inc     bx              ; bump pointer to output string
        loop    L1              ; and loop until all chars. sent
```

Example: Use the traditional MS-DOS function call to send a string to the first parallel printer port.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     bx,seg msg      ; DS:BX = address of message
        mov     ds,bx
        mov     bx,offset msg
        mov     cx,msg_len      ; CX = length of message
L1:     mov     dl,[bx]         ; get next character into DL
        mov     ah,05h          ; function 05H = printer output
        int     21h             ; transfer to MS-DOS
        inc     bx              ; bump pointer to output string
        loop    L1              ; and loop until all chars. sent
```

Example: Use the handle-based MS-DOS Write File or Device call and the predefined handle for the standard printer to send a string to the system list device.

```
msg     db      'This is a test message'
msg_len equ     $-msg
        .
        .
        .
        mov     dx,seg msg      ; DS:DX = address of message
        mov     ds,dx
        mov     dx,offset msg
        mov     cx,msg_len      ; CX = length of message
        mov     bx,4            ; BX = handle for standard printer
        mov     ah,40h          ; function 40H = write file/device
        int     21h             ; transfer to MS-DOS
        jc      error           ; jump if write operation failed
```

IOCTL

In versions 2.0 and later, MS-DOS has provided applications with the ability to communicate directly with device drivers through a set of subfunctions grouped under Interrupt 21H Function 44H (IOCTL). *See* SYSTEM CALLS: INTERRUPT 21H: Function 44H. The IOCTL subfunctions that are particularly applicable to the character I/O needs of application programs are

Subfunction	Name
00H	Get Device Data
01H	Set Device Data
02H	Receive Control Data from Character Device

(more)

Subfunction	Name
03H	Send Control Data to Character Device
06H	Check Input Status
07H	Check Output Status
0AH	Check if Handle is Remote (version 3.1 or later)
0CH	Generic I/O Control for Handles: Get/Set Output Iteration Count

Various bits in the device information word returned by Subfunction 00H can be tested by an application to determine whether a specific handle is associated with a character device or a file and whether the driver for the device can process control strings passed by Subfunctions 02H and 03H. The device information word also allows the program to test whether a character device is the CLOCK$, standard input, standard output, or NUL device and whether the device is in raw or cooked mode. The program can then use Subfunction 01H to select raw mode or cooked mode for subsequent I/O performed with the handle.

Subfunctions 02H and 03H allow control strings to be passed between the device driver and an application; they do not usually result in any physical I/O to the device. For example, a custom device driver might allow an application program to configure the serial port by writing a specific set of control parameters to the driver with Subfunction 03H. Similarly, the custom driver might respond to Subfunction 02H by passing the application a series of bytes that defines the current configuration and status of the serial port.

Subfunctions 06H and 07H can be used by application programs to test whether a device is ready to accept an output character or has a character ready for input. These subfunctions are particularly applicable to the serial communications ports and parallel printer ports because MS-DOS does not supply traditional function calls to test their status.

Subfunction 0AH can be used to determine whether the character device associated with a handle is local or remote—that is, attached to the computer the program is running on or attached to another computer on a local area network. A program should not ordinarily attempt to distinguish between local and remote devices during normal input and output, but the information can be useful in attempts to recover from error conditions. This subfunction is available only if Microsoft Networks is running.

Finally, Subfunction 0CH allows a program to query or set the number of times a device driver tries to send output to the printer before assuming the device is not available.

IOCTL programming examples

Example: Use IOCTL Subfunction 00H to obtain the device information word for the standard input handle and save it, and then use Subfunction 01H to place standard input into raw mode.

```
info    dw      ?               ; save device information word here
        .
        .
        .
```

(more)

```
        mov     ax,4400h        ; AH = function 44H, IOCTL
                                ; AL = subfunction 00H, get device
                                ; information word
        mov     bx,0            ; BX = handle for standard input
        int     21h             ; transfer to MS-DOS
        mov     info,dx         ; save device information word
                                ; (assumes DS = data segment)
        or      dl,20h          ; set raw mode bit
        mov     dh,0            ; and clear DH as MS-DOS requires
        mov     ax,4401h        ; AL = subfunction 01H, set device
                                ; information word
                                ; (BX still contains handle)
        int     21h             ; transfer to MS-DOS
```

Example: Use IOCTL Subfunction 06H to test whether a character is ready for input on the first serial port. The function returns AL = 0FFH if a character is ready and AL = 00H if not.

```
        mov     ax,4406H        ; AH = function 44H, IOCTL
                                ; AL = subfunction 06H, get
                                ; input status
        mov     bx,3            ; BX = handle for standard aux
        int     21h             ; transfer to MS-DOS
        or      al,al           ; test status of AUX driver
        jnz     ready           ; jump if input character ready
                                ; else no character is waiting
```

Jim Kyle
Chip Rabinowitz

Article 6
Interrupt-Driven Communications

In the earliest days of personal-computer communications, when speeds were no faster than 300 bits per second, primitive programs that moved characters to and from the remote system were adequate. The PC had time between characters to determine what it ought to do next and could spend that time keeping track of the status of the remote system.

Modern data-transfer rates, however, are four to eight times faster and leave little or no time to spare between characters. At 1200 bits per second, as many as three characters can be lost in the time required to scroll the display up one line. At such speeds, a technique to permit characters to be received and simultaneously displayed becomes necessary.

Mainframe systems have long made use of hardware interrupts to coordinate such activities. The processor goes about its normal activity; when a peripheral device needs attention, it sends an interrupt request to the processor. The processor interrupts its activity, services the request, and then goes back to what it was doing. Because the response is driven by the request, this type of processing is known as interrupt-driven. It gives the effect of doing two things at the same time without requiring two separate processors.

Successful telecommunication with PCs at modern data rates demands an interrupt-driven routine for data reception. This article discusses in detail the techniques for interrupt-driven communications and culminates in two sample program packages.

The article begins by establishing the purpose of communications programs and then discusses the capability of the simple functions provided by MS-DOS to achieve this goal. To see what must be done to supplement MS-DOS functions, the hardware (both the modem and the serial port) is examined. This leads to a discussion of the method MS-DOS has provided since version 2.0 for solving the problems of special hardware interfacing: the installable device driver.

With the background established, alternate paths to interrupt-driven communications are discussed — one following recommended MS-DOS techniques, the other following standard industry practice — and programs are developed for each.

Throughout this article, the discussion is restricted to the architecture and BIOS of the IBM PC family. MS-DOS systems not totally compatible with this architecture may require substantially different approaches at the detailed level, but the same general principles apply.

Purpose of Communications Programs

The primary purpose of any communications program is communicating — that is, transmitting information entered as keystrokes (or bytes read from a file) in a form suitable for transmission to a remote computer via phone lines and, conversely, converting information received from the remote computer into a display on the video screen (or data in a file).

Some years ago, the most abstract form of all communications programs was dubbed a modem engine, by analogy to Babbage's analytical engine or the inference-engine model used in artificial-intelligence development. The functions of the modem engine are common to all kinds of communications programs, from the simplest to the most complex, and can be described in a type of pseudo-C as follows:

```
The Modem Engine Pseudocode

     DO {  IF (input character is available)
              send_it_to_remote;
           IF (remote character is available)
              use_it_locally;
        } UNTIL (told_to_stop);
```

The essence of this modem-engine code is that the absence of an input character, or of a character from the remote computer, does not hang the loop in a wait state. Rather, the engine continues to cycle: If it finds work to do, it does it; if not, the engine keeps looking.

Of course, at times it is desirable to halt the continuous action of the modem engine. For example, when receiving a long message, it is nice to be able to pause and read the message before the lines scroll into oblivion. On the other hand, taking too long to study the screen means that incoming characters are lost. The answer is a technique called flow control, in which a special control character is sent to shut down transmission and some other character is later sent to start it up again.

Several conventions for flow control exist. One of the most widespread is known as XON/XOFF, from the old Teletype-33 keycap legends for the two control codes involved. In the original use, XOFF halted the paper tape reader and XON started it going again. In mid-1967, the General Electric Company began using these signals in its time-sharing computer services to control the flow of data, and the practice rapidly spread throughout the industry.

The sample program named ENGINE, shown later in this article, is an almost literal implementation of the modem-engine approach. This sample represents one extreme of simplicity in communications programs. The other sample program, CTERM.C, is much more complex, but the modem engine is still at its heart.

Using Simple MS-DOS Functions

Because MS-DOS provides, among its standard service functions, the capability of sending output to or reading input from the device named AUX (which defaults to COM1, the first serial port on most machines), a first attempt at implementing the modem engine using MS-DOS functions might look something like the following incomplete fragment of Microsoft Macro Assembler (MASM) code:

```
;Incomplete (and Unworkable) Implementation

LOOP:   MOV     AH,08h          ; read keyboard, no echo
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,04h          ; send to AUX device
        INT     21h
        MOV     AH,03h          ; read from AUX device
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,02h          ; send to screen
        INT     21h
        JMP     LOOP            ; keep doing it
```

The problem with this code is that it violates the keep-looking principle both at the keyboard and at the AUX port: Interrupt 21H Function 08H does not return until a keyboard character is available, so no data from the AUX port can be read until a key is pressed locally. Similarly, Function 03H waits for a character to become available from AUX, so no more keys can be recognized locally until the remote system sends a character. If nothing is received, the loop waits forever.

To overcome the problem at the keyboard end, Function 0BH can be used to determine if a key has been pressed before an attempt is made to read one, as shown in the following modification of the fragment:

```
;Improved, (but Still Unworkable) Implementation

LOOP:   MOV     AH,0Bh          ; test keyboard for char
        INT     21h
        OR      AL,AL           ; test for zero
        JZ      RMT             ; no char avail, skip
        MOV     AH,08h          ; have char, read it in
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,04h          ; send to AUX device
        INT     21h
RMT:
        MOV     AH,03h          ; read from AUX device
        INT     21h
        MOV     DL,AL           ; set up to send
        MOV     AH,02h          ; send to screen
        INT     21h
        JMP     LOOP            ; keep doing it
```

This code permits any input from AUX to be received without waiting for a local key to be pressed, but if AUX is slow about providing input, the program waits indefinitely before checking the keyboard again. Thus, the problem is only partially solved.

MS-DOS, however, simply does not provide any direct method of making the required tests for AUX or, for that matter, any of the serial port devices. That is why communications programs must be treated differently from most other types of programs under MS-DOS and why such programs must be intimately involved with machine details despite all accepted principles of portable program design.

The Hardware Involved

Personal-computer communications require at least two distinct pieces of hardware (separate devices, even though they are often combined on a single board). These hardware items are the serial port, which converts data from the computer's internal bus into a bit stream for transmission over a single external line, and the modem, which converts the bit stream into a form suitable for telephone-line (or, sometimes, radio) transmission.

The modem

The modem (a word coined from MOdulator-DEModulator) is a device that converts a stream of bits, represented as sequential changes of voltage level, into audio frequency signals suitable for transmission over voice-grade telephone circuits (modulation) and converts these signals back into a stream of bits that duplicates the original input (demodulation).

Specific characteristics of the audio signals involved were established by AT&T when that company monopolized the modem industry, and those characteristics then evolved into de facto standards when the monopoly vanished. They take several forms, depending on the data rate in use; these forms are normally identified by the original Bell specification number, such as 103 (for 600 bps and below) or 212A (for the 1200 bps standard).

The data rate is measured in bits per second (bps), often mistermed baud or even "baud per second." A baud measures the number of signals per second; as with knot (nautical miles per hour), the time reference is built in. If one signal change marks one bit, as is true for the Bell 103 standard, then baud and bps have equal values. However, they are not equivalent for more complex signals. For example, the Bell 212A diphase standard for 1200 bps uses two tone streams, each operating at 600 baud, to transmit data at 1200 bits per second.

For accuracy, this article uses bps, rather than baud, except where widespread industry misuse of baud has become standardized (as in "baud rate generator").

Originally, the modem itself was a box connected to the computer's serial port via a cable. Characteristics of this cable, its connectors, and its signals were standardized in the 1960s by the Electronic Industries Association (EIA), in Standard RS232C. Like the Bell standards for modems, RS232C has survived almost unchanged. Its characteristics are listed in Table 6-1.

Table 6-1. RS232C Signals.

DB25 Pin	232	Name	Description
1			Safety Ground
2	BA	TXD	Transmit Data
3	BB	RXD	Receive Data
4	CA	RTS	Request To Send
5	CB	CTS	Clear To Send
6	CC	DSR	Data Set Ready
7	AB	GND	Signal Ground
8	CF	DCD	Data Carrier Detected
20	CD	DTR	Data Terminal Ready
22	CE	RI	Ring Indicator

With the increasing popularity of personal computers, internal modems that plug into the PC's motherboard and combine the modem and a serial port became available.

The first such units were manufactured by Hayes Corporation, and like Bell and the EIA, they created a standard. Functionally, the internal modem is identical to the combination of a serial port, a connecting cable, and an external modem.

The serial port

Each serial port of a standard IBM PC connects the rest of the system to a type INS8250 Universal Asynchronous Receiver Transmitter (UART) integrated circuit (IC) chip developed by National Semiconductor Corporation. This chip, along with associated circuits in the port,

1. Converts data supplied via the system data bus into a sequence of voltage levels on the single TXD output line that represent binary digits.
2. Converts data received as a sequence of binary levels on the single RXD input line into bytes for the data bus.
3. Controls the modem's actions through the DTR and RTS output lines.
4. Provides status information to the processor; this information comes from the modem, via the DSR, DCD, CTS, and RI input lines, and from within the UART itself, which signals data available, data needed, or error detected.

The word *asynchronous* in the name of the IC comes from the Bell specifications. When computer data is transmitted, each bit's relationship to its neighbors must be preserved; this can be done in either of two ways. The most obvious method is to keep the bit stream strictly synchronized with a clock signal of known frequency and count the cycles to identify the bits. Such a transmission is known as synchronous, often abbreviated to synch or sometimes bisync for binary synchronous. The second method, first used with mechanical teleprinters, marks the start of each bit group with a defined start bit and the end with one or more defined stop bits, and it defines a duration for each bit time. Detection of a start bit marks the beginning of a received group; the signal is then sampled at each bit time until the stop bit is encountered. This method is known as asynchronous (or just asynch) and is the one used by the standard IBM PC.

The start bit is, by definition, exactly the same as that used to indicate binary zero, and the stop bit is the same as that indicating binary one. A zero signal is often called SPACE, and a one signal is called MARK, from terms used in the teleprinter industry.

During transmission, the least significant bit of the data is sent first, after the start bit. A parity bit, if used, appears as the most significant bit in the data group, before the stop bit or bits; it cannot be distinguished from a databit except by its position. Once the first stop bit is sent, the line remains in MARK (sometimes called idling) condition until a new start bit indicates the beginning of another group.

In most PC uses, the serial port transfers one 8-bit byte at a time, and the term *word* specifies a 16-bit quantity. In the UART world, however, a word is the unit of information sent by the chip in each chunk. The word length is part of the control information set into the chip during setup operations and can be 5, 6, 7, or 8 bits. This discussion follows UART conventions and refers to words, rather than to bytes.

One special type of signal, not often used in PC-to-PC communications but sometimes necessary in communicating with mainframe systems, is a BREAK. The BREAK is an all-SPACE condition that extends for more than one word time, including the stop-bit time. (Many systems require the BREAK to last at least 150 milliseconds regardless of data rate.) Because it cannot be generated by any normal data character transmission, the BREAK is used to interrupt, or break into, normal operation. The IBM PC's 8250 UART can generate the BREAK signal, but its duration must be determined by a program, rather than by the chip.

The 8250 UART architecture

The 8250 UART contains four major functional areas: receiver, transmitter, control circuits, and status circuits. Because these areas are closely related, some terms used in the following descriptions are, of necessity, forward references to subsequent paragraphs.

The major parts of the receiver are a shift register and a data register called the Received Data Register. The shift register assembles sequentially received data into word-parallel form by shifting the level of the RXD line into its front end at each bit time and, at the same time, shifting previous bits over. When the shift register is full, all bits in it are moved over to the data register, the shift register is cleared to all zeros, and the bit in the status circuits that indicates data ready is set. If an error is detected during receipt of that word, other bits in the status circuits are also set.

Similarly, the major parts of the transmitter are a holding register called the Transmit Holding Register and a shift register. Each word to be transmitted is transferred from the data bus to the holding register. If the holding register is not empty when this is done, the previous contents are lost. The transmitter's shift register converts word-parallel data into bit-serial form for transmission by shifting the most significant bit out to the TXD line once each bit time, at the same time shifting lower bits over and shifting in an idling bit at the low end of the register. When the last databit has been shifted out, any data in the holding register is moved to the shift register, the holding register is filled with idling bits in case no more data is forthcoming, and the bit in the status circuits that indicates the Transmit Holding Register is empty is set to indicate that another word can be transferred. The parity bit, if any, and stop bits are added to the transmitted stream after the last databit of each word is shifted out.

The control circuits establish three communications features: first, line control values, such as word length, whether or not (and how) parity is checked, and the number of stop bits; second, modem control values, such as the state of the DTR and RTS output lines; and third, the rate at which data is sent and received. These control values are established by two 8-bit registers and one 16-bit register, which are addressed as four 8-bit registers. They are the Line Control Register (LCR), the Modem Control Register (MCR), and the 16-bit BRG Divisor Latch, addressed as Baud0 and Baud1.

The BRG Divisor Latch sets the data rate by defining the bit time produced by the Programmable Baud Rate Generator (PBRG), a major part of the control circuits. The PBRG can provide any data speed from a few bits per second to 38400 bps; in the BIOS of the IBM PC, PC/XT, and PC/AT, though, only the range 110 through 9600 bps is supported. How the LCR and the MCR establish their control values, how the PBRG is programmed, and how interrupts are enabled are discussed later.

The fourth major area in the 8250 UART, the status circuits, records (in a pair of status registers) the conditions in the receive and transmit circuits, any errors that are detected, and any change in state of the RS232C input lines from the modem. When any status register's content changes, an interrupt request, if enabled, is generated to notify the rest of the PC system. This approach lets the PC attend to other matters without having to continually monitor the status of the serial port, yet it assures immediate action when something does occur.

The 8250 programming interface

Not all the registers mentioned in the preceding section are accessible to programmers. The shift registers, for example, can be read from or written to only by the 8250's internal circuits. There are 10 registers available to the programmer, and they are accessed by only seven distinct addresses (shown in Table 6-2). The Received Data Register and the Transmit Holding Register share a single address (a read gets the received data; a write goes to the holding register). In addition, both this address and that of the Interrupt Enable Register (IER) are shared with the PBRG Divisor Latch. A bit in the Line Control Register called the Divisor Latch Access Bit (DLAB) determines which register is addressed at any specific time.

In the IBM PC, the seven addresses used by the 8250 are selected by the low 3 bits of the port number (the higher bits select the specific port). Thus, each serial port occupies eight positions in the address space. However, only the lowest address used — the one in which the low 3 bits are all 0 — need be remembered in order to access all eight addresses.

Because of this, any serial port in the PC is referred to by an address that, in hexadecimal notation, ends with either 0 or 8: The COM1 port normally uses address 03F8H, and COM2 uses 02F8H. This lowest port address is usually called the base port address, and each addressable register is then referenced as an offset from this base value, as shown in Table 6-2.

Table 6-2. 8250 Port Offsets from Base Address.

Offset	Name	Description
If DLAB bit in LCR = 0:		
00H	DATA	Received Data Register if read from, Transmit Holding Register if written to
01H	IER	Interrupt Enable Register
If DLAB bit in LCR = 1:		
00H	Baud0	BRG Divisor Latch, low byte
01H	Baud1	BRG Divisor Latch, high byte
Not affected by DLAB bit:		
02H	IID	Interrupt Identifier Register
03H	LCR	Line Control Register
04H	MCR	Modem Control Register
05H	LSR	Line Status Register
06H	MSR	Modem Status Register

The control circuits

The control circuits of the 8250 include the Programmable Baud Rate Generator (PBRG), the Line Control Register (LCR), the Modem Control Register (MCR), and the Interrupt Enable Register (IER).

The PBRG establishes the bit time used for both transmitting and receiving data by dividing an external clock signal. To select a desired bit rate, the appropriate divisor is loaded into the PBRG's 16-bit Divisor Latch by setting the Divisor Latch Access Bit (DLAB) in the Line Control Register to 1 (which changes the functions of addresses 0 and 1) and then writing the divisor into Baud0 and Baud1. After the bit rate is selected, DLAB is changed back to 0, to permit normal operation of the DATA registers and the IER.

With the 1.8432 MHz external UART clock frequency used in standard IBM systems, divisor values (in decimal notation) for bit rates between 45.5 and 38400 bps are listed in Table 6-3. These speeds are established by a crystal contained in the serial port (or internal modem) and are totally unrelated to the speed of the processor's clock.

Table 6-3. Bit Rate Divisor Table for 8250/IBM.

BPS	Divisor
45.5	2532
50	2304
75	1536
110	1047
134.5	857
150	768
300	384
600	192
1200	96
1800	64
2000	58
2400	48
4800	24
9600	12
19200	6
38400	3

The remaining control circuits are the Line Control Register, the Modem Control Register, and the Interrupt Enable Register. Bits in the LCR control the assignment of offsets 0 and 1, transmission of the BREAK signal, parity generation, the number of stop bits, and the word length sent and received, as shown in Table 6-4.

Table 6-4. 8250 Line Control Register Bit Values.

Bit	Name	Binary	Meaning
Address Control:			
7	DLAB	0xxxxxxx	Offset 0 refers to DATA; offset 1 refers to IER
		1xxxxxxx	Offsets 0 and 1 refer to BRG Divisor Latch
BREAK Control:			
6	SETBRK	x0xxxxxx	Normal UART operation
		x1xxxxxx	Send BREAK signal

(more)

Table 6-4. *Continued.*

Bit	Name	Binary	Meaning
Parity Checking:			
5,4,3	GENPAR	xxxx0xxx	No parity bit
		xx001xxx	Parity bit is ODD
		xx011xxx	Parity bit is EVEN
		xx101xxx	Parity bit is 1
		xx111xxx	Parity bit is 0
Stop Bits:			
2	XSTOP	xxxxx0xx	Only 1 stop bit
		xxxxx1xx	2 stop bits (1.5 if WL = 5)
Word Length:			
1,0	WD5	xxxxxx00	Word length = 5
	WD6	xxxxxx01	Word length = 6
	WD7	xxxxxx10	Word length = 7
	WD8	xxxxxx11	Word length = 8

Two bits in the MCR (Table 6-5) control output lines DTR and RTS; two other MCR bits (OUT1 and OUT2) are left free by the UART to be assigned by the user; a fifth bit (TEST) puts the UART into a self-test mode of operation. The upper 3 bits have no effect on the UART. The MCR can be both read from and written to.

Both of the user-assignable bits are defined in the IBM PC. OUT1 is used by Hayes internal modems to cause a power-on reset of their circuits; OUT2 controls the passage of UART-generated interrupt request signals to the rest of the PC. Unless OUT2 is set to 1, interrupt signals from the UART cannot reach the rest of the PC, even though all other controls are properly set. This feature is documented, but obscurely, in the IBM *Technical Reference* manuals and the asynchronous-adapter schematic; it is easy to overlook when writing an interrupt-driven program for these machines.

Table 6-5. 8250 Modem Control Register Bit Values.

Name	Binary	Description
TEST	xxx1xxxx	Turns on UART self-test configuration.
OUT2	xxxx1xxx	Controls 8250 interrupt signals (User2 Output).
OUT1	xxxxx1xx	Resets Hayes 1200b internal modem (User1 Output).
RTS	xxxxxx1x	Sets RTS output to RS232C connector.
DTR	xxxxxxx1	Sets DTR output to RS232C connector.

The 8250 can generate any or all of four classes of interrupts, each individually enabled or disabled by setting the appropriate control bit in the Interrupt Enable Register (Table 6-6). Thus, setting the IER to 00H disables all the UART interrupts within the 8250 without regard to any other settings, such as OUT2, system interrupt masking, or the CLI/STI commands. The IER can be both read from and written to. Only the low 4 bits have any effect on the UART.

Table 6-6. 8250 Interrupt Enable Register Constants.

Binary	Action
xxxx1xxx	Enable Modem Status Interrupt.
xxxxx1xx	Enable Line Status Interrupt.
xxxxxx1x	Enable Transmit Register Interrupt.
xxxxxxx1	Enable Received Data Ready Interrupt.

The status circuits

The status circuits of the 8250 include the Line Status Register (LSR), the Modem Status Register (MSR), the Interrupt Identifier (IID) Register, and the interrupt-request generation system.

The 8250 includes circuitry that detects a received BREAK signal and also detects three classes of data-reception errors. Separate bits in the LSR (Table 6-7) are set to indicate that a BREAK has been received and to indicate any of the following: a parity error (if lateral parity is in use), a framing error (incoming bit = 0 at stop-bit time), or an overrun error (word not yet read from receive buffer by the time the next word must be moved into it).

The remaining bits of the LSR indicate the status of the Transmit Shift Register, the Transmit Holding Register, and the Received Data Register; the most significant bit of the LSR is not used and is always 0. The LSR is a read-only register; writing to it has no effect.

Table 6-7. 8250 Line Status Register Bit Values.

Bit	Binary	Meaning
7	0xxxxxxx	Always zero
6	x1xxxxxx	Transmit Shift Register empty
5	xx1xxxxx	Transmit Holding Register empty
4	xxx1xxxx	BREAK received
3	xxxx1xxx	Framing error
2	xxxxx1xx	Parity error
1	xxxxxx1x	Overrun error
0	xxxxxxx1	Received data ready

The MSR (Table 6-8) monitors the four RS232C lines that report modem status. The upper 4 bits of this register indicate the voltage level of the associated RS232C line; the lower 4 bits indicate that the voltage level has changed since the register was last read.

Table 6-8. 8250 Modem Status Register Bit Values.

Bit	Binary	Meaning
7	1xxxxxxx	Data Carrier Detected (DCD) level
6	x1xxxxxx	Ring Indicator (RI) level
5	xx1xxxxx	Data Set Ready (DSR) level
4	xxx1xxxx	Clear To Send (CTS) level
3	xxxx1xxx	DCD change
2	xxxxx1xx	RI change
1	xxxxxx1x	DSR change
0	xxxxxxx1	CTS change

As mentioned previously, four types of interrupts are generated. The four types are identified by flag values in the IID Register (Table 6-9). These flags are set as follows:

- Change of any bit value in the MSR sets the modem status flag.
- Setting of the BREAK Received bit or any of the three error bits in the LSR sets the line status flag.
- Setting of the Transmit Holding Register Empty bit in the LSR sets the transmit flag.
- Setting of the Received Data Ready bit in the LSR sets the receive flag.

The IID register indicates the interrupt type, even though the IER may be disabling that type of interrupt from generating any request. The IID is a read-only register; attempts to write to it have no effect.

Table 6-9. 8250 Interrupt Identification and Causes.

IID content	Meaning
xxxxxxx1B	No interrupt active
xxxxx000B	Modem Status Interrupt; bit changed in MSR
xxxxx010B	Transmit Register Interrupt; Transmit Holding Register empty, bit set in LSR
xxxxx100B	Received Data Ready Interrupt; Data Register full, bit set in LSR
xxxxx110B	Line Status Interrupt; BREAK or error bit set in LSR

As shown in Table 6-9, an all-zero value (which in most of the other registers is a totally disabling condition) means that a Modem Status Interrupt condition has not yet been serviced. A modem need not be connected, however, for a Modem Status Interrupt condition to occur; all that is required is for one of the RS232C non-data input lines to change state, thus changing the MSR.

Whenever a flag is set in the IID, the UART interrupt-request generator will, if enabled by the UART programming, generate an interrupt request to the processor. Two or more interrupts can be active at the same time; if so, more than one flag in the IID register is set.

The IID flag for each interrupt type (and the LSR or MSR bits associated with it) clears when the corresponding register is read (or, in one case, written to). For example, reading the content of the MSR clears the modem status flag; writing a byte to the DATA register clears the transmit flag; reading the DATA register clears the receive flag; reading the LSR clears the line status flag. The LSR or MSR bit does not clear until it has been read; the IID flag clears with the LSR or MSR bit.

Programming the UART

Each time power is applied, any serial-interface device must be programmed before it is used. This programming can be done by the computer's bootstrap sequence or as a part of the port initialization routines performed when a port driver is installed. Often, both techniques are used: The bootstrap provides default conditions, and these can be modified during initialization to meet the needs of each port driver used in a session.

When the 8250 chip is programmed, the BRG Divisor Latch should be set for the proper baud rate, the LCR and MCR should be loaded, the IER should be set, and all internal interrupt requests and the receive buffer should be cleared. The sequence in which these are done is not especially critical, but any pending interrupt requests should be cleared before they are permitted to pass on to the rest of the PC.

The following sample code performs these startup actions, setting up the chip in device COM1 (at port 03F8H) to operate at 1200 bps with a word length of 8 bits, no parity checking, and all UART interrupts enabled. (In practical code, all values for addresses and operating conditions would not be built in; these values are included in the example to clarify what is being done at each step.)

```
MOV     DX,03FBh        ; base port COM1 (03F8) + LCR (3)
MOV     AL,080h         ; enable Divisor Latch
OUT     DX,AL
MOV     DX,03F8h        ; set for Baud0
MOV     AX,96           ; set divisor to 1200 bps
OUT     DX,AL
INC     DX              ; to offset 1 for Baud1
MOV     AL,AH           ; high byte of divisor
OUT     DX,AL
MOV     DX,03FBh        ; back to the LCR offset
MOV     AL,03           ; DLAB = 0, Parity = N, WL = 8
OUT     DX,AL
MOV     DX,03F9h        ; offset 1 for IER
MOV     AL,0Fh          ; enable all ints in 8250
OUT     DX,AL
MOV     DX,03FCh        ; COM1 + MCR (4)
MOV     AL,0Bh          ; OUT2 + RTS + DTR bits
OUT     DX,AL
```

(more)

```
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 COM*x* so that every program can have all features. However, this approach is not compatible with existing terminal programs because it has never been a part of MS-DOS.

Comparison of the two methods

The traditional approach has several advantages, the most obvious being that the driver code can be fully tailored to the needs of the program. Because only one program will ever use the driver, no general cases need be considered.

However, if a user wants to keep communications capability available in a terminate-and-stay-resident (TSR) module for background use and also wants a different type of communications program running in the foreground (not, of course, while the background task is using the port), the background program and the foreground job must each have its own separate driver code. And, because such code usually includes buffer areas, the duplicated drivers represent wasted resources.

A single communications device driver that is installed when the system powers up and that remains active until shutdown avoids wasting resources by allowing both the background and foreground tasks to share the driver code. Until such drivers are common, however, it is unlikely that commercial software will be able to make use of them. In addition, such a driver must either provide totally general capabilities or it must include control interfaces so each user program can dynamically alter the driver to suit its needs.

At this time, the use of a single driver is an interesting exercise rather than a practical application, although a possible exception is a dedicated system in which all software is either custom designed or specially modified. In such a system, the generalized driver can provide significant improvement in the efficiency of resource allocation.

A Device-Driver Program Package

Despite the limitations mentioned in the preceding section, the first of the two complete packages in this article uses the concept of a separate device driver. The driver handles all hardware-dependent interfacing and thus permits extreme simplicity in all other modules of the package. This approach is presented first because it is especially well suited for introducing the concepts of communications programs. However, the package is not merely a tutorial device: It includes some features that are not available in most commercial programs.

The package itself consists of three separate programs. First is the device driver, which becomes a part of MS-DOS via the CONFIG.SYS file. Second is the modem engine, which is the actual terminal program. (A functionally similar component forms the heart of every communications program, whether it is written in assembly language or a high-level language and regardless of the machine or operating system in use.) Third is a separately executed support program that permits changing such driver characteristics as word length, parity, and baud rate.

In most programs that use the traditional approach, the driver and the support program are combined with the modem engine in a single unit and the resulting mass of detail obscures the essential simplicity of each part. Here, the parts are presented as separate modules to emphasize that simplicity.

The device driver: COMDVR.ASM

The device driver is written to augment the default COM1 and COM2 devices with other devices named ASY1 and ASY2 that use the same physical hardware but are logically separate. The driver (COMDVR.ASM) is implemented in MASM and is shown in the listing in Figure 6-1. Although the driver is written basically as a skeleton, it is designed to permit extensive expansion and can be used as a general-purpose sample of device-driver source code.

The code

```
Title   COMDVR  Driver for IBM COM Ports
;       Jim Kyle, 1987
;           Based on ideas from many sources......
;               including Mike Higgins, CLM March 1985;
;               public-domain INTBIOS program from BBS's;
;               COMBIOS.COM from CIS Programmers' SIG; and
;               ADVANCED MS-DOS by Ray Duncan.
Subttl  MS-DOS Driver Definitions

        Comment *       This comments out the Dbg macro.....
Dbg     Macro   Ltr1,Ltr2,Ltr3  ; used only to debug driver...
        Local   Xxx
        Push    Es              ; save all regs used
```

Figure 6-1. COMDVR.ASM. *(more)*

```
        Push    Di
        Push    Ax
        Les     Di,Cs:Dbgptr    ; get pointer to CRT
        Mov     Ax,Es:[di]
        Mov     Al,Ltr1         ; move in letters
        Stosw
        Mov     Al,Ltr2
        Stosw
        Mov     Al,Ltr3
        Stosw
        Cmp     Di,1600         ; top 10 lines only
        Jb      Xxx
        Xor     Di,Di
Xxx:    Mov     Word Ptr Cs:Dbgptr,Di
        Pop     Ax
        Pop     Di
        Pop     Es
        Endm
        *                       ; asterisk ends commented-out region
;
;               Device Type Codes
DevChr  Equ     8000h   ; this is a character device
DevBlk  Equ     0000h   ; this is a block (disk) device
DevIoc  Equ     4000h   ; this device accepts IOCTL requests
DevNon  Equ     2000h   ; non-IBM disk driver (block only)
DevOTB  Equ     2000h   ; MS-DOS 3.x out until busy supported (char)
DevOCR  Equ     0800h   ; MS-DOS 3.x open/close/rm supported
DevX32  Equ     0040h   ; MS-DOS 3.2 functions supported
DevSpc  Equ     0010h   ; accepts special interrupt 29H
DevClk  Equ     0008h   ; this is the CLOCK device
DevNul  Equ     0004h   ; this is the NUL device
DevSto  Equ     0002h   ; this is standard output
DevSti  Equ     0001h   ; this is standard input
;
;               Error Status BITS
StsErr  Equ     8000h   ; general error
StsBsy  Equ     0200h   ; device busy
StsDne  Equ     0100h   ; request completed
;
;               Error Reason values for lower-order bits
ErrWp   Equ     0       ; write protect error
ErrUu   Equ     1       ; unknown unit
ErrDnr  Equ     2       ; drive not ready
ErrUc   Equ     3       ; unknown command
ErrCrc  Equ     4       ; cyclical redundancy check error
ErrBsl  Equ     5       ; bad drive request structure length
ErrSl   Equ     6       ; seek error
ErrUm   Equ     7       ; unknown media
ErrSnf  Equ     8       ; sector not found
ErrPop  Equ     9       ; printer out of paper
ErrWf   Equ     10      ; write fault
```

Figure 6-1. Continued. *(more)*

```
ErrRf   Equ     11      ; read fault
ErrGf   Equ     12      ; general failure
;
;       Structure of an I/O request packet header.
;
Pack    Struc
Len     Db      ?       ; length of record
Prtno   Db      ?       ; unit code
Code    Db      ?       ; command code
Stat    Dw      ?       ; return status
Dosq    Dd      ?       ; (unused MS-DOS queue link pointer)
Devq    Dd      ?       ; (unused driver queue link pointer)
Media   Db      ?       ; media code on read/write
Xfer    Dw      ?       ; xfer address offset
Xseg    Dw      ?       ; xfer address segment
Count   Dw      ?       ; transfer byte count
Sector  Dw      ?       ; starting sector value (block only)
Pack    Ends

Subttl  IBM-PC Hardware Driver Definitions
page
;
;               8259 data
PIC_b   Equ     020h    ; port for EOI
PIC_e   Equ     021h    ; port for Int enabling
EOI     Equ     020h    ; EOI control word
;
;               8250 port offsets
RxBuf   Equ     0F8h    ; base address
Baud1   Equ     RxBuf+1 ; baud divisor high byte
IntEn   Equ     RxBuf+1 ; interrupt enable register
IntId   Equ     RxBuf+2 ; interrupt identification register
Lctrl   Equ     RxBuf+3 ; line control register
Mctrl   Equ     RxBuf+4 ; modem control register
Lstat   Equ     RxBuf+5 ; line status register
Mstat   Equ     RxBuf+6 ; modem status register
;
;               8250 LCR constants
Dlab    Equ     10000000b ; divisor latch access bit
SetBrk  Equ     01000000b ; send break control bit
StkPar  Equ     00100000b ; stick parity control bit
EvnPar  Equ     00010000b ; even parity bit
GenPar  Equ     00001000b ; generate parity bit
Xstop   Equ     00000100b ; extra stop bit
Wd8     Equ     00000011b ; word length = 8
Wd7     Equ     00000010b ; word length = 7
Wd6     Equ     00000001b ; word length = 6
;
;               8250 LSR constants
xsre    Equ     01000000b ; xmt SR empty
xhre    Equ     00100000b ; xmt HR empty
```

Figure 6-1. Continued. *(more)*

```
BrkRcv  Equ     00010000b ; break received
FrmErr  Equ     00001000b ; framing error
ParErr  Equ     00000100b ; parity error
OveRun  Equ     00000010b ; overrun error
rdta    Equ     00000001b ; received data ready
AnyErr  Equ     BrkRcv+FrmErr+ParErr+OveRun
;
;               8250 MCR constants
LpBk    Equ     00010000b ; UART out loops to in (test)
Usr2    Equ     00001000b ; Gates 8250 interrupts
Usr1    Equ     00000100b ; aux user1 output
SetRTS  Equ     00000010b ; sets RTS output
SetDTR  Equ     00000001b ; sets DTR output
;
;               8250 MSR constants
CDlvl   Equ     10000000b ; carrier detect level
RIlvl   Equ     01000000b ; ring indicator level
DSRlvl  Equ     00100000b ; DSR level
CTSlvl  Equ     00010000b ; CTS level
CDchg   Equ     00001000b ; Carrier Detect change
RIchg   Equ     00000100b ; Ring Indicator change
DSRchg  Equ     00000010b ; DSR change
CTSchg  Equ     00000001b ; CTS change
;
;               8250 IER constants
S_Int   Equ     00001000b ; enable status interrupt
E_Int   Equ     00000100b ; enable error interrupt
X_Int   Equ     00000010b ; enable transmit interrupt
R_Int   Equ     00000001b ; enable receive interrupt
Allint  Equ     00001111b ; enable all interrupts

Subttl  Definitions for THIS Driver
page
;
;               Bit definitions for the output status byte
;                      ( this driver only )
LinIdl  Equ     0ffh    ; if all bits off, xmitter is idle
LinXof  Equ     1       ; output is suspended by XOFF
LinDSR  Equ     2       ; output is suspended until DSR comes on again
LinCTS  Equ     4       ; output is suspended until CTS comes on again
;
;               Bit definitions for the input status byte
;                       ( this driver only )
BadInp  Equ     1       ; input line errors have been detected
LostDt  Equ     2       ; receiver buffer overflowed, data lost
OffLin  Equ     4       ; device is off line now
;
;               Bit definitions for the special characteristics words
;                       ( this driver only )
;               InSpec controls how input from the UART is treated
;
```

Figure 6-1. Continued. *(more)*

```
InEpc   Equ     0001h   ; errors translate to codes with parity bit on
;
;               OutSpec controls how output to the UART is treated
;
OutDSR  Equ     0001h   ; DSR is used to throttle output data
OutCTS  Equ     0002h   ; CTS is used to throttle output data
OutXon  Equ     0004h   ; XON/XOFF is used to throttle output data
OutCdf  Equ     0010h   ; carrier detect is off-line signal
OutDrf  Equ     0020h   ; DSR is off-line signal
;
Unit    Struc           ; each unit has a structure defining its state:
Port    Dw      ?       ; I/O port address
Vect    Dw      ?       ; interrupt vector offset (NOT interrupt number!)
Isradr  Dw      ?       ; offset to interrupt service routine
OtStat  Db      Wd8     ; default LCR bit settings during INIT,
                        ; output status bits after
InStat  Db      Usr2+SetRTS+SetDTR   ; MCR bit settings during INIT,
                        ; input status bits after
InSpec  Dw      InEpc   ; special mode bits for INPUT
OutSpec Dw      OutXon  ; special mode bits for OUTPUT
Baud    Dw      96      ; current baud rate divisor value (1200 b)
Ifirst  Dw      0       ; offset of first character in input buffer
Iavail  Dw      0       ; offset of next available byte
Ibuf    Dw      ?       ; pointer to input buffer
Ofirst  Dw      0       ; offset of first character in output buffer
Oavail  Dw      0       ; offset of next avail byte in output buffer
Obuf    Dw      ?       ; pointer to output buffer
Unit    Ends

;
;       Beginning of driver code and data
;
Driver  Segment
        Assume  Cs:driver, ds:driver, es:driver
        Org     0               ; drivers start at 0

        Dw      Async2,-1       ; pointer to next device
        Dw      DevChr + DevIoc ; character device with IOCTL
        Dw      Strtegy         ; offset of Strategy routine
        Dw      Request1        ; offset of interrupt entry point 1
        Db      'ASY1    '      ; device 1 name
Async2:
        Dw      -1,-1           ; pointer to next device: MS-DOS fills in
        Dw      DevChr + DevIoc ; character device with IOCTL
        Dw      Strtegy         ; offset of Strategy routine
        Dw      Request2        ; offset of interrupt entry point 2
        Db      'ASY2    '      ; device 2 name

;dbgptr Dd      0b0000000h
;
;       Following is the storage area for the request packet pointer
```

Figure 6-1. Continued. *(more)*

```
;
PackHd  Dd      0
;
;         baud rate conversion table
Asy_baudt Dw            50,2304         ; first value is desired baud rate
          Dw            75,1536         ; second is divisor register value
          Dw           110,1047
          Dw           134, 857
          Dw           150, 786
          Dw           300, 384
          Dw           600, 192
          Dw          1200,  96
          Dw          1800,  64
          Dw          2000,  58
          Dw          2400,  48
          Dw          3600,  32
          Dw          4800,  24
          Dw          7200,  16
          Dw          9600,  12

; table of structures
;       ASY1 defaults to the COM1 port, INT 0CH vector, XON,
;       no parity, 8 databits, 1 stop bit, and 1200 baud
Asy_tab1:
        Unit    <3f8h,30h,asy1isr,,,,,,,,in1buf,,,out1buf>

;       ASY2 defaults to the COM2 port, INT 0BH vector, XON,
;       no parity, 8 databits, 1 stop bit, and 1200 baud
Asy_tab2:
        Unit    <2f8h,2ch,asy2isr,,,,,,,,in2buf,,,out2buf>

Bufsiz  Equ     256        ; input buffer size
Bufmsk  =       Bufsiz-1   ; mask for calculating offsets modulo bufsiz
In1buf  Db      Bufsiz DUP (?)
Out1buf Db      Bufsiz DUP (?)
In2buf  Db      Bufsiz DUP (?)
Out2buf Db      Bufsiz DUP (?)
;
;               Following is a table of offsets to all the driver functions

Asy_funcs:
        Dw      Init            ;  0 initialize driver
        Dw      Mchek           ;  1 media check (block only)
        Dw      BldBPB          ;  2 build BPB (block only)
        Dw      Ioctlin         ;  3 IOCTL read
        Dw      Read            ;  4 read
        Dw      Ndread          ;  5 nondestructive read
        Dw      Rxstat          ;  6 input status
        Dw      Inflush         ;  7 flush input buffer
        Dw      Write           ;  8 write
        Dw      Write           ;  9 write with verify
```

Figure 6-1. Continued. *(more)*

```
        Dw      Txstat          ; 10 output status
        Dw      Txflush         ; 11 flush output buffer
        Dw      Ioctlout        ; 12 IOCTL write
; Following are not used in this driver.....
        Dw      Zexit           ; 13 open (3.x only, not used)
        Dw      Zexit           ; 14 close (3.x only, not used)
        Dw      Zexit           ; 15 rem med (3.x only, not used)
        Dw      Zexit           ; 16 out until bsy (3.x only, not used)
        Dw      Zexit           ; 17
        Dw      Zexit           ; 18
        Dw      Zexit           ; 19 generic IOCTL request (3.2 only)
        Dw      Zexit           ; 20
        Dw      Zexit           ; 21
        Dw      Zexit           ; 22
        Dw      Zexit           ; 23 get logical drive map (3.2 only)
        Dw      Zexit           ; 24 set logical drive map (3.2 only)

Subttl  Driver Code
Page
;
;       The Strategy routine itself:
;
Strtegy Proc    Far
;       dbg     'S','R',' '
        Mov     Word Ptr CS:PackHd,BX   ; store the offset
        Mov     Word Ptr CS:PackHd+2,ES ; store the segment
        Ret
Strtegy Endp
;
Request1:                       ; async1 has been requested
        Push    Si              ; save SI
        Lea     Si,Asy_tab1     ; get the device unit table address
        Jmp     Short   Gen_request

Request2:                       ; async2 has been requested
        Push    Si              ; save SI
        Lea     Si,Asy_tab2     ; get unit table two's address

Gen_request:
;       dbg     'R','R',' '
        Pushf                   ; save all regs
        Cld
        Push    Ax
        Push    Bx
        Push    Cx
        Push    Dx
        Push    Di
        Push    Bp
        Push    Ds
        Push    Es
        Push    Cs              ; set DS = CS
```

Figure 6-1. Continued. *(more)*

```
         Pop     Ds
         Les     Bx,PackHd       ; get packet pointer
         Lea     Di,Asy_funcs    ; point DI to jump table
         Mov     Al,es:code[bx]  ; command code
         Cbw
         Add     Ax,Ax           ; double to word
         Add     Di,ax
         Jmp     [di]            ; go do it
;
;        Exit from driver request
;
ExitP    Proc    Far
Bsyexit:
         Mov     Ax,StsBsy
         Jmp     Short   Exit

Mchek:
BldBPB:
Zexit:   Xor     Ax,Ax
Exit:    Les     Bx,PackHd       ; get packet pointer
         Or      Ax,StsDne
         Mov     Es:Stat[Bx],Ax  ; set return status
         Pop     Es              ; restore registers
         Pop     Ds
         Pop     Bp
         Pop     Di
         Pop     Dx
         Pop     Cx
         Pop     Bx
         Pop     Ax
         Popf
         Pop     Si
         Ret
ExitP    Endp

Subttl   Driver Service Routines
Page

;        Read data from device

Read:
;        dbg     'R','d',' '
         Mov     Cx,Es:Count[bx] ; get requested nbr
         Mov     Di,Es:Xfer[bx]  ; get target pointer
         Mov     Dx,Es:Xseg[bx]
         Push    Bx              ; save for count fixup
         Push    Es
         Mov     Es,Dx
         Test    InStat[si],BadInp Or LostDt
         Je      No_lerr         ; no error so far...
         Add     Sp,4            ; error, flush SP
```

Figure 6-1. Continued. *(more)*

```
        And     InStat[si],Not ( BadInp Or LostDt )
        Mov     Ax,ErrRf        ; error, report it
        Jmp     Exit
No_lerr:
        Call    Get_in          ; go for one
        Or      Ah,Ah
        Jnz     Got_all         ; none to get now
        Stosb                   ; store it
        Loop    No_lerr         ; go for more
Got_all:
        Pop     Es
        Pop     Bx
        Sub     Di,Es:Xfer[bx]  ; calc number stored
        Mov     Es:Count[bx],Di ; return as count
        Jmp     Zexit

;       Nondestructive read from device

Ndread:
        Mov     Di,ifirst[si]
        Cmp     Di,iavail[si]
        Jne     Ndget
        Jmp     Bsyexit         ; buffer empty
Ndget:
        Push    Bx
        Mov     Bx,ibuf[si]
        Mov     Al,[bx+di]
        Pop     Bx
        Mov     Es:media[bx],al ; return char
        Jmp     Zexit

;       Input status request

Rxstat:
        Mov     Di,ifirst[si]
        Cmp     Di,iavail[si]
        Jne     Rxful
        Jmp     Bsyexit         ; buffer empty
Rxful:
        Jmp     Zexit           ; have data

;       Input flush request

Inflush:
        Mov     Ax,iavail[si]
        Mov     Ifirst[si],ax
        Jmp     Zexit

;       Output data to device

```

Figure 6-1. Continued. *(more)*

```
Write:
;       dbg     'W','r',' '
        Mov     Cx,es:count[bx]
        Mov     Di,es:xfer[bx]
        Mov     Ax,es:xseg[bx]
        Mov     Es,ax
Wlup:
        Mov     Al,es:[di]      ; get the byte
        Inc     Di
Wwait:
        Call    Put_out         ; put away
        Cmp     Ah,0
        Jne     Wwait           ; wait for room!
        Call    Start_output    ; get it going
        Loop    Wlup

        Jmp     Zexit

;       Output status request

Txstat:
        Mov     Ax,ofirst[si]
        Dec     Ax
        And     Ax,bufmsk
        Cmp     Ax,oavail[si]
        Jne     Txroom
        Jmp     Bsyexit         ; buffer full
Txroom:
        Jmp     Zexit           ; room exists

;       IOCTL read request, return line parameters

Ioctlin:
        Mov     Cx,es:count[bx]
        Mov     Di,es:xfer[bx]
        Mov     Dx,es:xseg[bx]
        Mov     Es,dx
        Cmp     Cx,10
        Je      Doiocin
        Mov     Ax,errbsl
        Jmp     Exit
Doiocin:
        Mov     Dx,port[si]     ; base port
        Mov     Dl,Lctrl        ; line status
        Mov     Cx,4            ; LCR, MCR, LSR, MSR
Getport:
        In      Al,dx
        Stos    Byte Ptr [DI]
        Inc     Dx
        Loop    Getport

```

Figure 6-1. Continued. *(more)*

```
        Mov     Ax,InSpec[si]   ; spec in flags
        Stos    Word Ptr [DI]
        Mov     Ax,OutSpec[si]  ; out flags
        Stos    Word Ptr [DI]
        Mov     Ax,baud[si]     ; baud rate
        Mov     Bx,di
        Mov     Di,offset Asy_baudt+2
        Mov     Cx,15
Baudcin:
        Cmp     [di],ax
        Je      Yesinb
        Add     Di,4
        Loop    Baudcin
Yesinb:
        Mov     Ax,-2[di]
        Mov     Di,bx
        Stos    Word Ptr [DI]
        Jmp     Zexit

;       Flush output buffer request

Txflush:
        Mov     Ax,oavail[si]
        Mov     Ofirst[si],ax
        Jmp     Zexit

;       IOCTL request: change line parameters for this driver

Ioctlout:
        Mov     Cx,es:count[bx]
        Mov     Di,es:xfer[bx]
        Mov     Dx,es:xseg[bx]
        Mov     Es,dx
        Cmp     Cx,10
        Je      Doiocout
        Mov     Ax,errbsl
        Jmp     Exit

Doiocout:
        Mov     Dx,port[si]     ; base port
        Mov     Dl,Lctrl        ; line ctrl
        Mov     Al,es:[di]
        Inc     Di
        Or      Al,Dlab         ; set baud
        Out     Dx,al
        Clc
        Jnc     $+2
        Inc     Dx              ; mdm ctrl
        Mov     Al,es:[di]
        Or      Al,Usr2         ; Int Gate
        Out     Dx,al
```

Figure 6-1. Continued. *(more)*

```
        Add     Di,3            ; skip LSR,MSR
        Mov     Ax,es:[di]
        Add     Di,2
        Mov     InSpec[si],ax
        Mov     Ax,es:[di]
        Add     Di,2
        Mov     OutSpec[si],ax
        Mov     Ax,es:[di]      ; set baud
        Mov     Bx,di
        Mov     Di,offset Asy_baudt
        Mov     Cx,15
Baudcout:
        Cmp     [di],ax
        Je      Yesoutb
        Add     Di,4
        Loop    Baudcout

        Mov     Dl,Lctrl        ; line ctrl
        In      Al,dx           ; get LCR data
        And     Al,not Dlab     ; strip
        Clc
        Jnc     $+2
        Out     Dx,al           ; put back
        Mov     Ax,ErrUm        ; "unknown media"
        Jmp     Exit

Yesoutb:
        Mov     Ax,2[di]        ; get divisor
        Mov     Baud[si],ax     ; save to report later
        Mov     Dx,port[si]     ; set divisor
        Out     Dx,al
        Clc
        Jnc     $+2
        Inc     Dx
        Mov     Al,ah
        Out     Dx,al
        Clc
        Jnc     $+2
        Mov     Dl,Lctrl        ; line ctrl
        In      Al,dx           ; get LCR data
        And     Al,not Dlab     ; strip
        Clc
        Jnc     $+2
        Out     Dx,al           ; put back
        Jmp     Zexit

Subttl  Ring Buffer Routines
Page

Put_out Proc    Near    ; puts AL into output ring buffer
        Push    Cx
```

Figure 6-1. Continued. *(more)*

```
        Push    Di
        Pushf
        Cli
        Mov     Cx,oavail[si]   ; put ptr
        Mov     Di,cx
        Inc     Cx              ; bump
        And     Cx,bufmsk
        Cmp     Cx,ofirst[si]   ; overflow?
        Je      Poerr           ; yes, don't
        Add     Di,obuf[si]     ; no
        Mov     [di],al         ; put in buffer
        Mov     Oavail[si],cx
;       dbg     'p','o',' '
        Mov     Ah,0
        Jmp     Short   Poret
Poerr:
        Mov     Ah,-1
Poret:
        Popf
        Pop     Di
        Pop     Cx
        Ret
Put_out Endp

Get_out Proc    Near    ; gets next character from output ring buffer
        Push    Cx
        Push    Di
        Pushf
        Cli
        Mov     Di,ofirst[si]   ; get ptr
        Cmp     Di,oavail[si]   ; put ptr
        Jne     Ngoerr
        Mov     Ah,-1           ; empty
        Jmp     Short   Goret
Ngoerr:
;       dbg     'g','o',' '
        Mov     Cx,di
        Add     Di,obuf[si]
        Mov     Al,[di]         ; get char
        Mov     Ah,0
        Inc     Cx              ; bump ptr
        And     Cx,bufmsk       ; wrap
        Mov     Ofirst[si],cx
Goret:
        Popf
        Pop     Di
        Pop     Cx
        Ret
Get_out Endp

Put_in  Proc    Near    ; puts the char from AL into input ring buffer
```

Figure 6-1. Continued. *(more)*

```
        Push    Cx
        Push    Di
        Pushf
        Cli
        Mov     Di,iavail[si]
        Mov     Cx,di
        Inc     Cx
        And     Cx,bufmsk
        Cmp     Cx,ifirst[si]
        Jne     Npierr
        Mov     Ah,-1
        Jmp     Short   Piret
Npierr:
        Add     Di,ibuf[si]
        Mov     [di],al
        Mov     Iavail[si],cx
;       dbg     'p','i',' '
        Mov     Ah,0
Piret:
        Popf
        Pop     Di
        Pop     Cx
        Ret
Put_in  Endp

Get_in  Proc    Near    ; gets one from input ring buffer into AL
        Push    Cx
        Push    Di
        Pushf
        Cli
        Mov     Di,ifirst[si]
        Cmp     Di,iavail[si]
        Je      Gierr
        Mov     Cx,di
        Add     Di,ibuf[si]
        Mov     Al,[di]
        Mov     Ah,0
;       dbg     'g','i',' '
        Inc     Cx
        And     Cx,bufmsk
        Mov     Ifirst[si],cx
        Jmp     Short   Giret
Gierr:
        Mov     Ah,-1
Giret:
        Popf
        Pop     Di
        Pop     Cx
        Ret
Get_in  Endp

```

Figure 6-1. Continued. *(more)*

```
Subttl  Interrupt Dispatcher Routine
Page

Asy1isr:
        Sti
        Push    Si
        Lea     Si,asy_tab1
        Jmp     Short   Int_serve

Asy2isr:
        Sti
        Push    Si
        Lea     Si,asy_tab2

Int_serve:
        Push    Ax              ; save all regs
        Push    Bx
        Push    Cx
        Push    Dx
        Push    Di
        Push    Ds
        Push    Cs              ; set DS = CS
        Pop     Ds
Int_exit:
;       dbg     'I','x',' '
        Mov     Dx,Port[si]     ; base address
        Mov     Dl,IntId        ; check Int ID
        In      Al,Dx
        Cmp     Al,00h          ; dispatch filter
        Je      Int_modem
        Jmp     Int_mo_no
Int_modem:
;       dbg     'M','S',' '
        Mov     Dl,Mstat
        In      Al,dx           ; read MSR content
        Test    Al,CDlvl        ; carrier present?
        Jnz     Msdsr           ; yes, test for DSR
        Test    OutSpec[si],OutCdf      ; no, is CD off line?
        Jz      Msdsr
        Or      InStat[si],OffLin
Msdsr:
        Test    Al,DSRlvl       ; DSR present?
        Jnz     Dsron           ; yes, handle it
        Test    OutSpec[si],OutDSR      ; no, is DSR throttle?
        Jz      Dsroff
        Or      OtStat[si],LinDSR       ; yes, throttle down
Dsroff:
        Test    OutSpec[si],OutDrf      ; is DSR off line?
        Jz      Mscts
        Or      InStat[si],OffLin       ; yes, set flag
        Jmp     Short   Mscts
```

Figure 6-1. Continued. *(more)*

```
Dsron:
        Test    OtStat[si],LinDSR       ; throttled for DSR?
        Jz      Mscts
        Xor     OtStat[si],LinDSR       ; yes, clear it out
        Call    Start_output
Mscts:
        Test    Al,CTSlvl       ; CTS present?
        Jnz     Ctson           ; yes, handle it
        Test    OutSpec[si],OutCTS      ; no, is CTS throttle?
        Jz      Int_exit2
        Or      OtStat[si],LinCTS       ; yes, shut it down
        Jmp     Short   Int_exit2
Ctson:
        Test    OtStat[si],LinCTS       ; throttled for CTS?
        Jz      Int_exit2
        Xor     OtStat[si],LinCTS       ; yes, clear it out
        Jmp     Short   Int_exit1
Int_mo_no:
        Cmp     Al,02h
        Jne     Int_tx_no
Int_txmit:
;       dbg     'T','x',' '
Int_exit1:
        Call    Start_output    ; try to send another
Int_exit2:
        Jmp     Int_exit
Int_tx_no:
        Cmp     Al,04h
        Jne     Int_rec_no
Int_receive:
;       dbg     'R','x',' '
        Mov     Dx,port[si]
        In      Al,dx           ; take char from 8250
        Test    OutSpec[si],OutXon  ; is XON/XOFF enabled?
        Jz      Stuff_in        ; no
        Cmp     Al,'S' And 01FH ; yes, is this XOFF?
        Jne     Isq             ; no, check for XON
        Or      OtStat[si],LinXof ; yes, disable output
        Jmp     Int_exit2       ; don't store this one
Isq:
        Cmp     Al,'Q' And 01FH ; is this XON?
        Jne     Stuff_in        ; no, save it
        Test    OtStat[si],LinXof ; yes, waiting?
        Jz      Int_exit2       ; no, ignore it
        Xor     OtStat[si],LinXof ; yes, clear the XOFF bit
        Jmp     Int_exit1       ; and try to resume xmit
Int_rec_no:
        Cmp     Al,06h
        Jne     Int_done
Int_rxstat:
;       dbg     'E','R',' '
```

Figure 6-1. Continued. *(more)*

```
        Mov     Dl,Lstat
        In      Al,dx
        Test    InSpec[si],InEpc ; return them as codes?
        Jz      Nocode          ; no, just set error alarm
        And     Al,AnyErr       ; yes, mask off all but error bits
        Or      Al,080h
Stuff_in:
        Call    Put_in          ; put input char in buffer
        Cmp     Ah,0            ; did it fit?
        Je      Int_exit3       ; yes, all OK
        Or      InStat[si],LostDt  ; no, set DataLost bit
Int_exit3:
        Jmp     Int_exit
Nocode:
        Or      InStat[si],BadInp
        Jmp     Int_exit3
Int_done:
        Clc
        Jnc     $+2
        Mov     Al,EOI          ; all done now
        Out     PIC_b,Al
        Pop     Ds              ; restore regs
        Pop     Di
        Pop     Dx
        Pop     Cx
        Pop     Bx
        Pop     Ax
        Pop     Si
        Iret

Start_output    Proc    Near
        Test    OtStat[si],LinIdl ; Blocked?
        Jnz     Dont_start      ; yes, no output
        Mov     Dx,port[si]     ; no, check UART
        Mov     Dl,Lstat
        In      Al,Dx
        Test    Al,xhre         ; empty?
        Jz      Dont_start      ; no
        Call    Get_out         ; yes, anything waiting?
        Or      Ah,Ah
        Jnz     Dont_start      ; no
        Mov     Dl,RxBuf        ; yes, send it out
        Out     Dx,al
;       dbg     's','o',' '
Dont_start:
        ret
Start_output    Endp

Subttl  Initialization Request Routine
Page

```

Figure 6-1. Continued. *(more)*

```
Init:   Lea     Di,$            ; release rest...
        Mov     Es:Xfer[bx],Di
        Mov     Es:Xseg[bx],Cs

        Mov     Dx,Port[si]     ; base port
        Mov     Dl,Lctrl
        Mov     Al,Dlab         ; enable divisor
        Out     Dx,Al
        Clc
        Jnc     $+2
        Mov     Dl,RxBuf
        Mov     Ax,Baud[si]     ; set baud
        Out     Dx,Al
        Clc
        Jnc     $+2
        Inc     Dx
        Mov     Al,Ah
        Out     Dx,Al
        Clc
        Jnc     $+2

        Mov     Dl,Lctrl        ; set LCR
        Mov     Al,OtStat[si]   ; from table
        Out     Dx,Al
        Mov     OtStat[si],0    ; clear status
        Clc
        Jnc     $+2
        Mov     Dl,IntEn        ; IER
        Mov     Al,AllInt       ; enable ints in 8250
        Out     Dx,Al
        Clc
        Jnc     $+2
        Mov     Dl,Mctrl        ; set MCR
        Mov     Al,InStat[si]   ; from table
        Out     Dx,Al
        Mov     InStat[si],0    ; clear status

ClRgs:  Mov     Dl,Lstat        ; clear LSR
        In      Al,Dx
        Mov     Dl,RxBuf        ; clear RX reg
        In      Al,Dx
        Mov     Dl,Mstat        ; clear MSR
        In      Al,Dx
        Mov     Dl,IntId        ; IID reg
        In      Al,Dx
        In      Al,Dx
        Test    Al,1            ; int pending?
        Jz      ClRgs           ; yes, repeat

        Cli
        Xor     Ax,Ax           ; set int vec
```

Figure 6-1. Continued. *(more)*

```
880 :           Mov     Es,Ax
881 :           Mov     Di,Vect[si]
882 :           Mov     Ax,IsrAdr[si]     ; from table
883 :           Stosw
884 :           Mov     Es:[di],cs
885 :
886 :           In      Al,PIC_e          ; get 8259
887 :           And     Al,0E7h           ; com1/2 mask
888 :           Clc
889 :           Jnb     $+2
890 :           Out     PIC_e,Al
891 :           Sti
892 :
893 :           Mov     Al,EOI            ; now send EOI just in case
894 :           Out     PIC_b,Al
895 :
896 : ;         dbg     'D','I',' '       ; driver installed
897 :           Jmp     Zexit
898 :
899 : Driver    Ends
900 :           End
```

Figure 6-1. Continued.

The first part of the driver source code (after the necessary MASM housekeeping details in lines 1 through 8) is a commented-out macro definition (lines 10 through 32). This macro is used only during debugging and is part of a debugging technique that requires no sophisticated hardware and no more complex debugging program than the venerable DEBUG.COM. (Debugging techniques are discussed after the presentation of the driver program itself.)

Definitions

The actual driver source program consists of three sets of EQU definitions (lines 34 through 194), followed by the modular code and data areas (lines 197 through 900). The first set of definitions (lines 34 through 82) gives symbolic names to the permissible values for MS-DOS device-driver control bits and the device-driver structures.

The second set of definitions (lines 84 through 145) assigns names to the ports and bit values that are associated with the IBM hardware—both the 8259 PIC and the 8250 UART. The third set of definitions (lines 147 through 194) assigns names to the control values and structures associated with this driver.

The definition method used here is recommended for all drivers. To move this driver from the IBM architecture to some other hardware, the major change required to the program would be reassignment of the port addresses and bit values in lines 84 through 145.

The control values and structures for this specific driver (defined in the third EQU set) provide the means by which the separate support program can modify the actions of each of the two logical drivers. They also permit the driver to return status information to both the support program and the using program as necessary. Only a few features are implemented, but adequate space for expansion is provided. The addition of a few more definitions in this area and one or two extra procedures in the code section would do all that is necessary to extend the driver's capabilities to such features as automatic expansion of tab characters, case conversion, and so forth, should they be desired.

Headers and structure tables

The driver code itself starts with a linked pair of device-driver header blocks, one for *ASY1* (lines 201 through 207) and the other for *ASY2* (lines 208 through 213). Following the headers, in lines 215 through 236, are a commented-out space reservation used by the debugging procedure (line 215), the pointer to the command packet (line 219), and the baud-rate conversion table (lines 221 through 236).

The conversion table is followed by structure tables containing all data unique to *ASY1* (lines 239 through 242) and *ASY2* (lines 244 through 247). After the structure tables, buffer areas are reserved in lines 249 through 254. One input buffer and one output buffer are reserved for each port. All buffers are the same size; for simplicity, buffer size is given a name (at line 249) so that it can be changed by editing a single line of the program.

The size is arbitrary in this case, but if file transfers are anticipated, the buffer should be able to hold at least 2 seconds' worth of data (240 bytes at 1200 bps) to avoid data loss during writes to disk. Whatever size is chosen should be a power of 2 for simple pointer arithmetic and, if video display is intended, should not be less than 8 bytes, to prevent losing characters when the screen scrolls.

If additional ports are desired, more headers can be added after line 213; corresponding structure tables for each driver, plus matching pairs of buffers, would also be necessary. The final part of this area is the dispatch table (lines 256 through 284), which lists offsets of all request routines in the driver; its use is discussed below.

Strategy and Request routines

With all data taken care of, the program code begins at the Strategy routine (lines 289 through 296), which is used by both ports. This code saves the command packet address passed to it by MS-DOS for use by the Request routine and returns to MS-DOS.

The Request routines (lines 298 through 567) are also shared by both ports, but the two drivers are distinguished by the address placed into the SI register. This address points to the structure table that is unique to each port and contains such data as the port's base address, the associated hardware interrupt vector, the interrupt service routine offset within the driver's segment, the base offsets of the input and output buffers for that port, two pointers for each of the buffers, and the input and output status conditions (including baud rate) for the port. The only difference between one port's driver and the other's is the data pointed to by SI; all Request routine code is shared by both ports.

Each driver's Request routine has a unique entry point (at line 298 for *ASY1* and at line 303 for *ASY2*) that saves the original content of the SI register and then loads it with the address of the structure table for that driver. The routines then join as a common stream at line 307 (*Gen_request*).

This common code preserves all other registers used (lines 309 through 318), sets DS equal to CS (lines 319 and 320), retrieves the command-packet pointer saved by the Strategy routine (line 321), uses the pointer to get the command code (line 323), uses the code to calculate an offset into a table of addresses (lines 324 through 326), and performs an indexed jump (lines 322 and 327) by way of the dispatch table (lines 256 through 284) to the routine that executes the requested command (at line 336, 360, 389, 404, 414, 421, 441, 453, 500, or 829).

Although the device-driver specifications for MS-DOS version 3.2 list command request codes ranging from 0 to 24, not all are used. Earlier versions of MS-DOS permitted only 0 to 12 (versions 2.x) or 0 to 16 (versions 3.0 and 3.1) codes. In this driver, all 24 codes are accounted for; those not implemented in this driver return a DONE and NO ERROR status to the caller. Because the Request routine is called only by MS-DOS itself, there is no check for invalid codes. Actually, because the header attribute bits are *not* set to specify that codes 13 through 24 are valid, the 24 bytes occupied by their table entries (lines 273 through 284) could be saved by omitting the entries. They are included only to show how nonexistent commands can be accommodated.

Immediately following the dispatch indexed jump, at lines 329 through 353 within the same PROC declaration, is the common code used by all Request routines to store status information in the command packet, restore the registers, and return to the caller. The alternative entry points for BUSY status (line 332), NO ERROR status (line 338), or an error code (in the AX register at entry to *Exit*, line 339) not only save several bytes of redundant code but also improve readability of the code by providing unique single labels for BUSY, NO ERROR, and ERROR return conditions.

All of the Request routines, except for the *Init* code at line 829, immediately follow the dispatching shell in lines 358 through 568. Each is simplified to perform just one task, such as read data in or write data out. The *Read* routine (lines 360 through 385) is typical: First, the requested byte count and user's buffer address are obtained from the command packet. Next, the pointer to the command packet is saved with a PUSH instruction, so that the ES and BX registers can be used for a pointer to the port's input buffer.

Before the *Get_in* routine that actually accesses the input buffer is called, the input status byte is checked (line 368). If an error condition is flagged, lines 370 through 373 clear the status flag, flush the saved pointers from the stack, and jump to the error-return exit from the driver. If no error exists, line 375 calls *Get_in* to access the input buffer and lines 376 and 377 determine whether a byte was obtained. If a byte is found, it is stored in the user's buffer by line 378, and line 379 loops back to get another byte until the requested count has been obtained or until no more bytes are available. In practice, the count is an upper limit and the loop is normally broken when data runs out.

No matter how it happens, control eventually reaches the *Got_all* routine and lines 381 and 382, where the saved pointers to the command packet are restored from the stack. Lines 383 and 384 adjust the count value in the packet to reflect the actual number of bytes obtained. Finally, line 385 jumps to the normal, no-error exit from the driver.

Buffering

Both buffers for each driver are of the type known as circular, or ring, buffers. Effectively, such a buffer is endless; it is accessed via pointers, and when a pointer increments past the end of the buffer, the pointer returns to the buffer's beginning. Two pointers are used here for each buffer, one to put data into it and one to get data out. The *get* pointer always points to the next byte to be read; the *put* pointer points to where the next byte will be written, just past the last byte written to the buffer.

If both pointers point to the same byte, the buffer is empty; the next byte to be read has not yet been written. The full-buffer condition is more difficult to test for: The *put* pointer is incremented and compared with the *get* pointer; if they are equal, doing a write would force a false buffer-empty condition, so the buffer must be full.

All buffer manipulation is done via four procedures (lines 569 through 674). *Put_out* (lines 572 through 596) writes a byte to the driver's output buffer or returns a buffer-full indication by setting AH to 0FFH. *Get_out* (lines 598 through 622) gets a byte from the output buffer or returns 0FFH in AH to indicate that no byte is available. *Put_in* (lines 624 through 648) and *Get_in* (lines 650 through 674) do exactly the same as *Put_out* and *Get_out*, but for the input buffer. These procedures are used both by the Request routines and by the hardware interrupt service routine (ISR).

Interrupt service routines

The most complex part of this driver is the ISR (lines 676 through 806), which decides which of the four possible services for a port is to be performed and where. Like the Request routines, the ISR provides unique entry points for each port (line 679 for *ASY1* and line 685 for *ASY2*); these entry points first preserve the SI register and then load it with the address of the port's structure table. With SI indicating where the actions are to be performed, the two entries then merge at line 690 into common code that first preserves all registers to be used by the ISR (lines 690 through 698) and then tests for each of the four possible types of service and performs each requested action.

Much of the complexity of the ISR is in the decoding of modem-status conditions. Because the resulting information is not used by this driver (although it could be used to prevent attempts to transmit while off line), these ISR options can be removed so that only the Transmit and Receive interrupts are serviced. To do this, *AllInt* (at line 145) should be changed from the OR of all four bits to include only the transmit and receive bits (03H, or 00000011B).

The transmit and receive portions of the ISR incorporate XON/XOFF flow control (for transmitted data only) by default. This control is done at the ISR level, rather than in the using program, to minimize the time required to respond to an incoming XOFF signal. Presence of the flow-control decisions adds complexity to what would otherwise be extremely simple actions.

Flow control is enabled or disabled by setting the *OutSpec* word in the structure table with the Driver Status utility (presented later) via the IOCTL function (Interrupt 21H Function 44H). When flow control is enabled, any XOFF character (11H) that is received halts all outgoing data until XON (13H) is received. No XOFF or XON is retained in the input buffer to be sent on to any program, although all patterns other than XOFF and XON *are* passed through by the driver. When flow control is disabled, the driver passes all patterns in both directions. For binary file transfer, flow control must be disabled.

The transmit action is simple: The code merely calls the *Start_output* procedure at line 750. *Start_output* is described in detail below.

The receive action is almost as simple as transmit, except for the flow-control testing. First, the ISR takes the received byte from the UART (lines 758 and 759) to avoid any chance of an overrun error. The ISR then tests the input specifier (at line 760) to determine whether flow control is in effect. If it is not, processing jumps directly to line 784 to store the received byte in the input buffer with *Put_in* (line 785).

If flow control is active, however, the received byte is compared with the XOFF character (lines 762 through 765). If the byte matches, output is disabled and the byte is ignored. If the byte is not XOFF, it is compared with XON (lines 766 through 768). If it is not XON either, control jumps to line 784. If the byte is XON, output is re-enabled if it was disabled. Regardless, the XON byte itself is ignored.

When control reaches *Stuff_in* at line 784, *Put_in* is called to store the received byte in the input buffer. If there is no room for it, a lost-databit is set in the input status flags (line 788); otherwise, the receive routine is finished.

If the interrupt was a line-status action, the LSR is read (lines 776 through 779). If the input specifier so directs, the content is converted to an IBM PC extended graphics character by setting bit 7 to 1 and the character is stored in the input buffer as if it were a received byte. Otherwise, the Line Status interrupt merely sets the generic *BadInp* error bit in the input status flags, which can be read with the IOCTL Read function of the driver.

When all ISR action is complete, lines 794 through 806 restore machine conditions to those existing at the time of the interrupt and return to the interrupted procedure.

The *Start_output* routine

Start_output (lines 808 through 824) is a routine that, like the four buffer procedures, is used by both the Request routines and the ISR. Its purpose is to initiate transmission of a byte, provided that output is not blocked by flow control, the UART Transmit Holding Register is empty, and a byte to be transmitted exists in the output ring buffer. This routine uses the *Get_out* buffer routine to access the buffer and determine whether a byte is available. If all conditions are met, the byte is sent to the UART holding register by lines 819 and 820.

The Initialization Request routine

The Initialization Request routine (lines 829 through 897) is critical to successful operation of the driver. This routine is placed last in the package so that it can be discarded as soon as it has served its purpose by installing the driver. It is essential to clear each register of the 8250 by reading its contents before enabling the interrupts and to loop through this action until the 8250 finally shows no requests pending. The strange *Clc jnc $+2* sequence that appears repeatedly in this routine is a time delay required by high-speed machines (6 MHz and up) so that the 8250 has time to settle before another access is attempted; the delay does no harm on slower machines.

Using COMDVR

The first step in using this device driver is assembling it with the Microsoft Macro Assembler (MASM). Next, use the Microsoft Object Linker (LINK) to create a .EXE file. Convert the .EXE file into a binary image file with the EXE2BIN utility. Finally, include the line *DEVICE=COMDVR.SYS* in the CONFIG.SYS file so that COMDVR will be installed when the system is restarted.

Note: The number and colon at the beginning of each line in the program listings in this article are for reference only and should not be included in the source file.

Figure 6-2 shows the sequence of actions required, assuming that EDLIN is used for modifying (or creating) the CONFIG.SYS file and that all commands are issued from the root directory of the boot drive.

Creating the driver:

```
C>MASM COMDVR;   <Enter>
C>LINK COMDVR;   <Enter>
C>EXE2BIN COMDVR.EXE COMDVR.SYS   <Enter>
```

Modifying CONFIG.SYS (^Z = press Ctrl-Z):

```
C>EDLIN CONFIG.SYS   <Enter>
*#I   <Enter>
*DEVICE=COMDVR.SYS   <Enter>
*^Z   <Enter>
*E   <Enter>
```

Figure 6-2. Assembling, linking, and installing COMDVR.

Because the devices installed by COMDVR do not use the standard MS-DOS device names, no conflict occurs with any program that uses conventional port references. Such a program will not use the driver, and no problems should result if the program is well behaved and restores all interrupt vectors before returning to MS-DOS.

Device-driver debugging techniques

The debugging of device drivers, like debugging for any part of MS-DOS itself, is more difficult than normal program checking because the debugging program, DEBUG.COM or DEBUG.EXE, itself uses MS-DOS functions to display output. When these functions are being checked, their use by DEBUG destroys the data being examined. And because MS-DOS always saves its return address in the same location, any call to a function from inside the operating system usually causes a system lockup that can be cured only by shutting the system down and powering up again.

One way to overcome this difficulty is to purchase costly debugging tools. An easier way is to bypass the problem: Instead of using MS-DOS functions to track program operation, write data directly to video RAM, as in the macro *DBG* (lines 10 through 32 of COMDVR.ASM).

This macro is invoked with a three-character parameter string at each point in the program a progress report is desired. Each invocation has its own unique three-character string so that the sequence of actions can be read from the screen. When invoked, *DBG* expands into code that saves all registers and then writes the three-character string to video RAM. Only the top 10 lines of the screen (800 characters, or 1600 bytes) are used: The macro uses a single far pointer to the area and treats the video RAM like a ring buffer.

The pointer, *Dbgptr* (line 215), is set up for use with the monochrome adapter and points to location B000:0000H; to use a CGA or EGA (in CGA mode), the location should be changed to B800:0000H.

Most of the frequently used Request routines, such as *Read* and *Write*, have calls to *DBG* as their first lines (for example, lines 361 and 422). As shown, these calls are commented out, but for debugging, the source file should be edited so that all the calls and the macro itself are enabled.

With *DBG* active, the top 10 lines of the display are overwritten with a continual sequence of reports, such as *RR Tx*, put directly into video RAM. Because MS-DOS functions are not used, no interference with the driver itself can occur.

Although this technique prevents normal use of the system during debugging, it greatly simplifies the problem of knowing what is happening in time-critical areas, such as hardware interrupt service. In addition, all invocations of *DBG* in the critical areas are in conditional code that is executed only when the driver is working as it should.

Failure to display the *pi* message, for instance, indicates that the received-data hardware interrupt is not being serviced, and absence of *go* after an *Ix* report shows that data is not being sent out as it should.

Of course, once debugging is complete, the calls to *DBG* should be deleted or commented out. Such calls are usually edited out of the source code before release. In this case, they remain to demonstrate the technique and, most particularly, to show placement of the calls to provide maximum information with minimal clutter on the screen.

A simple modem engine

The second part of this package is the modem engine itself (ENGINE.ASM), shown in the listing in Figure 6-3. The main loop of this program consists of only a dozen lines of code (lines 9 through 20). Of these, five (lines 9 through 13) are devoted to establishing initial contact between the program and the serial-port driver and two (lines 19 and 20) are for returning to command level at the program's end.

Thus, only five lines of code (lines 14 through 18) actually carry out the bulk of the program as far as the main loop is concerned. Four of these lines are calls to subroutines that get and put data from and to the console and the serial port; the fifth is the JMP that closes the loop. This structure underscores the fact that a basic modem engine is simply a data-transfer loop.

```
        TITLE   engine

CODE    SEGMENT PUBLIC 'CODE'

        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE

        ORG     0100h

START:  mov     dx,offset devnm ; open named device (ASY1)
        mov     ax,3d02h
        int     21h
        mov     handle,ax       ; save the handle
        jc      quit
alltim: call    getmdm          ; main engine loop
        call    putcrt
        call    getkbd
        call    putmdm
        jmp     alltim
quit:   mov     ah,4ch          ; come here to quit
        int     21h

getmdm  proc                    ; get input from modem
        mov     cx,256
        mov     bx,handle
        mov     dx,offset mbufr
        mov     ax,3F00h
        int     21h
        jc      quit
        mov     mdlen,ax
        ret
getmdm  endp

getkbd  proc                    ; get input from keyboard
        mov     kblen,0         ; first zero the count
        mov     ah,11           ; key pressed?
        int     21h
        inc     al
        jnz     nogk            ; no
        mov     ah,7            ; yes, get it
        int     21h
        cmp     al,3            ; was it Ctrl-C?
        je      quit            ; yes, get out
        mov     kbufr,al        ; no, save it
        inc     kblen
        cmp     al,13           ; was it Enter?
        jne     nogk            ; no
```

Figure 6-3. ENGINE.ASM. *(more)*

```
47 :          mov     byte ptr kbufr+1,10 ; yes, add LF
48 :          inc     kblen
49 : nogk:    ret
50 : getkbd   endp
51 :
52 : putmdm   proc                        ; put output to modem
53 :          mov     cx,kblen
54 :          jcxz    nopm
55 :          mov     bx,handle
56 :          mov     dx,offset kbufr
57 :          mov     ax,4000h
58 :          int     21h
59 :          jc      quit
60 : nopm:    ret
61 : putmdm   endp
62 :
63 : putcrt   proc                        ; put output to CRT
64 :          mov     cx,mdlen
65 :          jcxz    nopc
66 :          mov     bx,1
67 :          mov     dx,offset mbufr
68 :          mov     ah,40h
69 :          int     21h
70 :          jc      quit
71 : nopc:    ret
72 : putcrt   endp
73 :
74 : devnm    db      'ASY1',0            ; miscellaneous data and buffers
75 : handle   dw      0
76 : kblen    dw      0
77 : mdlen    dw      0
78 : mbufr    db      256 dup (0)
79 : kbufr    db      80 dup (0)
80 :
81 : CODE     ENDS
82 :          END     START
```

Figure 6-3. Continued.

Because the details of timing and data conversion are handled by the driver code, each of the four subroutines is — to show just how simple the whole process is — essentially a buffered interface to the MS-DOS Read File or Device or Write File or Device routine.

For example, the *getmdm* procedure (lines 22 through 31) asks MS-DOS to read a maximum of 256 bytes from the serial device and then stores the number actually read in a word named *mdlen.* The driver returns immediately, without waiting for data, so the normal number of bytes returned is either 0 or 1. If screen scrolling causes the loop to be delayed, the count might be higher, but it should never exceed about a dozen characters.

When called, the *putcrt* procedure (lines 63 through 72) checks the value in *mdlen.* If the value is zero, *putcrt* does nothing; otherwise, it asks MS-DOS to write that number of bytes from *mbufr* (where *getmdm* put them) to the display, and then it returns.

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

Note that *getkbd* does not use the Read File or Device function, because that would wait for a keystroke and the loop must never wait for reception. Instead, it uses the MS-DOS functions that test keyboard status (0BH) and read a key without echo (07H). In addition, special treatment is given to the Enter key (lines 45 through 48): A linefeed is inserted in *kbufr* immediately behind Enter and *kblen* is set to 2.

A Ctrl-C keystroke ends program operation; it is detected in *getkbd* (line 41) and causes immediate transfer to the *quit* label (line 19) at the end of the main loop. Because ENGINE uses only permanently resident routines, there is no need for any uninstallation before returning to the MS-DOS command prompt.

ENGINE.ASM is written to be used as a .COM file. Assemble and link it the same as COMDVR.SYS (Figure 6-2) but use the extension COM instead of SYS; no change to CONFIG.SYS is needed.

The driver-status utility: CDVUTL.C

The driver-status utility program CDVUTL.C, presented in Figure 6-4, permits either of the two drivers (*ASY1 and ASY2*) to be reconfigured after being installed, to suit different needs. After one of the drivers has been specified (port 1 or port 2), the baud rate, word length, parity, and number of stop bits can be changed; each change is made independently, with no effect on any of the other characteristics. Additionally, flow control can be switched between two types of hardware handshaking—the software XON/XOFF control or disabled—and error reporting can be switched between character-oriented and message-oriented operation.

```
/* cdvutl.c - COMDVR Utility
 *      Jim Kyle - 1987
 *      for use with COMDVR.SYS Device Driver
 */

#include <stdio.h>                  /* i/o definitions      */
#include <conio.h>                  /* special console i/o  */
#include <stdlib.h>                 /* misc definitions     */
#include <dos.h>                    /* defines intdos()     */

/*      the following define the driver status bits       */

#define HWINT 0x0800            /* MCR, first word, HW Ints gated */
#define o_DTR 0x0200            /* MCR, first word, output DTR    */
#define o_RTS 0x0100            /* MCR, first word, output RTS    */

#define m_PG  0x0010            /* LCR, first word, parity ON     */
#define m_PE  0x0008            /* LCR, first word, parity EVEN   */
```

Figure 6-4. CDVUTL.C *(more)*

```
#define m_XS  0x0004           /* LCR, first word, 2 stop bits   */
#define m_WL  0x0003           /* LCR, first word, wordlen mask  */

#define i_CD  0x8000           /* MSR, 2nd word, Carrier Detect  */
#define i_RI  0x4000           /* MSR, 2nd word, Ring Indicator  */
#define i_DSR 0x2000           /* MSR, 2nd word, Data Set Ready  */
#define i_CTS 0x1000           /* MSR, 2nd word, Clear to Send   */

#define l_SRE 0x0040           /* LSR, 2nd word, Xmtr SR Empty   */
#define l_HRE 0x0020           /* LSR, 2nd word, Xmtr HR Empty   */
#define l_BRK 0x0010           /* LSR, 2nd word, Break Received  */
#define l_ER1 0x0008           /* LSR, 2nd word, FrmErr          */
#define l_ER2 0x0004           /* LSR, 2nd word, ParErr          */
#define l_ER3 0x0002           /* LSR, 2nd word, OveRun          */
#define l_RRF 0x0001           /* LSR, 2nd word, Rcvr DR Full    */

/*           now define CLS string for ANSI.SYS            */
#define CLS    "\033[2J"

FILE * dvp;
union REGS rvs;
int iobf [ 5 ];

main ()
{ cputs ( "\nCDVUTL - COMDVR Utility Version 1.0 - 1987\n" );
  disp ();                         /* do dispatch loop           */
}

disp ()                            /* dispatcher; infinite loop  */
{ int c,
    u;
  u = 1;
  while ( 1 )
    { cputs ( "\r\n\tCommand (? for help): " );
      switch ( tolower ( c = getche ()))   /* dispatch           */
        {
        case '1' :                  /* select port 1             */
          fclose ( dvp );
          dvp = fopen ( "ASY1", "rb+" );
             u = 1;
          break;

        case '2' :                  /* select port 2             */
          fclose ( dvp );
          dvp = fopen ( "ASY2", "rb+" );
          u = 2;
          break;

        case 'b' :                  /* set baud rate             */
          if ( iobf [ 4 ] == 300 )
            iobf [ 4 ] = 1200;
```

Figure 6-4. Continued. *(more)*

```
          else
            if ( iobf [ 4 ] == 1200 )
              iobf [ 4 ] = 2400;
          else
            if ( iobf [ 4 ] == 2400 )
              iobf [ 4 ] = 9600;
          else
            iobf [ 4 ] = 300;
          iocwr ();
          break;

        case 'e' :                  /* set parity even            */
          iobf [ 0 ] |= ( m_PG + m_PE );
          iocwr ();
          break;

        case 'f' :                  /* toggle flow control        */
          if ( iobf [ 3 ] == 1 )
            iobf [ 3 ] = 2;
          else
            if ( iobf [ 3 ] == 2 )
              iobf [ 3 ] = 4;
          else
            if ( iobf [ 3 ] == 4 )
              iobf [ 3 ] = 0;
          else
            iobf [ 3 ] = 1;
          iocwr ();
          break;

        case 'i' :                  /* initialize MCR/LCR to 8N1 : */
          iobf [ 0 ] = ( HWINT + o_DTR + o_RTS + m_WL );
          iocwr ();
          break;

        case '?' :                  /* this help list             */
          cputs ( CLS );            /* clear the display          */
         center ( "COMMAND LIST \n" );
center ( "1 = select port 1           L = toggle word LENGTH  " );
center ( "2 = select port 2           N = set parity to NONE  " );
center ( "B = set BAUD rate           O = set parity to ODD   " );
center ( "E = set parity to EVEN      R = toggle error REPORTS" );
center ( "F = toggle FLOW control     S = toggle STOP bits    " );
center ( "I = INITIALIZE ints, etc.   Q = QUIT                " );
         continue;

        case 'l' :                  /* toggle word length         */
          iobf [ 0 ] ^= 1;
          iocwr ();
          break;

```

Figure 6-4. Continued. *(more)*

```
        case 'n' :                  /* set parity off             */
          iobf [ 0 ] &=~ ( m_PG + m_PE );
          iocwr ();
          break;

        case 'o' :                  /* set parity odd             */
          iobf [ 0 ] |= m_PG;
          iobf [ 0 ] &=~ m_PE;
          iocwr ();
          break;

        case 'r' :                  /* toggle error reports       */
          iobf [ 2 ] ^= 1;
          iocwr ();
          break;

        case 's' :                  /* toggle stop bits           */
          iobf [ 0 ] ^= m_XS;
          iocwr ();
          break;

        case 'q' :
          fclose ( dvp );
          exit ( 0 );               /* break the loop, get out    */
        }
      cputs ( CLS );                /* clear the display          */
      center ( "CURRENT COMDVR STATUS" );
      report ( u, dvp );            /* report current status      */
    }
}

center ( s ) char * s;             /* centers a string on CRT    */
{ int i ;
  for ( i = 80 - strlen ( s ); i > 0; i -= 2 )
    putch ( ' ' );
  cputs ( s );
  cputs ( "\r\n" );
}

iocwr ()                           /* IOCTL Write to COMDVR      */
{ rvs . x . ax = 0x4403;
  rvs . x . bx = fileno ( dvp );
  rvs . x . cx = 10;
  rvs . x . dx = ( int ) iobf;
  intdos ( & rvs, & rvs );
}

char * onoff ( x ) int x ;
{ return ( x ? " ON" : " OFF" );
}

```

Figure 6-4. Continued. *(more)*

```
report ( unit ) int unit ;
{ char temp [ 80 ];
  rvs . x . ax = 0x4402;
  rvs . x . bx = fileno ( dvp );
  rvs . x . cx = 10;
  rvs . x . dx = ( int ) iobf;
  intdos ( & rvs, & rvs );         /* use IOCTL Read to get data */
  sprintf ( temp, "\nDevice ASY%d\t%d BPS, %d-c-%c\r\n\n",
           unit, iobf [ 4 ],          /* baud rate                */
           5 + ( iobf [ 0 ] & m_WL ), /* word length              */
           ( iobf [ 0 ] & m_PG ?
             ( iobf [ 0 ] & m_PE ? 'E' : 'O' ) : 'N' ),
           ( iobf [ 0 ] & m_XS ? '2' : '1' )); /* stop bits      */
  cputs ( temp );

  cputs ( "Hardware Interrupts are" );
  cputs ( onoff ( iobf [ 0 ] & HWINT ));
  cputs ( ", Data Terminal Rdy" );
  cputs ( onoff ( iobf [ 0 ] & o_DTR ));
  cputs ( ", Rqst To Send" );
  cputs ( onoff ( iobf [ 0 ] & o_RTS ));
  cputs ( ".\r\n" );

  cputs ( "Carrier Detect" );
  cputs ( onoff ( iobf [ 1 ] & i_CD ));
  cputs ( ", Data Set Rdy" );
  cputs ( onoff ( iobf [ 1 ] & i_DSR ));
  cputs ( ", Clear to Send" );
  cputs ( onoff ( iobf [ 1 ] & i_CTS ));
  cputs ( ", Ring Indicator" );
  cputs ( onoff ( iobf [ 1 ] & i_RI ));
  cputs ( ".\r\n" );

  cputs ( l_SRE & iobf [ 1 ] ? "Xmtr SR Empty, " : "" );
  cputs ( l_HRE & iobf [ 1 ] ? "Xmtr HR Empty, " : "" );
  cputs ( l_BRK & iobf [ 1 ] ? "Break Received, " : "" );
  cputs ( l_ER1 & iobf [ 1 ] ? "Framing Error, " : "" );
  cputs ( l_ER2 & iobf [ 1 ] ? "Parity Error, " : "" );
  cputs ( l_ER3 & iobf [ 1 ] ? "Overrun Error, " : "" );
  cputs ( l_RRF & iobf [ 1 ] ? "Rcvr DR Full, " : "" );
  cputs ( "\b\b.\r\n" );

  cputs ( "Reception errors " );
  if ( iobf [ 2 ] == 1 )
    cputs ( "are encoded as graphics in buffer" );
  else
    cputs ( "set failure flag" );
  cputs ( ".\r\n" );

  cputs ( "Outgoing Flow Control " );
  if ( iobf [ 3 ] & 4 )
```

Figure 6-4. Continued. *(more)*

```
223 :     cuts ( "by XON and XOFF" );
224 :   else
225 :     if ( iobf [ 3 ] & 2 )
226 :       cputs ( "by RTS and CTS" );
227 :   else
228 :     if ( iobf [ 3 ] & 1 )
229 :       cputs ( "by DTR and DSR" );
230 :   else
231 :     cputs ( "disabled" );
232 :   cputs ( ".\r\n" );
233 : }
234 :
235 : /*end of cdvutl.c */
```

Figure 6-4. Continued.

Although CDVUTL appears complicated, most of the complexity is concentrated in the routines that map driver bit settings into on-screen display text. Each such mapping requires several lines of source code to generate only a few words of the display report. Table 6-10 summarizes the functions found in this program.

Table 6-10. CDVUTL Program Functions.

Lines	Name	Description
42–45	*main()*	Conventional entry point.
47–150	*disp()*	Main dispatching loop.
152–158	*center()*	Centers text on CRT.
160–166	*iocwr()*	Writes control string to driver with IOCTL Write.
168–170	*onoff()*	Returns pointer to ON or OFF.
172–233	*report()*	Reads driver status and reports it on display.

The long list of *#define* operations at the start of the listing (lines 11 through 33) helps make the bitmapping comprehensible by assigning a symbolic name to each significant bit in the four UART registers.

The *main()* procedure of CDVUTL displays a banner line and then calls the dispatcher routine, *disp()*, to start operation. CDVUTL makes no use of either command-line parameters or the environment, so the usual argument declarations are omitted.

Upon entry to *disp()*, the first action is to establish the default driver as *ASY1* by setting *u = 1* and opening *ASY1* (line 50); the program then enters an apparent infinite loop (lines 51 through 149).

With each repetition, the loop first prompts for a command (line 52) and then gets the next keystroke and uses it to control a huge *switch()* statement (lines 53 through 145). If no case matches the key pressed, the *switch()* statement does nothing; the program simply displays a report of all current conditions at the selected driver (lines 146 through 148) and then closes the loop back to issue a new prompt and get another keystroke.

However, if the key pressed matches one of the cases in the *switch()* statement, the corresponding command is executed. The digits *1* (line 55) and *2* (line 61) select the driver to be affected. The *?* key (line 105) causes the list of valid command keys to be displayed. The *q* key (line 142) causes the program to terminate by calling *exit(0)* and is the only exit from the infinite loop. The other valid keys all change one or more bits in the IOCTL control string to modify corresponding attributes of the driver and then send the string to the driver by using the MS-DOS IOCTL Write function (Interrupt 21H Function 44H Subfunction 03H) via function *iocwr()* (lines 160 through 166).

After the command is executed (except for the *q* command, which terminates operation of CDVUTL and returns to MS-DOS command level, and the *?* command, which displays the command list), the *report()* function (lines 172 through 233) is called (at line 148) to display all of the driver's attributes, including those just changed. This function issues an IOCTL Read command (Interrupt 21H Function 44H Subfunction 02H, in lines 174 through 178) to get new status information into the control string and then uses a sequence of bit filtering (lines 179 through 232) to translate the obtained status information into words for display.

The special console I/O routines provided in Microsoft C libraries have been used extensively in this routine. Other compilers may require changes in the names of such library routines as *getch* or *dosint* as well as in the names of *#include* files (lines 6 through 9).

Each of the actual command sequences changes only a few bits in one of the 10 bytes of the command string and then writes the string to the driver. A full-featured communications program might make several changes at one time — for example, switching from 7-bit, even parity, XON/XOFF flow control to 8-bit, no parity, without flow control to prevent losing any bytes with values of 11H or 13H while performing a binary file transfer with error-correcting protocol. In such a case, the program could make all required changes to the control string before issuing a single IOCTL Write to put them into effect.

The Traditional Approach

Because the necessary device driver has never been a part of MS-DOS, most communications programs are written to provide and install their own port driver code and remove it before returning to MS-DOS. The second sample program package in this article illustrates this approach. Although the major part of the package is written in Microsoft C, three assembly-language modules are required to provide the hardware interrupt service routines, the exception handler, and faster video display. They are discussed first.

The hardware ISR module

The first module is a handler to service UART interrupts. Code for this handler, including routines to install it at entry and remove it on exit, appears in CH1.ASM, shown in Figure 6-5.

```
        TITLE   CH1.ASM

; CH1.ASM -- support file for CTERM.C terminal emulator
;       set up to work with COM2
;       for use with Microsoft C and SMALL model only...

_TEXT   segment byte public 'CODE'
_TEXT   ends
_DATA   segment byte public 'DATA'
_DATA   ends
CONST   segment byte public 'CONST'
CONST   ends
_BSS    segment byte public 'BSS'
_BSS    ends

DGROUP  GROUP   CONST, _BSS, _DATA
        assume  cs:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP

_TEXT   segment

        public  _i_m,_rdmdm,_Send_Byte,_wrtmdm,_set_mdm,_u_m

bport   EQU     02F8h             ; COM2 base address, use 03F8H for COM1
getiv   EQU     350Bh             ; COM2 vectors, use 0CH for COM1
putiv   EQU     250Bh
imrmsk  EQU     00001000b         ; COM2 mask, use 00000100b for COM1
oiv_o   DW      0                 ; old int vector save space
oiv_s   DW      0

bf_pp   DW      in_bf             ; put pointer (last used)
bf_gp   DW      in_bf             ; get pointer (next to use)
bf_bg   DW      in_bf             ; start of buffer
bf_fi   DW      b_last            ; end of buffer

in_bf   DB      512 DUP (?)       ; input buffer

b_last  EQU     $                 ; address just past buffer end

bd_dv   DW      0417h             ; baud rate divisors (0=110 bps)
        DW      0300h             ; code 1 =  150 bps
        DW      0180h             ; code 2 =  300 bps
        DW      00C0h             ; code 3 =  600 bps
        DW      0060h             ; code 4 = 1200 bps
        DW      0030h             ; code 5 = 2400 bps
        DW      0018h             ; code 6 = 4800 bps
        DW      000Ch             ; code 7 = 9600 bps

_set_mdm proc   near              ; replaces BIOS 'init' function
        PUSH    BP
        MOV     BP,SP             ; establish stackframe pointer
        PUSH    ES                ; save registers
```

Figure 6-5. CH1.ASM *(more)*

```
        PUSH    DS
        MOV     AX,CS           ; point them to CODE segment
        MOV     DS,AX
        MOV     ES,AX
        MOV     AH,[BP+4]       ; get parameter passed by C
        MOV     DX,BPORT+3      ; point to Line Control Reg
        MOV     AL,80h          ; set DLAB bit (see text)
        OUT     DX,AL
        MOV     DL,AH           ; shift param to BAUD field
        MOV     CL,4
        ROL     DL,CL
        AND     DX,00001110b    ; mask out all other bits
        MOV     DI,OFFSET bd_dv
        ADD     DI,DX           ; make pointer to true divisor
        MOV     DX,BPORT+1      ; set to high byte first
        MOV     AL,[DI+1]
        OUT     DX,AL           ; put high byte into UART
        MOV     DX,BPORT        ; then to low byte
        MOV     AL,[DI]
        OUT     DX,AL
        MOV     AL,AH           ; now use rest of parameter
        AND     AL,00011111b    ; to set Line Control Reg
        MOV     DX,BPORT+3
        OUT     DX,AL
        MOV     DX,BPORT+2      ; Interrupt Enable Register
        MOV     AL,1            ; Receive type only
        OUT     DX,AL
        POP     DS              ; restore saved registers
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_set_mdm endp

_wrtmdm proc    near            ; write char to modem
_Send_Byte:                     ; name used by main program
        PUSH    BP
        MOV     BP,SP           ; set up pointer and save regs
        PUSH    ES
        PUSH    DS
        MOV     AX,CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     DX,BPORT+4      ; establish DTR, RTS, and OUT2
        MOV     AL,0Bh
        OUT     DX,AL
        MOV     DX,BPORT+6      ; check for on line, CTS
        MOV     BH,30h
        CALL    w_tmr
        JNZ     w_out           ; timed out
        MOV     DX,BPORT+5      ; check for UART ready
```

Figure 6-5. Continued. *(more)*

```
        MOV     BH,20h
        CALL    w_tmr
        JNZ     w_out           ; timed out
        MOV     DX,BPORT        ; send out to UART port
        MOV     AL,[BP+4]       ; get char passed from C
        OUT     DX,AL
w_out:  POP     DS              ; restore saved regs
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_wrtmdm endp

_rdmdm  proc    near            ; reads byte from buffer
        PUSH    BP
        MOV     BP,SP           ; set up ptr, save regs
        PUSH    ES
        PUSH    DS
        MOV     AX,CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     AX,0FFFFh       ; set for EOF flag
        MOV     BX,bf_gp        ; use "get" ptr
        CMP     BX,bf_pp        ; compare to "put"
        JZ      nochr           ; same, empty
        INC     BX              ; else char available
        CMP     BX,bf_fi        ; at end of bfr?
        JNZ     noend           ; no
        MOV     BX,bf_bg        ; yes, set to beg
noend:  MOV     AL,[BX]         ; get the char
        MOV     bf_gp,BX        ; update "get" ptr
        INC     AH              ; zero AH as flag
nochr:  POP     DS              ; restore regs
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_rdmdm  endp

w_tmr   proc    near
        MOV     BL,1            ; wait timer, double loop
w_tm1:  SUB     CX,CX           ; set up inner loop
w_tm2:  IN      AL,DX           ; check for requested response
        MOV     AH,AL           ; save what came in
        AND     AL,BH           ; mask with desired bits
        CMP     AL,BH           ; then compare
        JZ      w_tm3           ; got it, return with ZF set
        LOOP    w_tm2           ; else keep trying
        DEC     BL              ; until double loop expires
        JNZ     w_tm1
        OR      BH,BH           ; timed out, return NZ
```

Figure 6-5. Continued. *(more)*

```
w_tm3:  RET
w_tmr   endp

; hardware interrupt service routine
rts_m:  CLI
        PUSH    DS              ; save all regs
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    CS              ; set DS same as CS
        POP     DS
        MOV     DX,BPORT        ; grab the char from UART
        IN      AL,DX
        MOV     BX,bf_pp        ; use "put" ptr
        INC     BX              ; step to next slot
        CMP     BX,bf_fi        ; past end yet?
        JNZ     nofix           ; no
        MOV     BX,bf_bg        ; yes, set to begin
nofix:  MOV     [BX],AL         ; put char in buffer
        MOV     bf_pp,BX        ; update "put" ptr
        MOV     AL,20h          ; send EOI to 8259 chip
        OUT     20h,AL
        POP     DX              ; restore regs
        POP     CX
        POP     BX
        POP     AX
        POP     DS
        IRET

_i_m    proc    near            ; install modem service
        PUSH    BP
        MOV     BP,SP           ; save all regs used
        PUSH    ES
        PUSH    DS
        MOV     AX,CS           ; set DS,ES=CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     DX,BPORT+1      ; Interrupt Enable Reg
        MOV     AL,0Fh          ; enable all ints now
        OUT     DX,AL

im1:    MOV     DX,BPORT+2      ; clear junk from UART
        IN      AL,DX           ; read IID reg of UART
        MOV     AH,AL           ; save what came in
        TEST    AL,1            ; anything pending?
        JNZ     im5             ; no, all clear now
        CMP     AH,0            ; yes, Modem Status?
        JNZ     im2             ; no
        MOV     DX,BPORT+6      ; yes, read MSR to clear
        IN      AL,DX
```

Figure 6-5. Continued. *(more)*

```
im2:    CMP     AH,2            ; Transmit HR empty?
        JNZ     im3             ; no (no action needed)
im3:    CMP     AH,4            ; Received Data Ready?
        JNZ     im4             ; no
        MOV     DX,BPORT        ; yes, read it to clear
        IN      AL,DX
im4:    CMP     AH,6            ; Line Status?
        JNZ     im1             ; no, check for more
        MOV     DX,BPORT+5      ; yes, read LSR to clear
        IN      AL,DX
        JMP     im1             ; then check for more

im5:    MOV     DX,BPORT+4      ; set up working conditions
        MOV     AL,0Bh          ; DTR, RTS, OUT2 bits
        OUT     DX,AL
        MOV     AL,1            ; enable RCV interrupt only
        MOV     DX,BPORT+1
        OUT     DX,AL
        MOV     AX,GETIV        ; get old int vector
        INT     21h
        MOV     oiv_o,BX        ; save for restoring later
        MOV     oiv_s,ES
        MOV     DX,OFFSET rts_m ; set in new one
        MOV     AX,PUTIV
        INT     21h
        IN      AL,21h          ; now enable 8259 PIC
        AND     AL,NOT IMRMSK
        OUT     21h,AL
        MOV     AL,20h          ; then send out an EOI
        OUT     20h,AL
        POP     DS              ; restore regs
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_i_m    endp

_u_m    proc    near            ; uninstall modem service
        PUSH    BP
        MOV     BP,SP           ; save registers
        IN      AL,21h          ; disable COM int in 8259
        OR      AL,IMRMSK
        OUT     21h,AL
        PUSH    ES
        PUSH    DS
        MOV     AX,CS           ; set same as CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     AL,0            ; disable UART ints
        MOV     DX,BPORT+1
        OUT     DX,AL
```

Figure 6-5. Continued. *(more)*

```
256 :          MOV     DX,oiv_o        ; restore original vector
257 :          MOV     DS,oiv_s
258 :          MOV     AX,PUTIV
259 :          INT     21h
260 :          POP     DS              ; restore registers
261 :          POP     ES
262 :          MOV     SP,BP
263 :          POP     BP
264 :          RET
265 : _u_m     endp
266 :
267 : _TEXT    ends
268 :
269 :          END
```

Figure 6-5. Continued.

The routines in CH1 are set up to work only with port COM2; to use them with COM1, the three symbolic constants BPORT (base address), GETIV, and PUTIV must be changed to match the COM1 values. Also, as presented, this code is for use with the Microsoft C small memory model only; for use with other memory models, the C compiler manuals should be consulted for making the necessary changes. *See also* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program.

The parts of CH1 are listed in Table 6-11, as they occur in the listing. The leading underscore that is part of the name for each of the six functions is supplied by the C compiler; within the C program that calls the function, the underscore is omitted.

Table 6-11. CH1 Module Functions.

Lines	Name	Description
1–26		Administrative details.
27–46		Data areas.
48–84	*_set_mdm*	Initializes UART as specified by parameter passed from C.
86–114	*_wrtmdm*	Outputs character to UART.
87	*_Send_Byte*	Entry point for use if flow control is added to system.
116–140	*_rdmdm*	Gets character from buffer where ISR put it, or signals that no character available.
142–155	*w_tmr*	Wait timer; internal routine used to prevent infinite wait in case of problems.
157–182	*rts_m*	Hardware ISR; installed by *_i_m* and removed by *_u_m*.
184–240	*_i_m*	Installs ISR, saving old interrupt vector.
242–265	*_u_m*	Uninstalls ISR, restoring saved interrupt vector.

For simplest operation, the ISR used in this example (unlike the device driver) services *only* the received-data interrupt; the other three types of IRQ are disabled at the UART. Each time a byte is received by the UART, the ISR puts it into the buffer. The *_rdmdm* code, when called by the C program, gets a byte from the buffer if one is available. If not, *_rdmdm* returns the C EOF code (–1) to indicate that no byte can be obtained.

To send a byte, the C program can call either *_Send_Byte* or *_wrtmdm*; in the package as shown, these are alternative names for the same routine. In the more complex program from which this package was adapted, *_Send_Byte* is called when flow control is desired and the flow-control routine calls *_wrtmdm.* To implement flow control, line 87 should be deleted from CH1.ASM and a control function named *Send_Byte()* should be added to the main C program. Flow-control tests must occur in *Send_Byte()*; *_wrtmdm* performs the actual port interfacing.

To set the modem baud rate, word length, and parity, *_set_mdm* is called from the C program, with a setup parameter passed as an argument. The format of this parameter is shown in Table 6-12 and is identical to the IBM BIOS Interrupt 14H Function 00H (Initialization).

Table 6-12. *set_mdm()* Parameter Coding.

Binary	Meaning
000xxxxx	Set to 110 bps
001xxxxx	Set to 150 bps
010xxxxx	Set to 300 bps
011xxxxx	Set to 600 bps
100xxxxx	Set to 1200 bps
101xxxxx	Set to 2400 bps
110xxxxx	Set to 4800 bps
111xxxxx	Set to 9600 bps
xxxx0xxx	No parity
xxx01xxx	ODD Parity
xxx11xxx	EVEN Parity
xxxxx0xx	1 stop bit
xxxxx1xx	2 stop bits (1.5 if WL = 5)
xxxxxx00	Word length = 5
xxxxxx01	Word length = 6
xxxxxx10	Word length = 7
xxxxxx11	Word length = 8

The CH1 code provides a 512-byte ring buffer for incoming data; the buffer size should be adequate for reception at speeds up to 2400 bps without loss of data during scrolling.

The exception-handler module

For the ISR handler of CH1 to be usable, an exception handler is needed to prevent return of control to MS-DOS before *_u_m* restores the ISR vector to its original value. If a program using this code returns to MS-DOS without calling *_u_m*, the system is virtually certain to crash when line noise causes a received-data interrupt and the ISR code is no longer in memory.

A replacement exception handler (CH1A.ASM), including routines for installation, access, and removal, is shown in Figure 6-6. Like the ISR, this module is designed to work with Microsoft C (again, the small memory model only).

Note: This module does not provide for fatal disk errors; if one occurs, immediate restarting is necessary. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Exception Handlers.

```
        TITLE   CH1A.ASM

; CH1A.ASM -- support file for CTERM.C terminal emulator
;       this set of routines replaces Ctrl-C/Ctrl-BREAK
;       usage: void set_int(), rst_int();
;               int broke();    /* boolean if BREAK     */
;       for use with Microsoft C and SMALL model only...

_TEXT   segment byte public 'CODE'
_TEXT   ends
_DATA   segment byte public 'DATA'
_DATA   ends
CONST   segment byte public 'CONST'
CONST   ends
_BSS    segment byte public 'BSS'
_BSS    ends

DGROUP  GROUP   CONST, _BSS, _DATA
        ASSUME  CS:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP

_DATA   SEGMENT BYTE PUBLIC 'DATA'

OLDINT1B DD     0               ; storage for original INT 1BH vector

_DATA   ENDS

_TEXT   SEGMENT

        PUBLIC  _set_int,_rst_int,_broke

myint1b:
        mov     word ptr cs:brkflg,1Bh          ; make it nonzero
        iret
```

Figure 6-6. CH1A.ASM. *(more)*

```

myint23:
        mov     word ptr cs:brkflg,23h          ; make it nonzero
        iret

brkflg  dw      0               ; flag that BREAK occurred

_broke  proc    near            ; returns 0 if no break
        xor     ax,ax           ; prepare to reset flag
        xchg    ax,cs:brkflg    ; return current flag value
        ret
_broke  endp

_set_int proc near
        mov     ax,351bh        ; get interrupt vector for 1BH
        int     21h             ; (don't need to save for 23H)
        mov     word ptr oldint1b,bx     ; save offset in first word
        mov     word ptr oldint1b+2,es   ; save segment in second word

        push    ds              ; save our data segment
        mov     ax,cs           ; set DS to CS for now
        mov     ds,ax
        lea     dx,myint1b      ; DS:DX points to new routine
        mov     ax,251bh        ; set interrupt vector
        int     21h
        mov     ax,cs           ; set DS to CS for now
        mov     ds,ax
        lea     dx,myint23      ; DS:DX points to new routine
        mov     ax,2523h        ; set interrupt vector
        int     21h
        pop     ds              ; restore data segment
        ret
_set_int endp

_rst_int proc near
        push    ds              ; save our data segment
        lds     dx,oldint1b     ; DS:DX points to original
        mov     ax,251bh        ; set interrupt vector
        int     21h
        pop     ds              ; restore data segment
        ret
_rst_int endp

_TEXT   ends

        END
```

Figure 6-6. Continued.

The three functions in CH1A are *_set_int*, which saves the old vector value for Interrupt 1BH (ROM BIOS Control-Break) and then resets both that vector and the one for Interrupt 23H (Control-C Handler Address) to internal ISR code; *_rst_int*, which restores the original value for the Interrupt 1BH vector; and *_broke*, which returns the present value of an internal flag (and always clears the flag, just in case it had been set). The internal flag is set to a nonzero value in response to either of the revectored interrupts and is tested from the main C program via the *_broke* function.

The video display module

The final assembly-language module (CH2.ASM) used by the second package is shown in Figure 6-7. This module provides convenient screen clearing and cursor positioning via direct calls to the IBM BIOS, but this can be eliminated with minor rewriting of the 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.

```
        TITLE   CH2.ASM

; CH2.ASM -- support file for CTERM.C terminal emulator
;       for use with Microsoft C and SMALL model only...

_TEXT   segment byte public 'CODE'
_TEXT   ends
_DATA   segment byte public 'DATA'
_DATA   ends
CONST   segment byte public 'CONST'
CONST   ends
_BSS    segment byte public 'BSS'
_BSS    ends

DGROUP  GROUP   CONST, _BSS, _DATA
        assume  CS:_TEXT, DS:DGROUP, ES:DGROUP, SS:DGROUP

_TEXT   segment

        public  __cls,__color,__deol,__i_v,__key,__wrchr,__wrpos

atrib   DB      0               ; attribute
_colr   DB      0               ; color
v_bas   DW      0               ; video segment
v_ulc   DW      0               ; upper left corner cursor
v_lrc   DW      184Fh           ; lower right corner cursor
v_col   DW      0               ; current col/row

__key   proc    near            ; get keystroke
        PUSH    BP
        MOV     AH,1            ; check status via BIOS
        INT     16h
        MOV     AX,0FFFFh
        JZ      key00           ; none ready, return EOF
        MOV     AH,0            ; have one, read via BIOS
```

Figure 6-7. CH2.ASM. *(more)*

```
        INT     16h
key00:  POP     BP
        RET
__key   endp

__wrchr proc    near
        PUSH    BP
        MOV     BP,SP
        MOV     AL,[BP+4]       ; get char passed by C
        CMP     AL,' '
        JNB     prchr           ; printing char, go do it
        CMP     AL,8
        JNZ     notbs
        DEC     BYTE PTR v_col  ; process backspace
        MOV     AL,byte ptr v_col
        CMP     AL,byte ptr v_ulc
        JB      nxt_c           ; step to next column
        JMP     norml

notbs:  CMP     AL,9
        JNZ     notht
        MOV     AL,byte ptr v_col       ; process HTAB
        ADD     AL,8
        AND     AL,0F8h
        MOV     byte ptr v_col,AL
        CMP     AL,byte ptr v_lrc
        JA      nxt_c
        JMP     SHORT   norml

notht:  CMP     AL,0Ah
        JNZ     notlf
        MOV     AL,byte ptr v_col+1     ; process linefeed
        INC     AL
        CMP     AL,byte ptr v_lrc+1
        JBE     noht1
        CALL    scrol
        MOV     AL,byte ptr v_lrc+1
noht1:  MOV     byte ptr v_col+1,AL
        JMP     SHORT   norml

notlf:  CMP     AL,0Ch
        JNZ     ck_cr
        CALL    __cls           ; process formfeed
        JMP     SHORT   ignor

ck_cr:  CMP     AL,0Dh
        JNZ     ignor           ; ignore all other CTL chars
        MOV     AL,byte ptr v_ulc       ; process CR
        MOV     byte ptr v_col,AL
        JMP     SHORT   norml

```

Figure 6-7. Continued. *(more)*

```
prchr:  MOV     AH,_colr        ; process printing char
        PUSH    AX
        XOR     AH,AH
        MOV     AL,byte ptr v_col+1
        PUSH    AX
        MOV     AL,byte ptr v_col
        PUSH    AX
        CALL    wrtvr
        MOV     SP,BP
nxt_c:  INC     BYTE PTR v_col  ; advance to next column
        MOV     AL,byte ptr v_col
        CMP     AL,byte ptr v_lrc
        JLE     norml
        MOV     AL,0Dh          ; went off end, do CR/LF
        PUSH    AX
        CALL    _wrchr
        POP     AX
        MOV     AL,0Ah
        PUSH    AX
        CALL    _wrchr
        POP     AX
norml:  CALL    set_cur
ignor:  MOV     SP,BP
        POP     BP
        RET
_wrchr  endp

_i_v    proc    near            ; establish video base segment
        PUSH    BP
        MOV     BP,SP
        MOV     AX,0B000h       ; mono, B800 for CGA
        MOV     v_bas,AX        ; could be made automatic
        MOV     SP,BP
        POP     BP
        RET
_i_v    endp

_wrpos  proc    near            ; set cursor position
        PUSH    BP
        MOV     BP,SP
        MOV     DH,[BP+4]       ; row from C program
        MOV     DL,[BP+6]       ; col from C program
        MOV     v_col,DX        ; cursor position
        MOV     BH,atrib        ; attribute
        MOV     AH,2
        PUSH    BP
        INT     10h
        POP     BP
        MOV     AX,v_col        ; return cursor position
        MOV     SP,BP
        POP     BP
```

Figure 6-7. Continued. *(more)*

```
        RET
__wrpos endp

set_cur proc    near            ; set cursor to v_col
        PUSH    BP
        MOV     BP,SP
        MOV     DX,v_col        ; use where v_col says
        MOV     BH,atrib
        MOV     AH,2
        PUSH    BP
        INT     10h
        POP     BP
        MOV     AX,v_col
        MOV     SP,BP
        POP     BP
        RET
set_cur endp

__color proc    near            ; _color(fg, bg)
        PUSH    BP
        MOV     BP,SP
        MOV     AH,[BP+6]       ; background from C
        MOV     AL,[BP+4]       ; foreground from C
        MOV     CX,4
        SHL     AH,CL
        AND     AL,0Fh
        OR      AL,AH           ; pack up into 1 byte
        MOV     _colr,AL        ; store for handler's use
        XOR     AH,AH
        MOV     SP,BP
        POP     BP
        RET
__color endp

scrol   proc    near            ; scroll CRT up by one line
        PUSH    BP
        MOV     BP,SP
        MOV     AL,1            ; count of lines to scroll
        MOV     CX,v_ulc
        MOV     DX,v_lrc
        MOV     BH,_colr
        MOV     AH,6
        PUSH    BP
        INT     10h             ; use BIOS
        POP     BP
        MOV     SP,BP
        POP     BP
        RET
scrol   endp

__cls   proc    near            ; clear CRT
```

Figure 6-7. Continued. *(more)*

```
        PUSH    BP
        MOV     BP,SP
        MOV     AL,0            ; flags CLS to BIOS
        MOV     CX,v_ulc
        MOV     v_col,CX        ; set to HOME
        MOV     DX,v_lrc
        MOV     BH,_colr
        MOV     AH,6
        PUSH    BP
        INT     10h             ; use BIOS scroll up
        POP     BP
        CALL    set_cur         ; cursor to HOME
        MOV     SP,BP
        POP     BP
        RET
__cls   endp

__deol  proc    near            ; delete to end of line
        PUSH    BP
        MOV     BP,SP
        MOV     AL,' '
        MOV     AH,_colr        ; set up blanks
        PUSH    AX
        MOV     AL,byte ptr v_col+1
        XOR     AH,AH           ; set up row value
        PUSH    AX
        MOV     AL,byte ptr v_col

deol1:  CMP     AL,byte ptr v_lrc
        JA      deol2           ; at RH edge
        PUSH    AX              ; current location
        CALL    wrtvr           ; write a blank
        POP     AX
        INC     AL              ; next column
        JMP     deol1           ; do it again

deol2:  MOV     AX,v_col        ; return cursor position
        MOV     SP,BP
        POP     BP
        RET
__deol  endp

wrtvr   proc    near            ; write video RAM (col, row, char/atr)
        PUSH    BP
        MOV     BP,SP           ; set up arg ptr
        MOV     DL,[BP+4]       ; column
        MOV     DH,[BP+6]       ; row
        MOV     BX,[BP+8]       ; char/atr
        MOV     AL,80           ; calc offset
        MUL     DH
        XOR     DH,DH
```

Figure 6-7. Continued. *(more)*

```
        ADD     AX,DX
        ADD     AX,AX           ; adjust bytes to words
        PUSH    ES              ; save seg reg
        MOV     DI,AX
        MOV     AX,v_bas        ; set up segment
        MOV     ES,AX
        MOV     AX,BX           ; get the data
        STOSW                   ; put on screen
        POP     ES              ; restore regs
        MOV     SP,BP
        POP     BP
        RET
wrtvr   endp

_TEXT   ends

        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.

```
/* Terminal Emulator    (cterm.c)
 *      Jim Kyle, 1987
 *
 *      Uses files CH1, CH1A, and CH2 for MASM support...
 */

#include <stdio.h>
#include <conio.h>                      /* special console i/o      */
#include <stdlib.h>                     /* misc definitions         */
#include <dos.h>                        /* defines intdos()         */
#include <string.h>
#define BRK 'C'-'@'                     /* control characters       */
#define ESC '['-'@'
#define XON 'Q'-'@'
#define XOFF 'S'-'@'

#define True  1
#define False  0

#define Is_Function_Key(C) ( (C) == ESC )

static char capbfr [ 4096 ];            /* capture buffer           */
static int wh,
     ws;
```

Figure 6-8. CTERM.C. *(more)*

```

static int I,
    waitchr = 0,
    vflag = False,
    capbp,
    capbc,
    Ch,
    Want_7_Bit = True,
    ESC_Seq_State = 0;        /* escape sequence state variable     */

int _cx ,
     _cy,
     _atr = 0x07,                       /* white on black           */
     _pag = 0,
     oldtop = 0,
     oldbot = 0x184f;

FILE * in_file = NULL;         /* start with keyboard input         */
FILE * cap_file = NULL;

#include "cterm.h"             /* external declarations, etc.       */

int Wants_To_Abort ()          /* checks for interrupt of script    */
{ return broke ();
}
void

main ( argc, argv ) int argc ; /* main routine                      */
 char * argv [];
{ char * cp,
     * addext ();
  if ( argc > 1 )                /* check for script filename       */
    in_file = fopen ( addext ( argv [ 1 ], ".SCR" ), "r" );
  if ( argc > 2 )                /* check for capture filename      */
    cap_file = fopen ( addext ( argv [ 2 ], ".CAP" ), "w" );
  set_int ();                    /* install CH1 module              */
  Set_Vid ();                    /* get video setup                 */
  cls ();                        /* clear the screen                */
  cputs ( "Terminal Emulator" ); /* tell who's working              */
  cputs ( "\r\n< ESC for local commands >\r\n\n" );
  Want_7_Bit = True;
  ESC_Seq_State = 0;
  Init_Comm ();                  /* set up drivers, etc.            */
  while ( 1 )                           /* main loop                */
    { if (( Ch = kb_file ()) > 0 )      /* check local              */
        { if ( Is_Function_Key ( Ch ))
            { if ( docmd () < 0 )       /* command                  */
                break;
            }
          else
            Send_Byte ( Ch & 0x7F );    /* else send it             */
```

Figure 6-8. Continued. *(more)*

```
  }
if (( Ch = Read_Modem ()) >= 0 )   /* check remote             */
  { if ( Want_7_Bit )
      Ch &= 0x7F;                   /* trim off high bit        */
    switch ( ESC_Seq_State )       /* state machine            */
      {
      case 0 :                     /* no Esc sequence          */
        switch ( Ch )
          {
          case ESC   :             /* Esc char received        */
            ESC_Seq_State = 1;
            break;

          default :
            if ( Ch == waitchr )   /* wait if required         */
              waitchr = 0;
            if ( Ch == 12 )        /* clear screen on FF       */
              cls ();
            else
              if ( Ch != 127 )     /* ignore rubouts           */
                { putchx ( (char) Ch );  /* handle all others */
                  put_cap ( (char) Ch );
                }
          }
        break;

      case 1 : /* ESC -- process any escape sequences here    */
        switch ( Ch )
          {
          case 'A' :               /* VT52 up                  */
            ;                      /* nothing but stubs here   */
            ESC_Seq_State = 0;
            break;

          case 'B' :               /* VT52 down                */
            ;
            ESC_Seq_State = 0;
            break;

          case 'C' :               /* VT52 left                */
            ;
            ESC_Seq_State = 0;
            break;

          case 'D' :               /* VT52 right               */
            ;
            ESC_Seq_State = 0;
            break;

          case 'E' :               /* VT52 Erase CRT           */
            cls ();                /* actually do this one     */
```

Figure 6-8. Continued. *(more)*

```
                  ESC_Seq_State = 0;
                  break;

                case 'H' :              /* VT52 home cursor        */
                  locate ( 0, 0 );
                  ESC_Seq_State = 0;
                  break;

                case 'j' :              /* VT52 Erase to EOS       */
                  deos ();
                  ESC_Seq_State = 0;
                  break;

                case '[' :      /* ANSI.SYS - VT100 sequence       */
                  ESC_Seq_State = 2;
                  break;

                default :
                  putchx ( ESC );       /* pass thru all others    */
                  putchx ( (char) Ch );
                  ESC_Seq_State = 0;
                }
              break;

            case 2 :                    /* ANSI 3.64 decoder       */
              ESC_Seq_State = 0;        /* not implemented         */
            }
        }
      if ( broke ())                    /* check CH1A handlers     */
        { cputs ( "\r\n***BREAK***\r\n" );
          break;
        }
    }                                   /* end of main loop        */
  if ( cap_file )                       /* save any capture        */
    cap_flush ();
  Term_Comm ();                         /* restore when done       */
  rst_int ();                           /* restore break handlers  */
  exit ( 0 );                           /* be nice to MS-DOS       */
}

docmd ()                                /* local command shell     */
{ FILE * getfil ();
  int wp;
  wp = True;
  if ( ! in_file || vflag )
    cputs ( "\r\n\tCommand: " );        /* ask for command         */
  else
    wp = False;
  Ch = toupper ( kbd_wait ());          /* get response            */
  if ( wp )
    putchx ( (char) Ch );
```

Figure 6-8. Continued. *(more)*

```
switch  ( Ch )                     /* and act on it              */
 {
 case 'S' :
  if ( wp )
   cputs ( "low speed\r\n" );
  Set_Baud ( 300 );
  break;

 case 'D' :
  if ( wp )
   cputs ( "elay (1-9 sec): " );
  Ch = kbd_wait ();
  if ( wp )
   putchx ( (char) Ch );
  Delay ( 1000 * ( Ch - '0' ));
  if ( wp )
   putchx ( '\n' );
  break;

 case 'E' :
  if ( wp )
   cputs ( "ven Parity\r\n" );
  Set_Parity ( 2 );
  break;

 case 'F' :
  if ( wp )
   cputs ( "ast speed\r\n" );
  Set_Baud ( 1200 );
  break;

 case 'H' :
  if ( wp )
   { cputs ( "\r\n\tVALID COMMANDS:\r\n" );
    cputs ( "\tD = delay 0-9 seconds.\r\n" );
    cputs ( "\tE = even parity.\r\n" );
    cputs ( "\tF = (fast) 1200-baud.\r\n" );
    cputs ( "\tN = no parity.\r\n" );
    cputs ( "\tO = odd parity.\r\n" );
    cputs ( "\tQ = quit, return to DOS.\r\n" );
    cputs ( "\tR = reset modem.\r\n" );
    cputs ( "\tS = (slow) 300-baud.\r\n" );
    cputs ( "\tU = use script file.\r\n" );
    cputs ( "\tV = verify file input.\r\n" );
    cputs ( "\tW = wait for char." );
   }
  break;

 case 'N' :
  if ( wp )
```

Figure 6-8. Continued. *(more)*

```
          cputs ( "o Parity\r\n" );
        Set_Parity ( 1 );
        break;

      case 'O' :
        if ( wp )
          cputs ( "dd Parity\r\n" );
        Set_Parity ( 3 );
        break;

      case 'R' :
        if ( wp )
          cputs ( "ESET Comm Port\r\n" );
        Init_Comm ();
        break;

      case 'Q' :
        if ( wp )
          cputs ( " = QUIT Command\r\n" );
        Ch = ( - 1 );
        break;

      case 'U' :
        if ( in_file && ! vflag )
          putchx ( 'U' );
        cputs ( "se file: " );
        getfil ();
        cputs ( "File " );
        cputs ( in_file ? "Open\r\n" : "Bad\r\n" );
        waitchr = 0;
        break;

      case 'V' :
        if ( wp )
          { cputs ( "erify flag toggled " );
            cputs ( vflag ? "OFF\r\n" : "ON\r\n" );
          }
        vflag = vflag ? False : True;
        break;

      case 'W' :
        if ( wp )
          cputs ( "ait for: <" );
        waitchr = kbd_wait ();
        if ( waitchr == ' ' )
          waitchr = 0;
        if ( wp )
          { if ( waitchr )
              putchx ( (char) waitchr );
            else
              cputs ( "no wait" );
```

Figure 6-8. Continued. *(more)*

```
          cputs ( ">\r\n" );
        }
      break;

    default :
      if ( wp )
        { cputs ( "Don't know " );
          putchx ( (char) Ch );
          cputs ( "\r\nUse 'H' command for Help.\r\n" );
        }
      Ch = '?';
    }
  if ( wp )                             /* if window open...        */
    { cputs ( "\r\n[any key]\r" );
      while ( Read_Keyboard () == EOF ) /* wait for response        */
        ;
    }
  return Ch ;
}

kbd_wait ()                             /* wait for input           */
{ int c ;
  while (( c = kb_file ()) == ( - 1 ))
    ;
  return c & 255;
}

kb_file ()                              /* input from kb or file    */
{ int c ;
  if ( in_file )                        /* USING SCRIPT             */
    { c = Wants_To_Abort ();            /* use first as flag        */
      if ( waitchr && ! c )
        c = ( - 1 );                    /* then for char            */
      else
        if ( c ¦¦ ( c = getc ( in_file )) == EOF ¦¦ c == 26 )
          { fclose ( in_file );
            cputs ( "\r\nScript File Closed\r\n" );
            in_file = NULL;
            waitchr = 0;
            c = ( - 1 );
          }
      else
        if ( c == '\n' )                /* ignore LFs in file       */
          c = ( - 1 );
      if ( c == '\\' )                  /* process Esc sequence     */
        c = esc ();
      if ( vflag && c != ( - 1 ))       /* verify file char         */
        { putchx ( '{' );
          putchx ( (char) c );
          putchx ( '}' );
        }
```

Figure 6-8. Continued. *(more)*

```
  }
  else                                  /* USING CONSOLE            */
    c = Read_Keyboard ();               /* if not using file        */
  return ( c );
}

esc ()                                  /* script translator        */
{ int c ;
  c = getc ( in_file );                 /* control chars in file    */
  switch ( toupper ( c ))
    {
    case 'E' :
      c = ESC;
      break;

    case 'N' :
      c = '\n';
      break;

    case 'R' :
      c = '\r';
      break;

    case 'T' :
      c = '\t';
      break;

    case '^' :
      c = getc ( in_file ) & 31;
      break;
    }
  return ( c );
}

FILE * getfil ()
{ char fnm [ 20 ];
  getnam ( fnm, 15 );                   /* get the name             */
  if ( ! ( strchr ( fnm, '.' )))
    strcat ( fnm, ".SCR" );
  return ( in_file = fopen ( fnm, "r" ));
}

void getnam ( b, s ) char * b;          /* take input to buffer     */
 int s ;
{ while ( s -- > 0 )
    { if (( * b = (char) kbd_wait ()) != '\r' )
        putchx ( * b ++ );
      else
        break ;
    }
  putchx ( '\n' );
```

Figure 6-8. Continued. *(more)*

```
  * b = 0;
}

char * addext ( b,                       /* add default EXTension    */
     e ) char * b,
     * e;
{ static char bfr [ 20 ];
  if ( strchr ( b, '.' ))
    return ( b );
  strcpy ( bfr, b );
  strcat ( bfr, e );
  return ( bfr );
}

void put_cap ( c ) char c ;
{ if ( cap_file && c != 13 )             /* strip out CRs            */
    fputc ( c, cap_file );               /* use MS-DOS buffering     */
}

void cap_flush ()                        /* end Capture mode         */
{ if ( cap_file )
    { fclose ( cap_file );
      cap_file = NULL;
      cputs ( "\r\nCapture file closed\r\n" );
    }
}

/*      TIMER SUPPORT STUFF (IBMPC/MSDOS)        */
static long timr;                        /* timeout register         */

static union REGS rgv ;

long getmr ()
{ long now ;                             /* msec since midnite       */
  rgv.x.ax = 0x2c00;
  intdos ( & rgv, & rgv );
  now = rgv.h.ch;                        /* hours                    */
  now *= 60L;                            /* to minutes               */
  now += rgv.h.cl;                       /* plus min                 */
  now *= 60L;                            /* to seconds               */
  now += rgv.h.dh;                       /* plus sec                 */
  now *= 100L;                           /* to 1/100                 */
  now += rgv.h.dl;                       /* plus 1/100               */
  return ( 10L * now );                  /* msec value               */
}

void Delay ( n ) int n ;                 /* sleep for n msec         */
{ long wakeup ;
  wakeup = getmr () + ( long ) n;        /* wakeup time              */
  while ( getmr () < wakeup )
    ;                                    /* now sleep                */
```

Figure 6-8. Continued. *(more)*

```
}

void Start_Timer ( n ) int n ;         /* set timeout for n sec      */
{ timr = getmr () + ( long ) n * 1000L;
}

Timer_Expired ()        /* if timeout return 1 else return 0         */
{ return ( getmr () > timr );
}

Set_Vid ()
{ _i_v ();                              /* initialize video           */
  return 0;
}

void locate ( row, col ) int row ,
     col;
{ _cy = row % 25;
  _cx = col % 80;
  _wrpos ( row, col );                  /* use ML from CH2.ASM        */
}

void deol ()
{ _deol ();                             /* use ML from CH2.ASM        */
}

void deos ()
{ deol ();
  if ( _cy < 24 )                       /* if not last, clear         */
    { rgv.x.ax = 0x0600;
      rgv.x.bx = ( _atr << 8 );
      rgv.x.cx = ( _cy + 1 ) << 8;
      rgv.x.dx = 0x184F;
      int86 ( 0x10, & rgv, & rgv );
    }
  locate ( _cy, _cx );
}

void cls ()
{ _cls ();                              /* use ML                     */
}

void cursor ( yn ) int yn ;
{ rgv.x.cx = yn ? 0x0607 : 0x2607;      /* ON/OFF                     */
  rgv.x.ax = 0x0100;
  int86 ( 0x10, & rgv, & rgv );
}

void revvid ( yn ) int yn ;
{ if ( yn )
    _atr = _color ( 8, 7 );             /* black on white             */
```

Figure 6-8. Continued. *(more)*

```
  else
    _atr = _color ( 15, 0 );            /* white on black           */
}

putchx ( c ) char c ;                   /* put char to CRT          */
{ if ( c == '\n' )
    putch ( '\r' );
  putch ( c );
  return c ;
}

Read_Keyboard ()                    /* get keyboard character
                                       returns -1 if none present   */
{ int c ;
  if ( kbhit ())                        /* no char at all           */
    return ( getch ());
  return ( EOF );
}

/*      MODEM SUPPORT                   */
static char mparm,
     wrk [ 80 ];

void Init_Comm ()               /* initialize comm port stuff       */
{ static int ft = 0;                    /* firstime flag            */
  if ( ft ++ == 0 )
    i_m ();
  Set_Parity ( 1 );                     /* 8,N,1                    */
  Set_Baud ( 1200 );                    /* 1200 baud                */
}

#define B1200 0x80                      /* baudrate codes           */
#define B300 0x40

Set_Baud ( n ) int n ;                  /* n is baud rate           */
{ if ( n == 300 )
    mparm = ( mparm & 0x1F ) + B300;
  else
    if ( n == 1200 )
      mparm = ( mparm & 0x1F ) + B1200;
  else
    return 0;                           /* invalid speed            */
  sprintf ( wrk, "Baud rate = %d\r\n", n );
  cputs ( wrk );
  set_mdm ( mparm );
  return n ;
}

#define PAREVN 0x18                     /* MCR bits for commands    */
#define PARODD 0x10
#define PAROFF 0x00
```

Figure 6-8. Continued. *(more)*

```
#define STOP2 0x40
#define WORD8 0x03
#define WORD7 0x02
#define WORD6 0x01

Set_Parity ( n ) int n ;                 /* n is parity code           */
{ static int mmode;
  if ( n == 1 )
    mmode = ( WORD8 ¦ PAROFF );          /* off                        */
  else
    if ( n == 2 )
      mmode = ( WORD7 ¦ PAREVN );        /* on and even                */
  else
    if ( n == 3 )
      mmode = ( WORD7 ¦ PARODD );        /* on and odd                 */
  else
    return 0;                            /* invalid code               */
  mparm = ( mparm & 0xE0 ) + mmode;
  sprintf ( wrk, "Parity is %s\r\n", ( n == 1 ? "OFF" :
                                     ( n == 2 ? "EVEN" : "ODD" )));
  cputs ( wrk );
  set_mdm ( mparm );
  return n ;
}

Write_Modem ( c ) char c ;               /* return 1 if ok, else 0     */
{ wrtmdm ( c );
  return ( 1 );                          /* never any error            */
}

Read_Modem ()
{ return ( rdmdm ());                    /* from int bfr               */
}

void Term_Comm ()                /* uninstall comm port drivers        */
{ u_m ();
}

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

(more)

Table 6-13. *Continued.*

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

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

        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
        jz      label3                  ; quit if space found
        stosb                           ; else copy this character
        lodsb                           ; get next character
        jmp     label2

label3:
        xor     al,al                   ; store final NULL to
        stosb                           ; create ASCIIZ 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:                               ; now close files...
        mov     bx,handle1          ; handle for MYFILE.DAT
        mov     ah,3eh              ; Function 3EH = close file
        int     21h                 ; transfer to MS-DOS
        jc      error               ; 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)

```
                                ; position the file pointer...
        mov     cx,0            ; CX = high part of file offset
        mov     dx,32768        ; 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 ASCIIZ 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 ASCIIZ 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 ASCIIZ 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 ASCIIZ 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	D	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):

Time Format

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

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 specifed 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:                           ; now close files...
        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

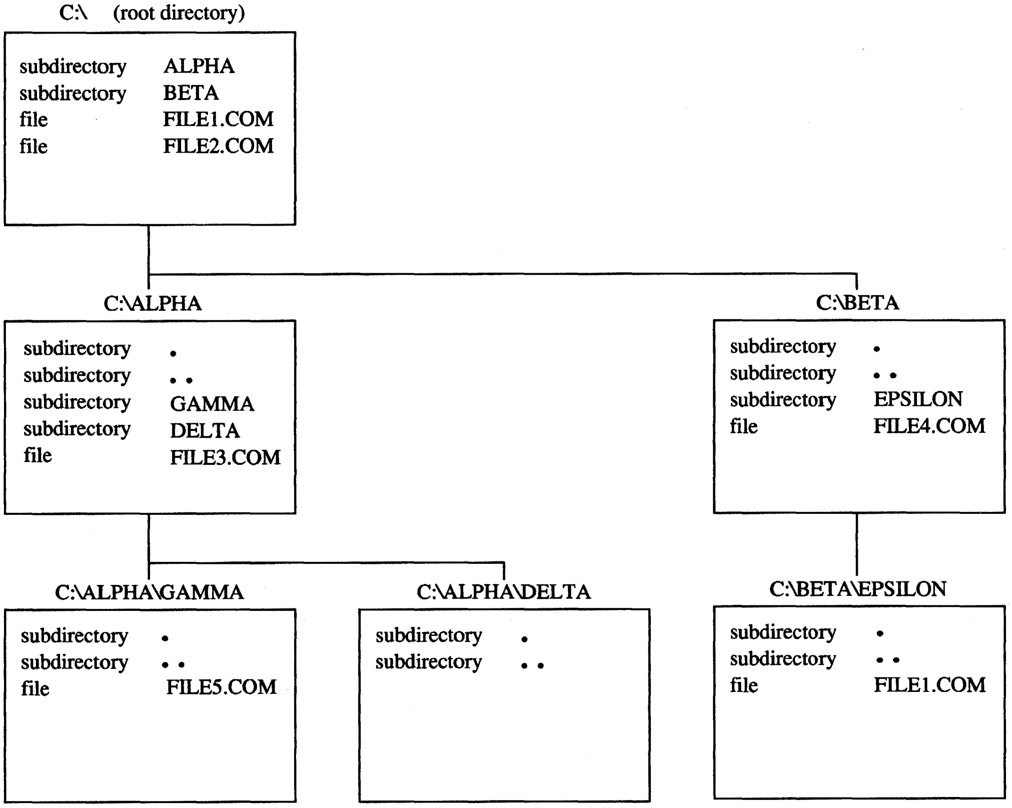


Figure 8-1. Typical hierarchical directory structure (MS-DOS versions 2.0 and later).

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.

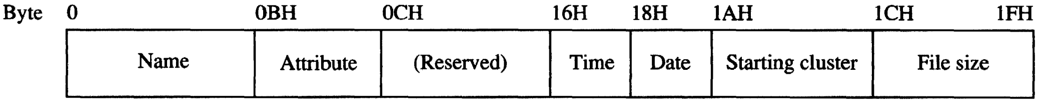


Figure 8-2. Format of a directory entry.

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.

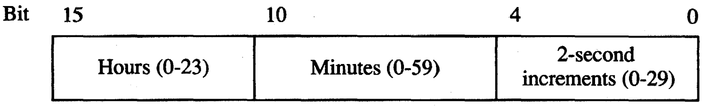


Figure 8-4. Format of the time field in a directory entry.

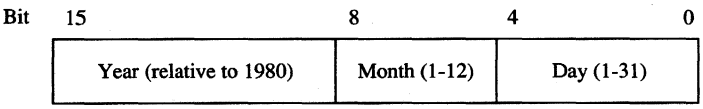


Figure 8-5. Format of the date field in a directory entry.

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 comand 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;
;
;-------------------------------------------------------------------------------

                PUBLIC  _SetDTA
_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 */
;
;----------------------------------------------------------------------------

                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 */
;
;----------------------------------------------------------------------------

                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 DGROUP: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 DGROUP:DTA + 8
                mov     di,offset DGROUP:VolLabel
                call    CopyName
                mov     ax,offset DGROUP: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 */
;
;----------------------------------------------------------------------------

                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                   ; copy new name into FCB ..
                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,

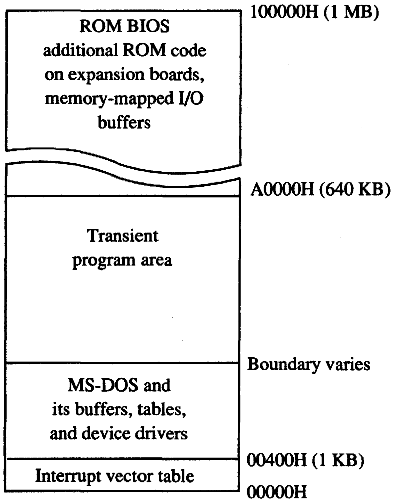


Figure 9-1. A diagram showing conventional memory in an IBM PC-compatible MS-DOS system. The bottom 1024 bytes of memory are used for the interrupt vector table. The memory above the vector table, up to the 640 KB boundary, is available for use by MS-DOS and the programs that run under its control. The 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
        mov     sp,offset stk   ; to a safe place...

                                ; 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

                                ; open working file...
        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:                           ; read working file...
        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.

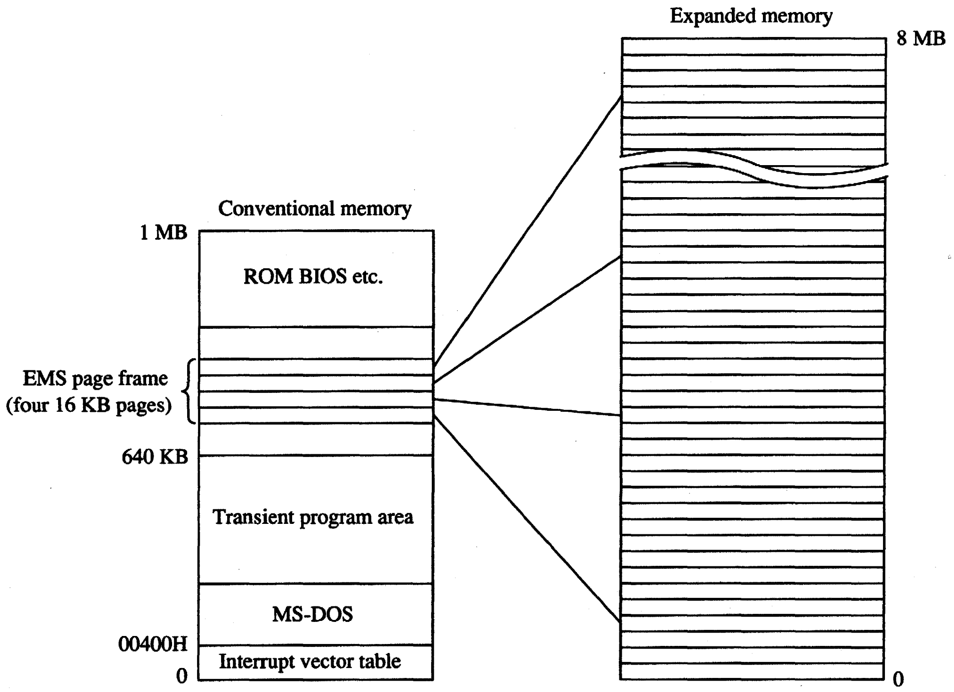


Figure 9-5. A sketch of the relationship of expanded memory to conventional memory; 16 KB pages of expanded memory are mapped into a 64 KB area, called the page frame, above the 640 KB boundary. The location of the page frame can be configured by the user to eliminate conflicts with ROMs or I/O buffers 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: EMMXXXX0. 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 *EMMXXXX0*. 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     'EMMXXXX0'      ; 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.
				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.

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	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 interrupt 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*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*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           ; segment * 16
        mov     dx,16
        mul     dx
        add     ax,bx           ; + offset -> linear address
        adc     dx,0
        mov     es:[si+1ah],ax
        mov     es:[si+1ch],dl

        shr     cx,1            ; convert length to words
        mov     ah,87h          ; Fxn 87H = block move
        int     15h             ; transfer to ROM BIOS

        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 (ASCIIZ) 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 pathname 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$
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:pars   ; 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
        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
```

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  equu    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 predefined hardware interrupt. Therefore, when it is invoked, an active TSR almost always

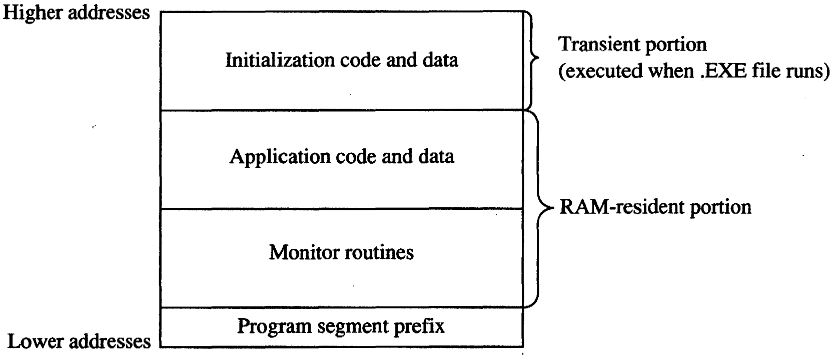


Figure 11-1. Organization of a TSR program in memory.

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
                push    ds                  ; preserve these registers
                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
                iret

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
                iret

ISR1B           ENDP


ISR23           PROC    far             ; INT 23H trap (MS-DOS Ctrl-C)

                mov     byte ptr cs:Trap23,TRUE
                iret

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 KBNum OR KBScroll) 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
                jb      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
                                        ; MS-DOS versions earlier than 3.1
LF1:            cmp     ss:LFbyte,0     ; CMP ErrorMode,0
                jne     LFnear
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.

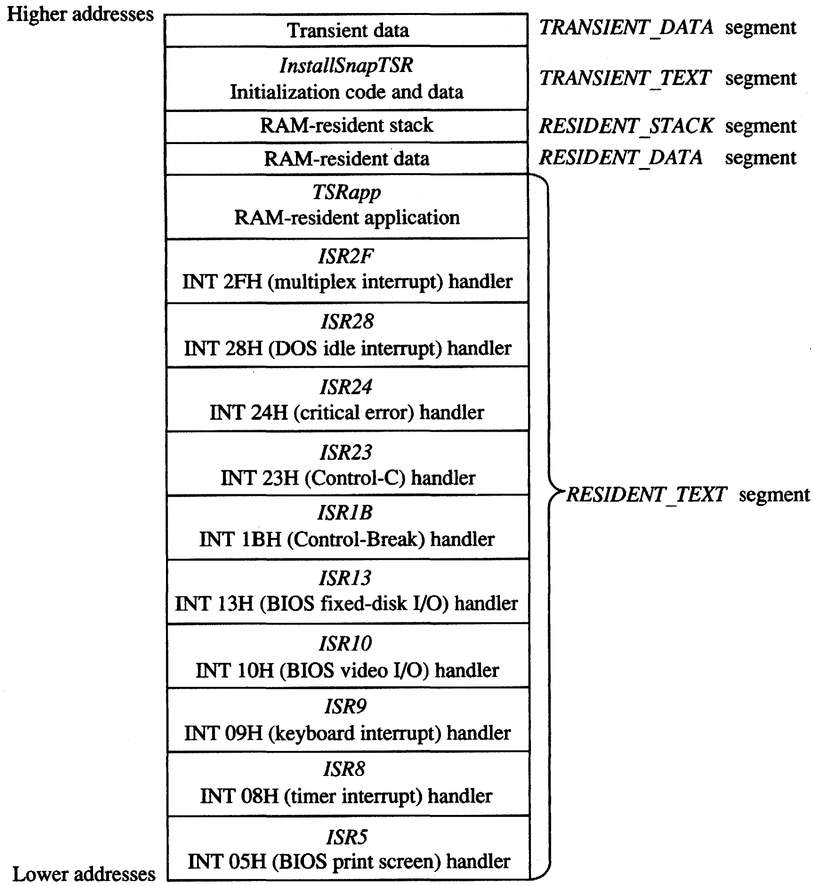

Higher addresses
Transient data
TRANSIENT_DATA segment
InstallSnapTSR
Initialization code and data
TRANSIENT_TEXT segment
RAM-resident stack
RESIDENT_STACK segment
RAM-resident data
RESIDENT_DATA segment
TSRapp
RAM-resident application
ISR2F
INT 2FH (multiplex interrupt) handler
ISR28
INT 28H (DOS idle interrupt) handler
ISR24
INT 24H (critical error) handler
ISR23
INT 23H (Control-C) handler
ISR1B
INT 1BH (Control-Break) handler
ISR13
INT 13H (BIOS fixed-disk I/O) handler
ISR10
INT 10H (BIOS video I/O) handler
ISR9
INT 09H (keyboard interrupt) handler
ISR8
INT 08H (timer interrupt) handler
ISR5
INT 05H (BIOS print screen) handler
RESIDENT_TEXT segment
Lower addresses

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 *IOStack* (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.

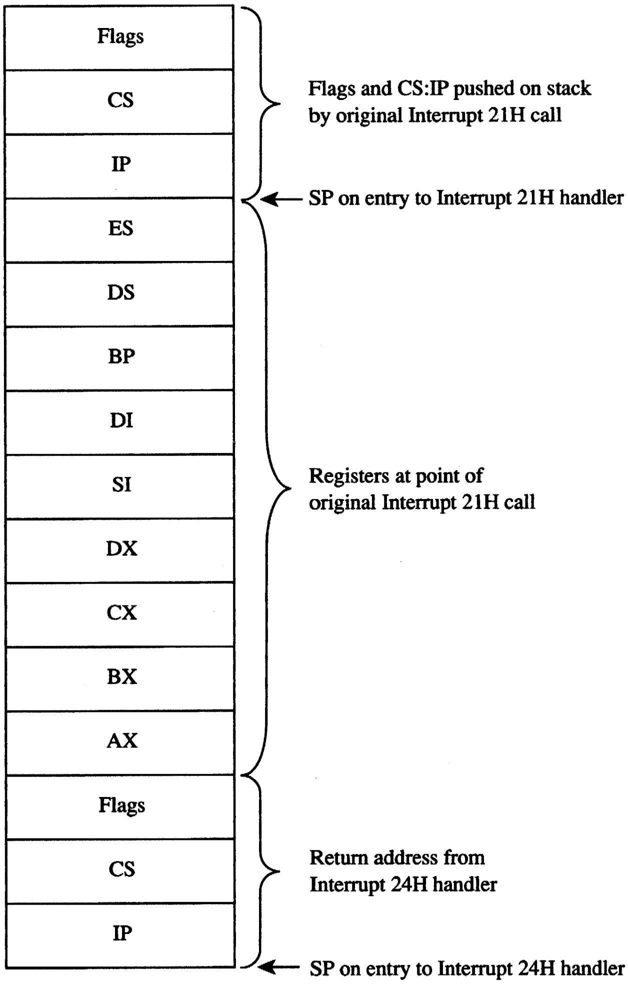

Flags
CS
IP
ES
DS
BP
DI
SI
DX
CX
BX
AX
Flags
CS
IP
Flags and CS:IP pushed on stack by original Interrupt 21H call
SP on entry to Interrupt 21H handler
Registers at point of original Interrupt 21H call
Return address from Interrupt 24H handler
SP on entry to Interrupt 24H handler

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
get24	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.
res24	Restores the address of the previous critical error handler, which was saved by a call to *get24*, into the Interrupt 24 vector.
int24	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      'aArRiIfF'      ; 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 *Ignore* for the program to continue or *Abort* 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 shutdown 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
        jz      retry           ; if BL = 01H retry operation
        cmp     bl,04h
        jz      cleanup         ; if BL = 04H clean up and exit
        cmp     bl,05h
        jz      panic           ; if BL = 05H exit immediately
        .
        .
        .
```

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
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
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
                                ; 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
        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).

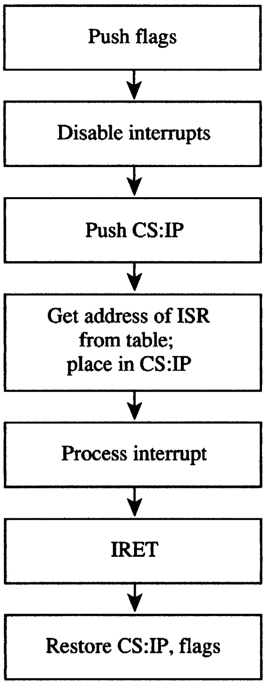


Figure 13-1. General interrupt sequence.

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.

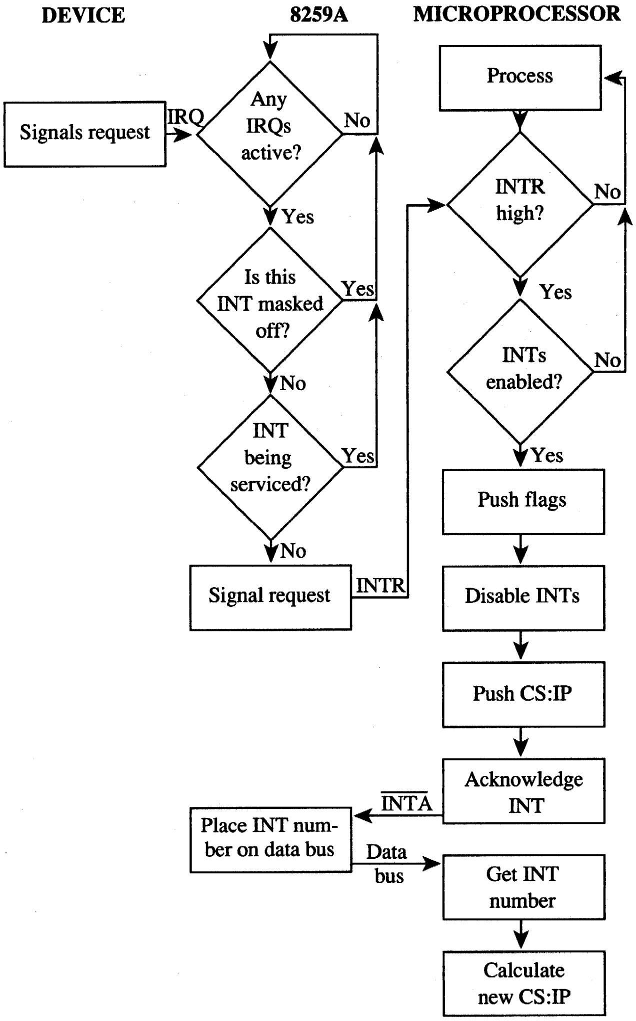


Figure 13-2. Maskable interrupt service.

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.

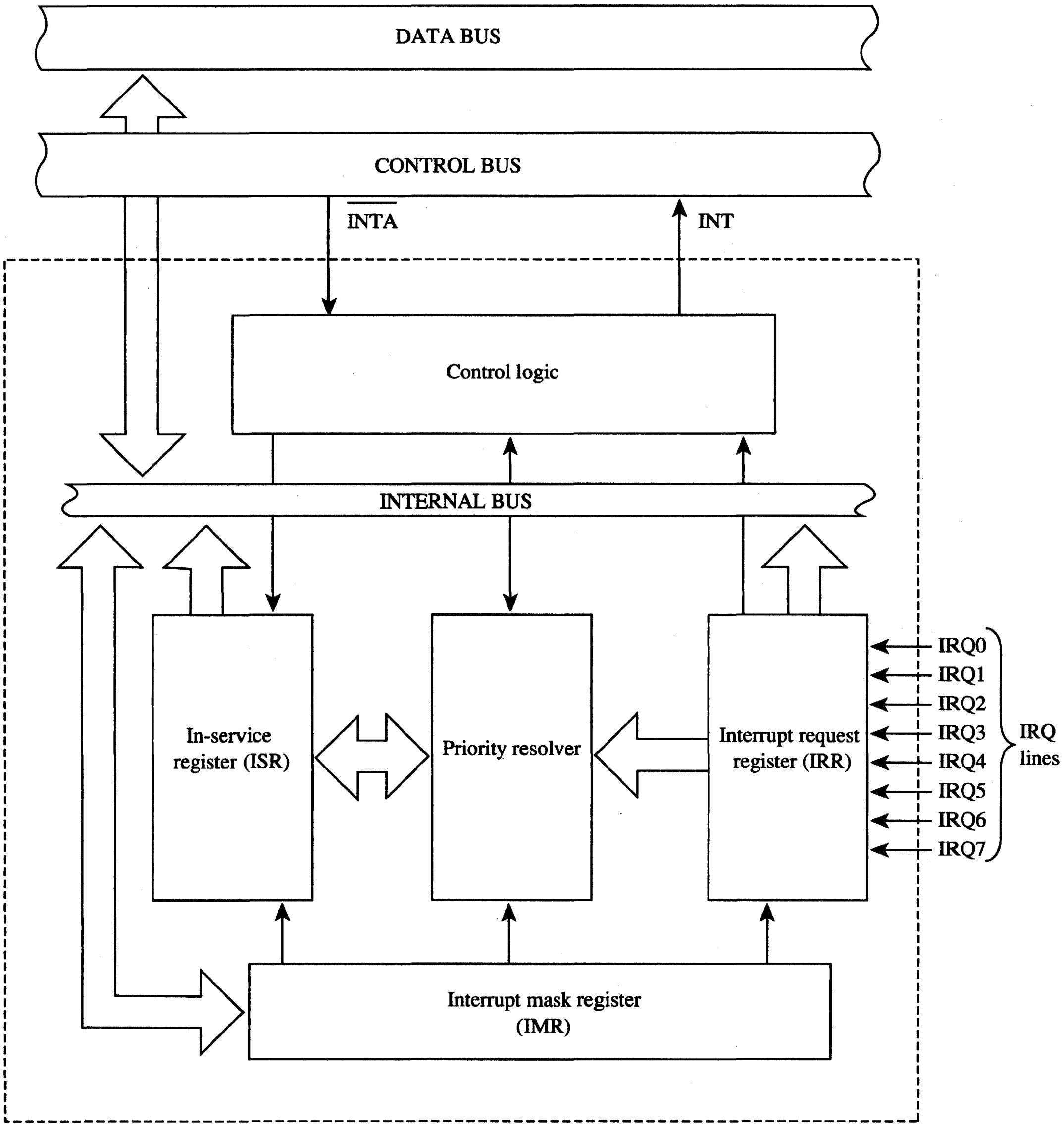


Figure 13-3. Block diagram of the 8259A Programmable Interrupt Controller.

The IRQ levels

When two or more unserviced 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

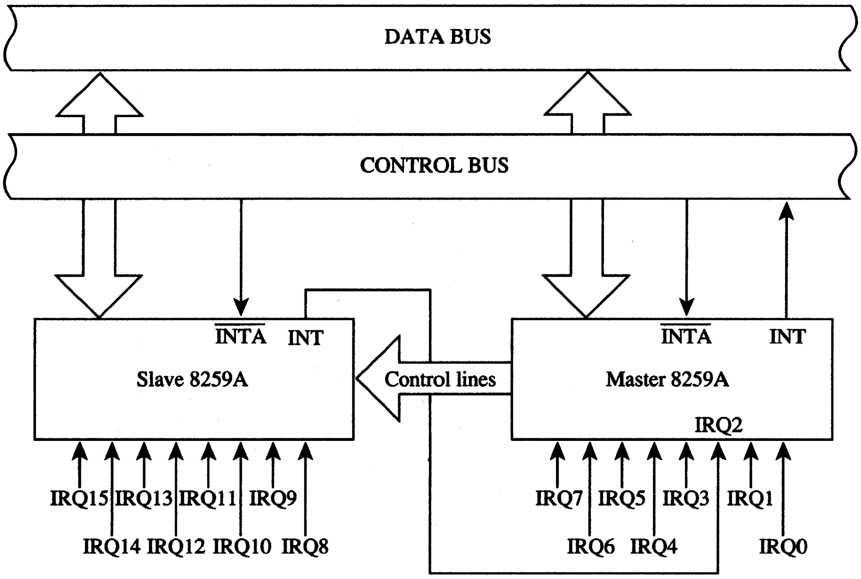


Figure 13-4. A graphic representation of the cascade effect for IRQ priorities.

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}
        add     al,'0'          ; and return digit in AL
        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	*stdin* (standard input)	CON
1	*stdout* (standard output)	CON
2	*stderr* (standard error)	CON
3	*stdaux* (standard auxiliary)	AUX
4	*stdlst* (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
< *file*	Contents of the specified *file* are used instead of the keyboard as the program's standard input.
< *device*	Program takes its standard input from the named *device* instead of from the keyboard.
> *device*	Program sends its standard output to the named *device* instead of to the video display.
> *file*	Program sends its standard output to the specified *file* instead of to the video display.
>> *file*	Program appends its standard output to the current contents of the specified *file* instead of to the video display.
p1 ¦ *p2*	Standard output of program *p1* is routed to become the standard input of program *p2* (output of *p1* is said to be piped to *p2*).

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 (1AH) 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    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

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
;

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,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    translt         ; 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  equu    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;
                                        /* read a character */
        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 EXECSORT's data segment
        mov     stk_seg,ss              ; save EXECSORT'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 EXECSORT 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.

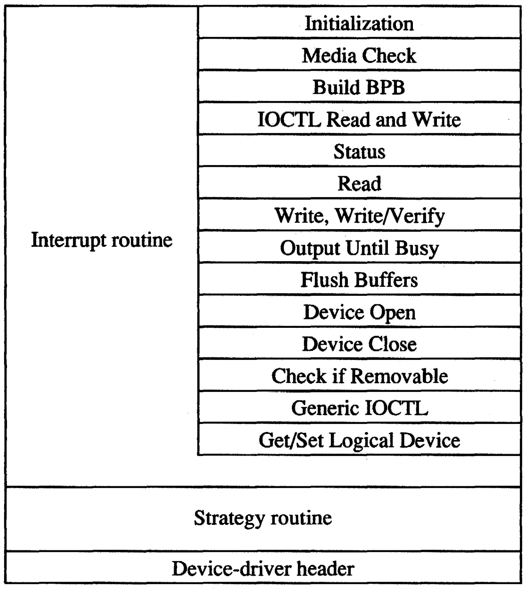


Figure 15-1. General structure of an MS-DOS installable device driver.

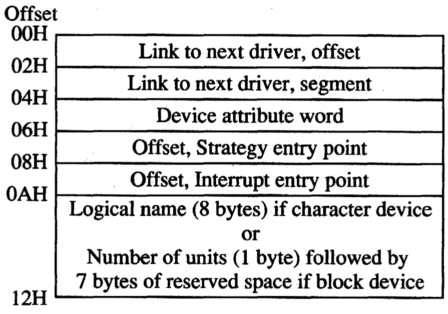

Offset
00H
02H
04H
06H
08H
0AH
12H
Link to next driver, offset
Link to next driver, segment
Device attribute word
Offset, Strategy entry point
Offset, Interrupt entry point
Logical name (8 bytes) if character device
or
Number of units (1 byte) followed by
7 bytes of reserved space if block device

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 (*stdout*)
0	1 if current standard input (*stdin*)

* 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.

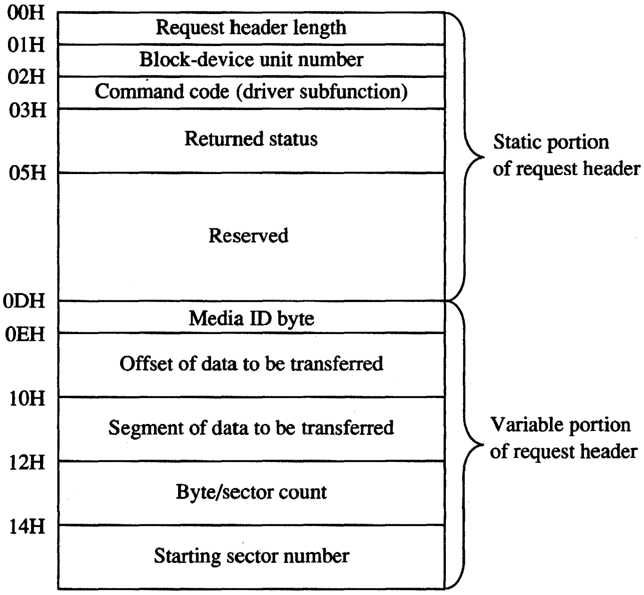

00H
Request header length
01H
Block-device unit number
02H
Command code (driver subfunction)
03H
Returned status
05H
Reserved
0DH
Media ID byte
0EH
Offset of data to be transferred
10H
Segment of data to be transferred
12H
Byte/sector count
14H
Starting sector number
Static portion of request header
Variable portion of request header

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

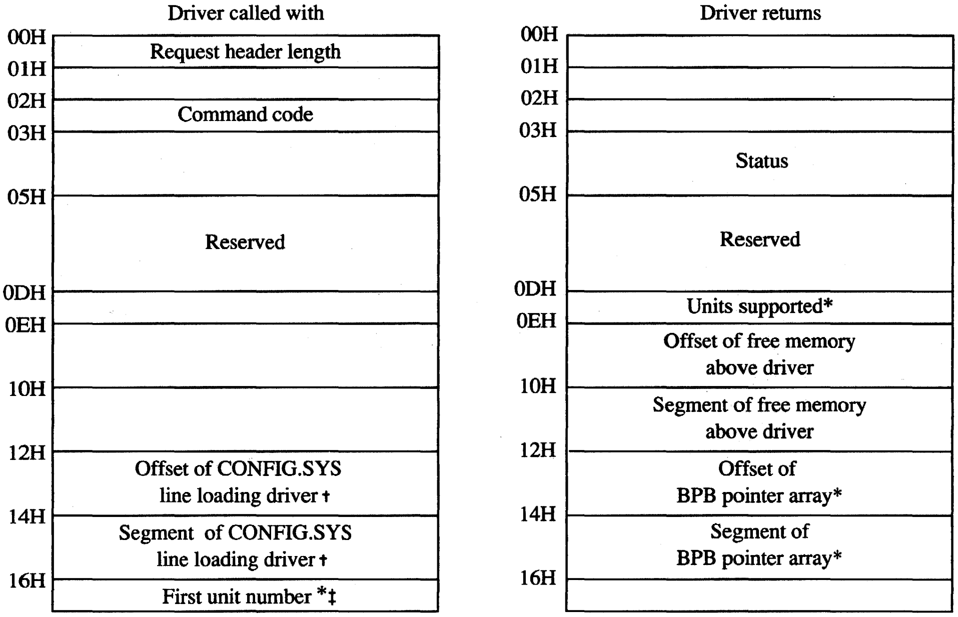
Functions 0 through 12 must be supported by a driver's Interrupt section under all versions of MS-DOS. Drivers tailored for versions 3.0 and 3.1 can optionally support an additional 4 functions defined under those versions of the operating system and drivers designed for version 3.2 can support 3 more, 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).



	Driver called with	Driver returns
00H	Request header length	
01H		
02H	Command code	
03H		Status
05H	Reserved	Reserved
0DH		Units supported*
0EH		Offset of free memory above driver
10H		Segment of free memory above driver
12H	Offset of CONFIG.SYS line loading driver †	Offset of BPB pointer array*
14H	Segment of CONFIG.SYS line loading driver †	Segment of BPB pointer array*
16H	First unit number *‡	

* 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.

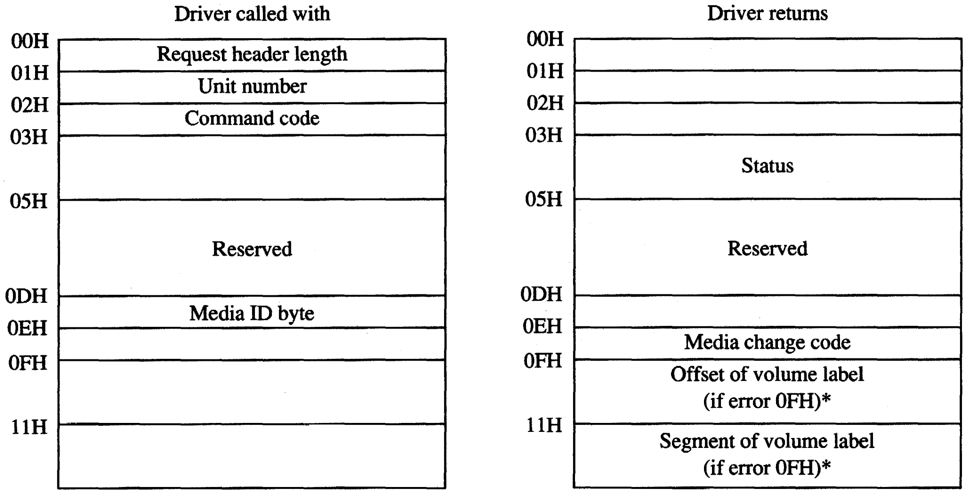
The Media Check function

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
0F0H	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.

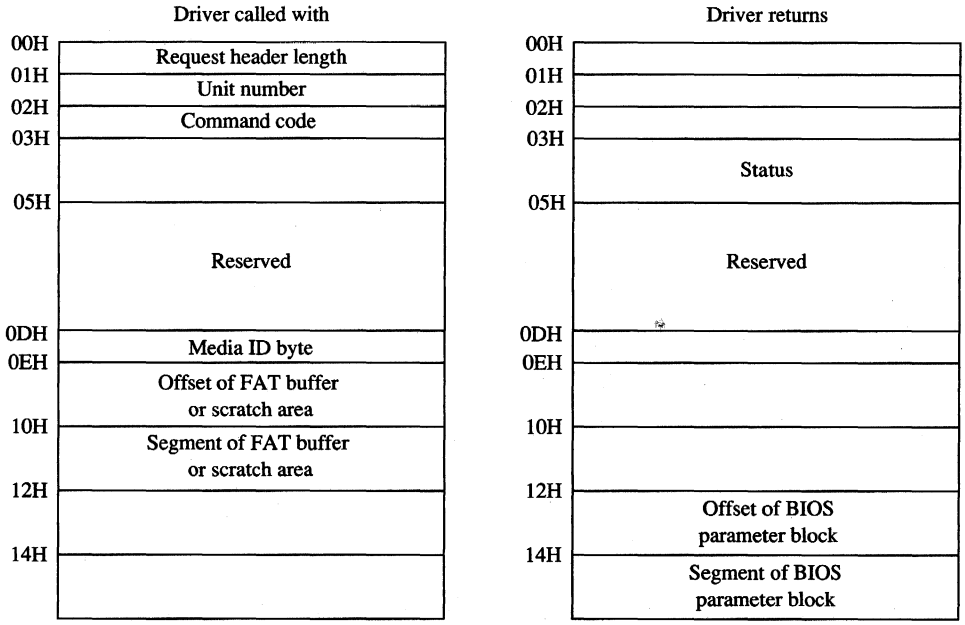


Figure 15-6. Build BPB request header (command code 2).

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).

	Driver called with	Driver returns
00H	Request header length	
01H	Unit number*	
02H	Command code	
03H		Status
05H	Reserved	Reserved
0DH	Media ID byte*	
0EH	Offset of data	
10H	Segment of data	
12H	Bytes/sectors requested	Bytes/sectors transferred
14H	Starting sector number*	
16H		Offset of volume label (if error 0FH)* †
18H		Segment of volume label (if error 0FH)* †

* 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.

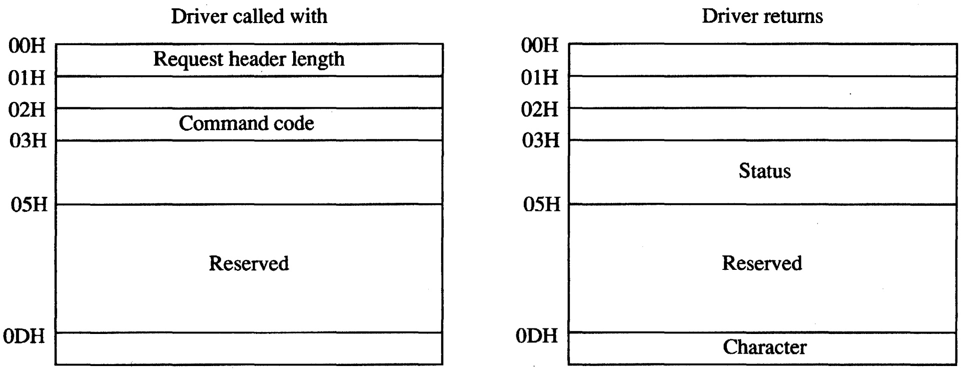


Figure 15-8. The Nondestructive Read request header.

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.

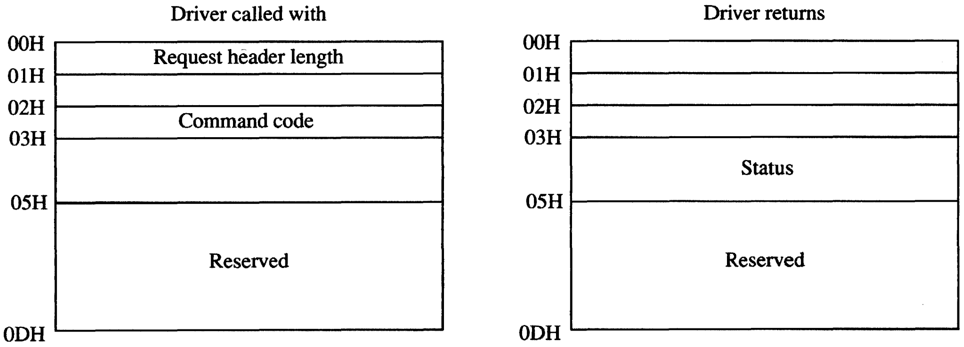


Figure 15-9. The request header for Input Status (command code 6), Flush Input Buffers (command code 7), Output Status (command code 10), and Flush Output Buffers (command code 11).

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.

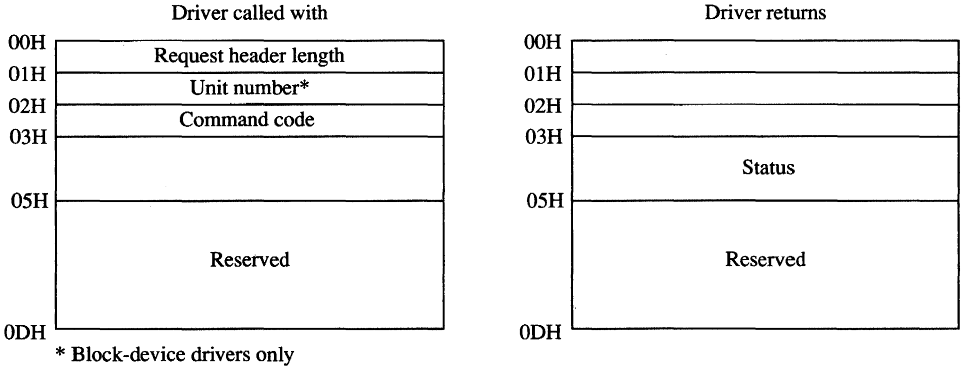
In block-device drivers, the Flush Buffer functions have no meaning. They should simply set the done flag in the status word of the request header and return.

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).



	Driver called with	Driver returns
00H	Request header length	
01H	Unit number*	
02H	Command code	
03H		Status
05H	Reserved	Reserved
0DH		

* Block-device drivers only

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.

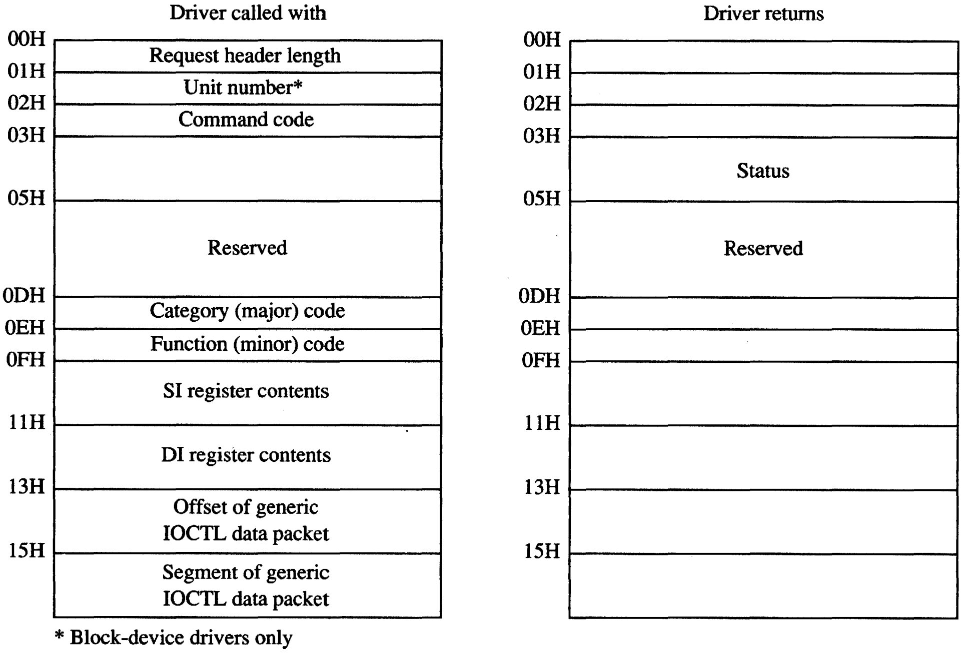
In block-device drivers, the Output Until Busy function should simply set the done flag in the status word of the request header and return.

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 Subfunctions 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.



	Driver called with	Driver returns
00H	Request header length	
01H	Unit number*	
02H	Command code	
03H		Status
05H	Reserved	Reserved
0DH	Category (major) code	
0EH	Function (minor) code	
0FH	SI register contents	
11H	DI register contents	
13H	Offset of generic IOCTL data packet	
15H	Segment of generic IOCTL data packet	

* Block-device drivers only

Figure 15-11. Generic IOCTL request header.

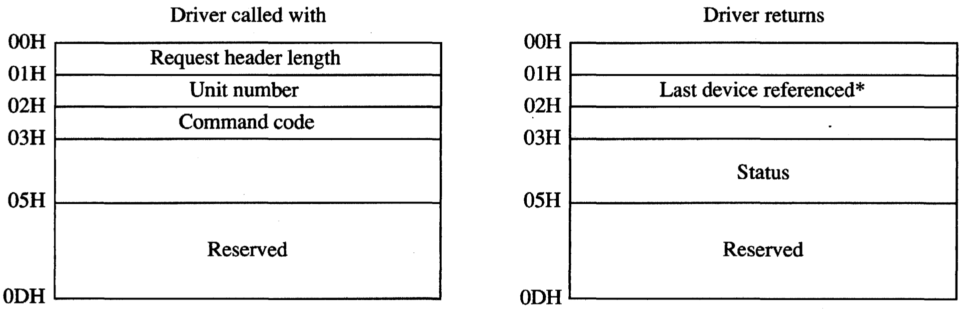
The Get Logical Device and Set Logical Device functions

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.



* Get Logical Device (Command code 23) only

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.

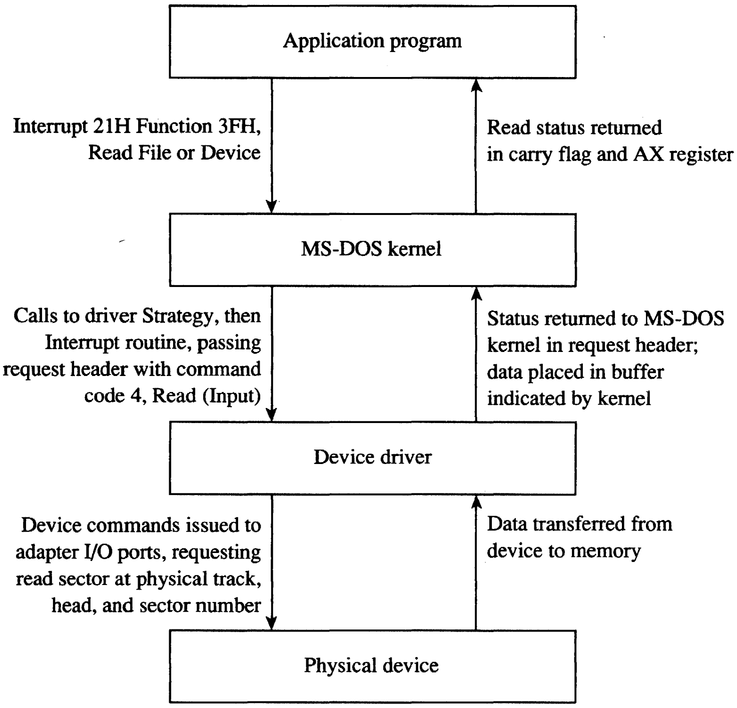

Application program
Interrupt 21H Function 3FH, Read File or Device
Read status returned in carry flag and AX register
MS-DOS kernel
Calls to driver Strategy, then Interrupt routine, passing request header with command code 4, Read (Input)
Status returned to MS-DOS kernel in request header; data placed in buffer indicated by kernel
Device driver
Device commands issued to adapter I/O ports, requesting read sector at physical track, head, and sector number
Data transferred from device to memory
Physical device

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)*

```
                                ; device-driver header
Header  dd      -1              ; 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)

        call    Setup           ; set up transfer variables

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∗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}:0345_{16}

and

1004_{16}: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*(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 multitasking, 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.

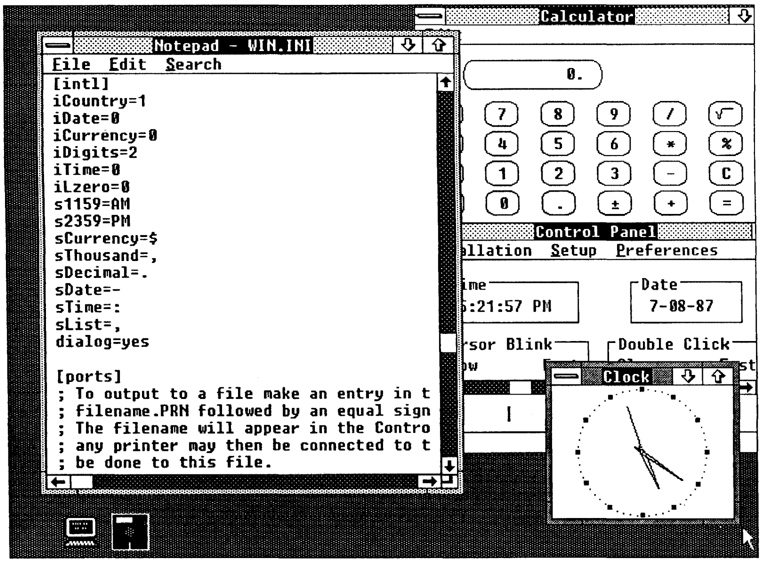
The Windows Display

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.



```
[intl]
iCountry=1
iDate=0
iCurrency=0
iDigits=2
iTime=0
iLzero=0
s1159=AM
s2359=PM
sCurrency=$
sThousand=,
sDecimal=.
sDate=-
sTime=:
sList=,
dialog=yes

[ports]
; To output to a file make an entry in t
; filename.PRN followed by an equal sign
; The filename will appear in the Contro
; any printer may then be connected to t
; be done to this file.
```

```
me
:21:57 PM
Date
7-08-87
sor Blink
Double Click
```

Figure 17-1. A typical Windows display.

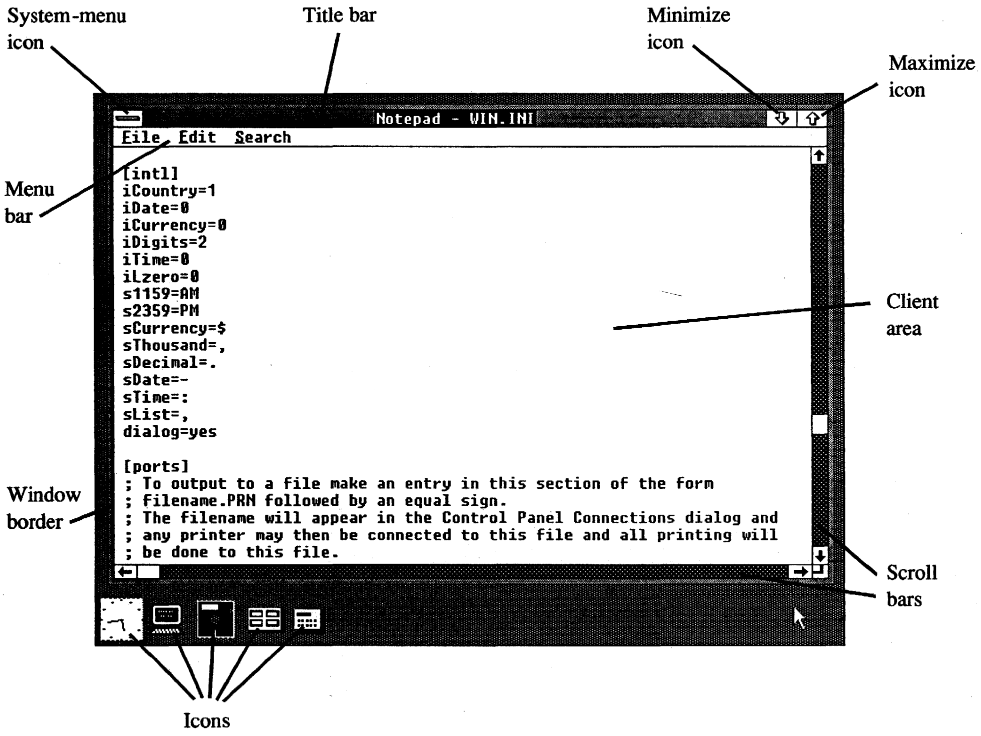
Parts of the window

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.



```
[intl]
iCountry=1
iDate=0
iCurrency=0
iDigits=2
iTime=0
iLzero=0
s1159=AM
s2359=PM
sCurrency=$
sThousand=,
sDecimal=.
sDate=-
sTime=:
sList=,
dialog=yes

[ports]
; To output to a file make an entry in this section of the form
; filename.PRN followed by an equal sign.
; The filename will appear in the Control Panel Connections dialog and
; any printer may then be connected to this file and all printing will
; be done to this file.
```

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.

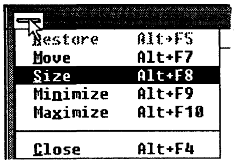


Figure 17-3. A system menu, displayed either when the user clicks the system-menu icon (top left corner) or presses Alt-Spacebar.

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

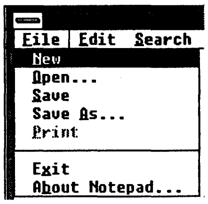


Figure 17-5. The NOTEPAD program's pop-up file menu.

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.

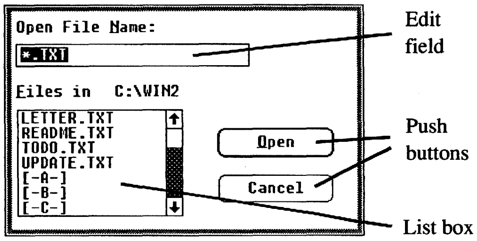
When a program is maximized to fill the full screen, Windows does not draw the window border. Programs that use a window of a fixed size do not have a window border either.

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.

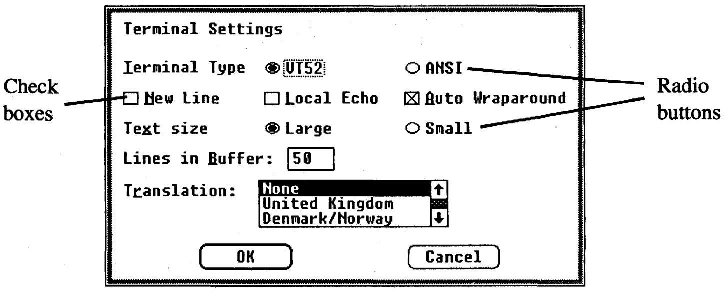


Open File Name:
*.TXT

Files in C:\WIN2

LETTER.TXT
README.TXT
TODO.TXT
UPDATE.TXT
[-A-]
[-B-]
[-C-]

Figure 17-6. A dialog box from the NOTEPAD program, with parts labeled.



Terminal Settings

Terminal Type ● VT52 ○ ANSI

☐ New Line ☐ Local Echo ☒ Auto Wraparound

Text size ● Large ○ Small

Lines in Buffer: 50

Translation: None
United Kingdom
Denmark/Norway

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.

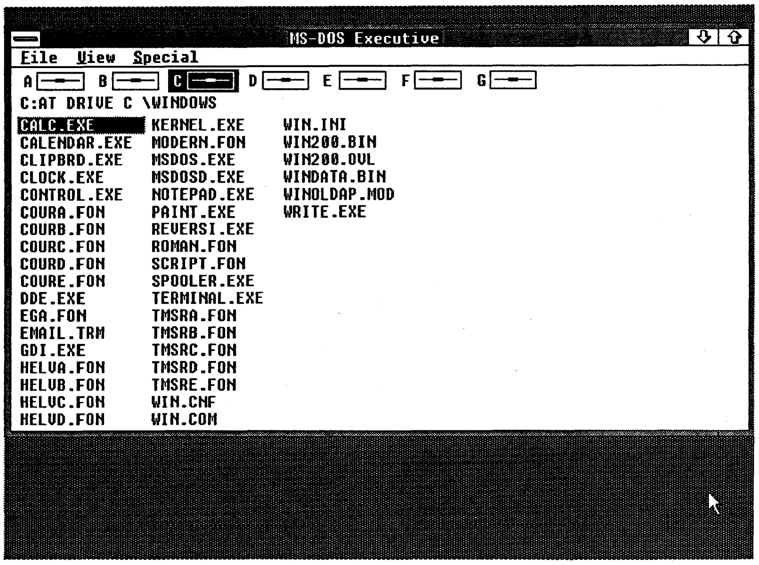
The MS-DOS Executive

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.



```
C:AT DRIVE C \WINDOWS
CALC.EXE      KERNEL.EXE    WIN.INI
CALENDAR.EXE  MODERN.FON    WIN200.BIN
CLIPBRD.EXE   MSDOS.EXE     WIN200.OVL
CLOCK.EXE     MSDOSD.EXE    WINDATA.BIN
CONTROL.EXE   NOTEPAD.EXE   WINOLDAP.MOD
COURA.FON     PAINT.EXE     WRITE.EXE
COURB.FON     REVERSI.EXE
COURC.FON     ROMAN.FON
COURD.FON     SCRIPT.FON
COURE.FON     SPOOLER.EXE
DDE.EXE       TERMINAL.EXE
EGA.FON       TMSRA.FON
EMAIL.TRM     TMSRB.FON
GDI.EXE       TMSRC.FON
HELVA.FON     TMSRD.FON
HELVB.FON     TMSRE.FON
HELVC.FON     WIN.CNF
HELVD.FON     WIN.COM
```

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 multitasking, 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.

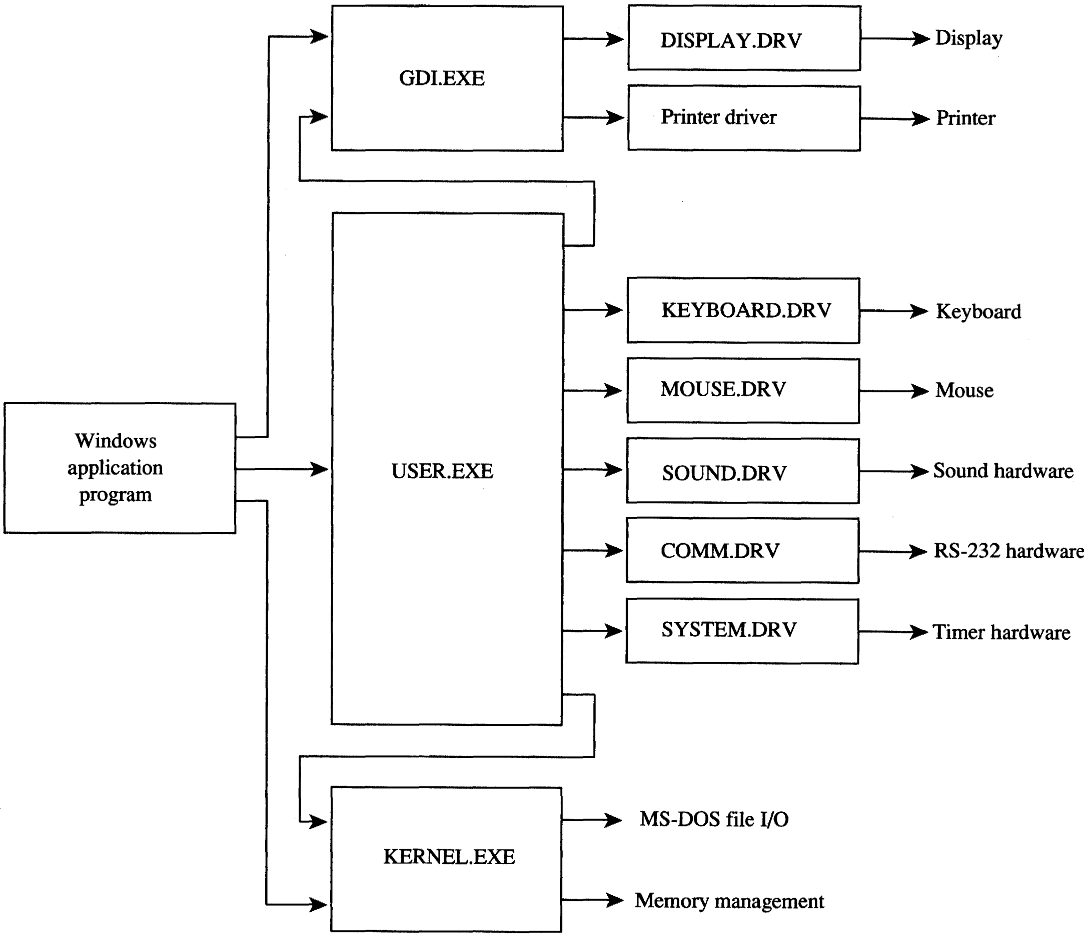


Figure 17-9. A simplified block diagram showing the relationships of an application program, Windows modules (GDI, USER, and KERNEL), driver modules, and system hardware.

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

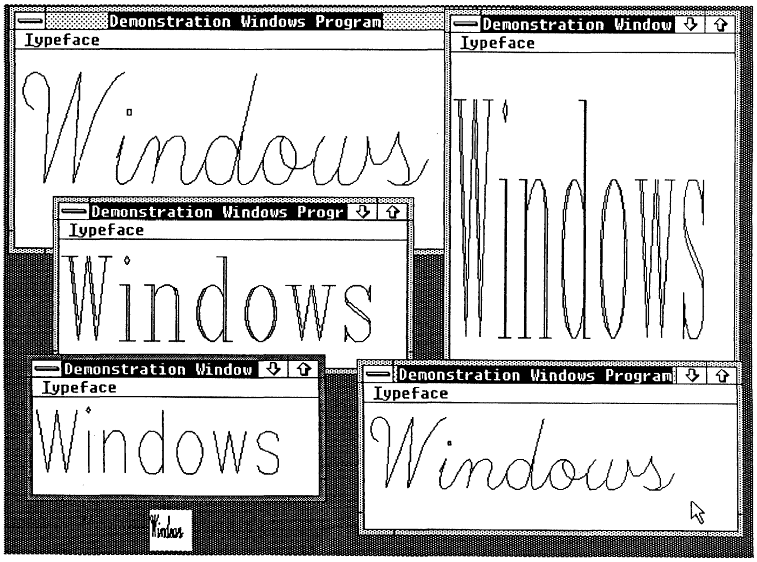


Figure 17-10. A display produced by the example program SAMPLE.

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.

2. Resource script: The resource script is an ASCII file that generally has the extension .RC. This file contains definitions of menus, dialog boxes, string tables, and keyboard accelerators used by the program. The resource script can also reference other files that contain icons, cursors, bitmaps, and fonts in binary form, as well as other read-only data defined by the programmer. When a program is running, Windows loads resources into memory only when they are needed and in most cases can discard them if additional memory space is required.

   SAMPLE.RC, the resource script for the SAMPLE program, is shown in Figure 17-12; it contains only the definition of the menu used in the program.

```
#include "sample.h"

Sample MENU
     BEGIN
          POPUP "&Typeface"
               BEGIN
                    MENUITEM "&Script",  IDM_SCRIPT, CHECKED
                    MENUITEM "&Modern",  IDM_MODERN
                    MENUITEM "&Roman",   IDM_ROMAN
               END
     END
```

Figure 17-12. The resource script for the SAMPLE program.

3. Header (or *include*) file: This file, with the extension .H, can contain definitions of constants or macros, as is customary in C programming. For Windows programs, the header file also reconciles constants used in both the resource script and the program source-code file. For example, in the SAMPLE.RC resource script, each item in the pop-up menu (*Script*, *Modern*, and *Roman*) also includes an identifier—IDM_SCRIPT, IDM_MODERN, and IDM_ROMAN, respectively. These identifiers are merely numbers that Windows uses to notify the program of the user's selection of a menu item. The same names are used to identify the menu selection in the C source-code file. And, because both the resource compiler and the source-code compiler must have access to these identifiers, the header file is included in both the resource script and the source-code file.

   The header file for the SAMPLE program, SAMPLE.H, is shown in Figure 17-13.

```
#define IDM_SCRIPT 1
#define IDM_MODERN 2
#define IDM_ROMAN  3
```

Figure 17-13. The SAMPLE.H header file.

4. Module-definition file: The module-definition file generally has a .DEF extension. The Windows linker uses this file in creating the executable .EXE file. The module-definition file specifies various attributes of the program's code and data segments, and it lists all imported and exported functions in the source-code file. In large programs that are divided into multiple code segments, the module-definition file allows the programmer to specify different attributes for each code segment.

The module-definition file for the SAMPLE program is named SAMPLE.DEF and is shown in Figure 17-14.

```
NAME           SAMPLE
DESCRIPTION    'Demonstration Windows Program'
STUB           'WINSTUB.EXE'
CODE           MOVABLE
DATA           MOVABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      4096
EXPORTS        WndProc
```

Figure 17-14. The SAMPLE.DEF module-definition file.

5. Make file: To facilitate construction of the executable file from these different components, Windows programmers often use the MAKE program to compile only those files that have changed since the last time the program was linked. To do this, the programmer first creates an ASCII text file called a make file. By convention, the make file has no extension.

The make file for the SAMPLE program is named SAMPLE and is shown in Figure 17-15. The programmer can create the SAMPLE.EXE executable file by executing

```
C>MAKE SAMPLE  <Enter>
```

A make file often contains several sections, each beginning with a target filename, followed by a colon and one or more dependent filenames, such as

```
sample.obj : sample.c sample.h
```

If either or both the SAMPLE.C and SAMPLE.H files have a later creation time than SAMPLE.OBJ, then MAKE runs the program or programs listed immediately below. In the case of the SAMPLE make file, the program is the C compiler, and it compiles the SAMPLE.C source code:

```
cl -c -Gsw -W2 -Zdp sample.c
```

Thus, if the programmer changes only one of the several files used in the development of SAMPLE, then running MAKE ensures that the executable file is brought up to date, while carrying out only the required steps.

```
sample.obj : sample.c sample.h
     cl -c -Gsw -W2 -Zdp sample.c

sample.res : sample.rc sample.h
     rc -r sample.rc

sample.exe : sample.obj sample.def sample.res
     link4 sample, /align:16, /map /line, slibw, sample
     rc sample.res
     mapsym sample
```

Figure 17-15. The make file for the SAMPLE program.

Construction of a Windows program

The make file shows the steps that create a program's .EXE file from the various components:

1. Compiling the source-code file:

   ```
   cl -c -Gsw -W2 -Zdp sample.c
   ```

   This step uses the CL.EXE C compiler to create a .OBJ object-module file. The command line switches are
   - -c: Compiles the program but does not link it. Windows programs must be linked with Windows' LINK4 linker, rather than with the LINK program the C compiler would normally invoke.
   - -Gsw: Includes two switches, -Gs and -Gw. The -Gs switch removes stack checks from the program. The -Gw switch inserts special prologue and epilogue code in all far functions defined in the program. This special code is required for Windows' memory management.
   - -W2: Compiles with warning level 2. This is the highest warning level, and it causes the compiler to display messages for conditions that may be acceptable in normal C programs but that can cause serious errors in a Windows program.
   - -Zdp: Includes two switches, -Zd and -Zp. The -Zd switch includes line numbers in the .OBJ file — helpful for debugging at the source-code level. The -Zp switch packs structures on byte boundaries. The -Zp switch is required, because data structures used within Windows are in a packed format.
2. Compiling the resource script:

   ```
   rc -r sample.rc
   ```

   This step runs the resource compiler and converts the ASCII .RC resource script into a binary .RES form. The -r switch indicates that the resource script should be compiled but the resources should not yet be added to the program's .EXE file.
3. Linking the program:

   ```
   link4 sample, /align:16, /map /line, slibw, sample
   ```

   This step uses the special Windows linker, LINK4. The first parameter listed is the name of the .OBJ file. The /align:16 switch instructs LINK4 to align segments in the .EXE file on 16-byte boundaries. The /map and /line switches cause LINK4 to create a .MAP file that contains program line numbers — again, useful for debugging source code. Next, slibw is a reference to the SLIBW.LIB file, which is an import library that contains module names and ordinal numbers for all Windows functions. The last parameter, sample, is the program's module-definition file, SAMPLE.DEF.
4. Adding the resources to the .EXE file:

   ```
   rc sample.res
   ```

This step runs the resource compiler a second time, using the compiled resource file, SAMPLE.RES. This time, the resource compiler adds the resources to the .EXE file.

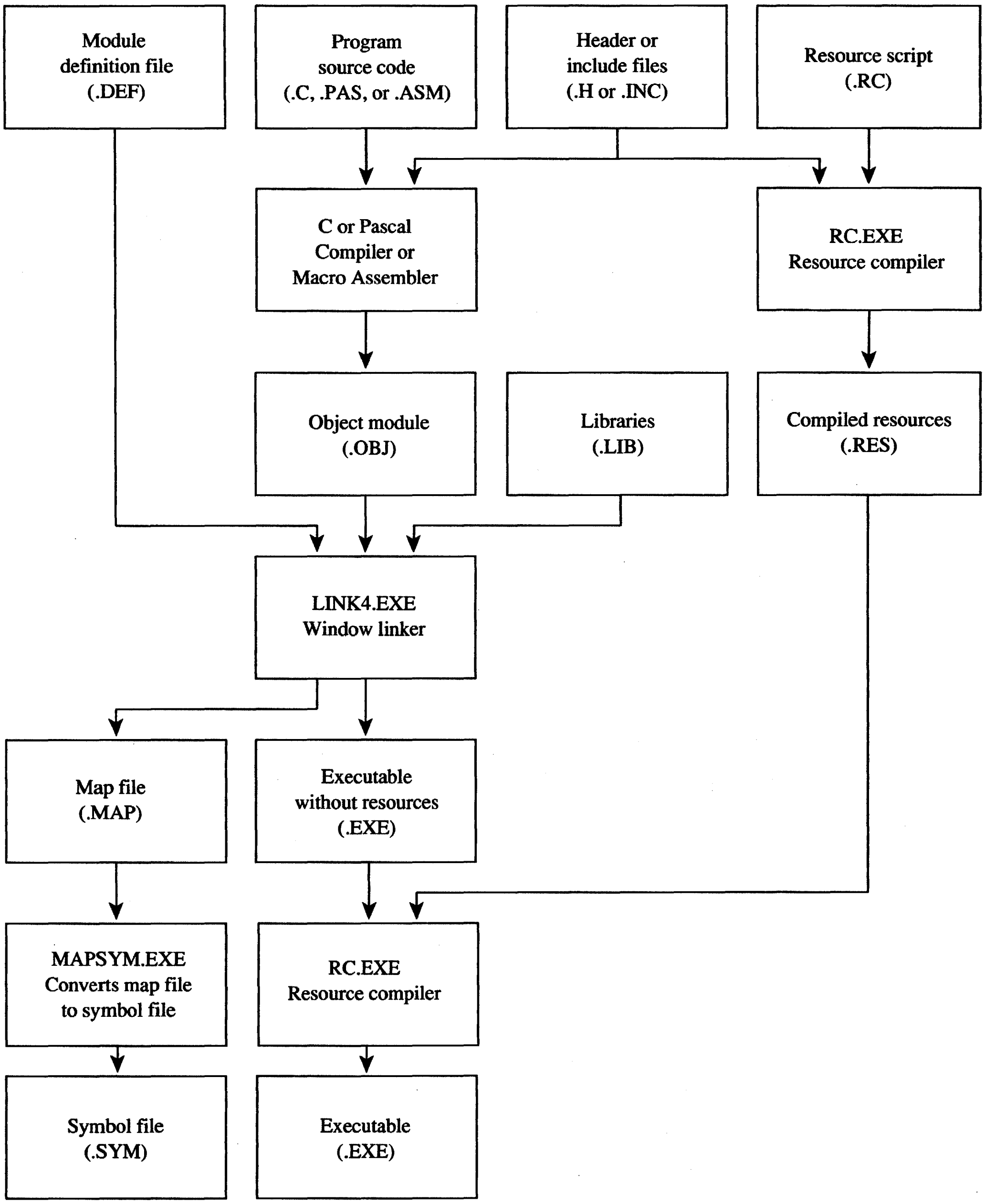

Module definition file (.DEF)
Program source code (.C, .PAS, or .ASM)
Header or include files (.H or .INC)
Resource script (.RC)
C or Pascal Compiler or Macro Assembler
RC.EXE Resource compiler
Object module (.OBJ)
Libraries (.LIB)
Compiled resources (.RES)
LINK4.EXE Window linker
Map file (.MAP)
Executable without resources (.EXE)
MAPSYM.EXE Converts map file to symbol file
RC.EXE Resource compiler
Symbol file (.SYM)
Executable (.EXE)

Figure 17-16. A block diagram showing the creation of a Windows .EXE file.

5. Creating a symbol (.SYM) file from the linker's map (.MAP) file:

   ```
   mapsym sample
   ```

   This step is required for symbolic debugging with SYMDEB.

Figure 17-16 on the preceding page shows how the various components of a Windows program fit into the creation of a .EXE file.

Program initialization

The SAMPLE.C program shown in Figure 17-11 contains some code that appears in almost every Windows program. The statement

```
#include <windows.h>
```

appears at the top of every Windows source-code file written in C. The WINDOWS.H file, provided with the Microsoft Windows Software Development Kit, contains templates for all Windows functions, structure definitions, and #define statements for many mnemonic identifiers.

Some of the variable names in SAMPLE.C may look unusual to C programmers because they begin with a prefix notation that denotes the data type of the variable. Windows programmers are encouraged to use this type of notation. Some of the more common prefixes are

Prefix	**Data Type**
i *or* n	Integer (16-bit signed integer)
w	Word (16-bit unsigned integer)
l	Long (32-bit signed integer)
dw	Doubleword (32-bit unsigned integer)
h	Handle (16-bit unsigned integer)
sz	Null-terminated string
lpsz	Long pointer to null-terminated string
lpfn	Long pointer to a function

The program's entry point (following some startup code) is the WinMain function, which is passed the following parameters: a handle to the current instance of the program (hInstance), a handle to the most recent previous instance of the program (hPrevInstance), a long pointer to the program's command line (lpszCmdLine), and a number (nCmdShow) that indicates whether the program should initially be displayed as a normally sized window or as an icon.

The first job SAMPLE performs in the WinMain function is to register a window class — a structure that describes characteristics of the windows that will be created in the class. These characteristics include background color, the type of cursor to be displayed in the window, the window's initial menu and icon, and the window function (the structure member called lpfnWndProc).

Multiple instances of a program can share the same window class, so SAMPLE registers the window class only for the first instance of the program:

```
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) ;
          }
```

The SAMPLE program then creates a window using the CreateWindow call, displays it to the screen by calling ShowWindow, and updates the client area by calling UpdateWindow:

```
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) ;
```

The first parameter to CreateWindow is the name of the window class. The second parameter is the actual text that appears in the window's title bar. The third parameter is the style of the window—in this case, the WINDOWS.H identifier WS_OVERLAPPEDWINDOW. The WS_OVERLAPPEDWINDOW is the most common window style. The fourth through seventh parameters specify the initial position and size of the window. The identifier CW_USEDEFAULT tells Windows to position and size the window according to the default rules.

After creating and displaying a Window, the SAMPLE program enters a piece of code called the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
return msg.wParam ;
```

This loop continues to execute until the GetMessage call returns zero. When that happens, the program instance terminates and the memory required for the instance is freed.

The Windows messaging system

Interactive programs written for the normal MS-DOS environment generally obtain user input only from the keyboard, using either an MS-DOS or a ROM BIOS software interrupt to check for keystrokes. When the user types something, such programs act on the keystroke and then return to wait for the next keystroke.

Programs written for Windows, however, can receive user input from a variety of sources, including the keyboard, the mouse, the Windows timer, menus, scroll bars, and controls, such as buttons and edit boxes.

Moreover, a Windows program must be informed of other events occurring within the system. For instance, the user of a Windows program might choose to make its window smaller or larger. Windows must make the program aware of this change so that the program can adjust its screen output to fit the new window size. Thus, for example, if a Windows program is minimized as an icon and the user maximizes its window to fill the full screen, Windows must inform the program that the size of the client area has changed and needs to be re-created.

Windows carries out this job of keeping a program informed of other events through the use of formatted messages. In effect, Windows sends these messages to the program. The Windows program receives and acts upon the messages.

This messaging makes the relationship between Windows and a Windows program much different from the relationship between MS-DOS and an MS-DOS program. Whereas MS-DOS does not provide information until a program requests it through an MS-DOS function call, Windows must continually notify a program of all the events that affect its window.

Window messages can be separated into two major categories: queued and nonqueued.

Queued messages are similar to the keyboard information an MS-DOS program obtains from MS-DOS. When the Windows user presses a key on the keyboard, moves the mouse, or presses one of the mouse buttons, Windows saves information about the event (in the form of a data structure) in the system message queue. Each message is destined for a particular window in a particular instance of a Windows program. Windows therefore determines which window should get the information and then places the message in the instance's own message queue.

A Windows program retrieves information from its queue in the message loop:

```
while (GetMessage (&msg, NULL, 0, 0))
        {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
        }
```

The *msg* variable is a structure. During the GetMessage call, Windows fills in the fields of this structure with information about the message. The fields are as follows:

- *hwnd*: The handle for the window that is to receive the message.
- *iMessage*: A numeric code identifying the type of message (for example, keyboard or mouse).
- *wParam*: A 16-bit value containing information specific to the message. *See* The Windows Messages below.
- *lParam*: A 32-bit value containing information specific to the message.
- *time*: The time, in milliseconds, that the message was placed in the queue. The time is a 32-bit value relative to the time at which the current Windows session began.
- *pt.x*: The horizontal coordinate of the mouse cursor at the time the event occurred.
- *pt.y*: The vertical coordinate of the mouse cursor at the time the event occurred.

GetMessage always returns a nonzero value except when it receives a quit message. The quit message causes the message loop to end. The program should then terminate and return control to Windows.

Within the message loop, the TranslateMessage function translates physical keystrokes into character-code messages. Windows places these translated messages into the program's message queue.

The DispatchMessage function essentially makes a call to the window function of the window specified by the hwnd field. This window function (WndProc in SAMPLE) is indicated in the lpfnWndProc field of the window class structure.

When DispatchMessage passes the message to the window function, Windows uses the first four fields of the message structure as parameters to the function. The window function can then process the message. In SAMPLE, for instance, the four fields passed to WndProc are *hwnd* (the handle to the window), *iMessage* (the numeric message identifier), *wParam*, and *lParam*. Although Windows does not pass the time and mouse-position information fields as parameters to the window function, this information is available through the Windows functions GetMessageTime and GetMessagePos.

A Windows program obtains only a few specific types of messages through its message queue. These are keyboard messages, mouse messages, timer messages, the paint message that tells the program it must re-create the client area of its window, and the quit message that tells the program it is being terminated.

In addition to the queued messages, however, a program's window function also receives many nonqueued messages. Windows sends these nonqueued messages by bypassing the message loop and calling the program's window function directly.

Many of these nonqueued messages are derived from queued messages. For example, when the user clicks the mouse on the menu bar, a mouse-click message is placed in the program's message queue. The GetMessage function retrieves the message and the DispatchMessage function sends it to the program's window function. However, because this mouse message affects a nonclient area of the window (an area outside the window's client area), the window function normally does not process it. Instead, the function passes the message back to Windows. In this example, the message tells Windows to invoke a pop-up menu. Windows calls up the menu and then sends the window function several nonqueued messages to inform the program of this action.

A Windows program is thus message driven. Once a program reaches the message loop, it acts only when the window function receives a message. And, although a program receives many messages that affect the window, the program usually processes only some of them, sending the rest to Windows for normal default processing.

The Windows messages

Windows can send a window function more than 100 different messages. The WINDOWS.H header file includes identifiers for all these messages so that C programmers do not have to remember the message numbers. Some of the more common messages and the meanings of the *wParam* and *lParam* parameters are discussed here:

WM_CREATE. Windows sends a window function this nonqueued message while processing the CreateWindow call. The *lParam* parameter is a pointer to a creation structure. A window function can perform some program initialization during the WM_CREATE message.

WM_MOVE. Windows sends a window function the nonqueued WM_MOVE message when the window has been moved to another part of the display. The *lParam* parameter gives the new coordinates of the window relative to the upper left corner of the screen.

WM_SIZE. This nonqueued message indicates that the size of the window has been changed. The new size is encoded in the *lParam* parameter. Programs often save this window size for later use.

WM_PAINT. This queued message indicates that a region in the window's client area needs repainting. (The message queue can contain only one WM_PAINT message.)

WM_COMMAND. This nonqueued message signals a program that a user has selected a menu item or has triggered a keyboard accelerator. Child-window controls also use WM_COMMAND to send messages to the parent window.

WM_KEYDOWN. The *wParam* parameter of this queued message is a virtual key code that identifies the key being pressed. The *lParam* parameter includes flags that indicate the previous key state and the number of keypresses the message represents.

WM_KEYUP. This queued message tells a window function that a key has been released. The *wParam* parameter is a virtual key code.

WM_CHAR. This queued message is generated from WM_KEYDOWN messages during the TranslateMessage call. The *wParam* parameter is the ASCII code of a keyboard key.

WM_MOUSEMOVE. Windows uses this queued message to tell a program about mouse movement. The *lParam* parameter contains the coordinates of the mouse relative to the upper left corner of the client area of the window. The *wParam* parameter contains flags that indicate whether any mouse buttons or the Shift or Ctrl keys are currently pressed.

*WM_*x*BUTTONDOWN.* This queued message tells a program that a button on the mouse was depressed while the mouse was in the window's client area. The *x* can be either L, R, or M for the left, right, or middle mouse button. The *wParam* and *lParam* parameters are the same as for WM_MOUSEMOVE.

*WM_*x*BUTTONUP.* This queued message tells a program that the user has released a mouse button.

*WM_*x*BUTTONDBLCLK.* When the user double-clicks a mouse button, Windows generates a WM_*x*BUTTONDOWN message for the first click and a queued WM_*x*BUTTONDBLCLK message for the second click.

WM_TIMER. When a Windows program sets a timer with the SetTimer function, Windows places a WM_TIMER message in the message queue at periodic intervals. The *wParam* parameter is a timer ID. (If the message queue already contains a WM_TIMER message, Windows does not add another one to the queue.)

WM_VSCROLL. A Windows program that includes a vertical scroll bar in its window receives nonqueued WM_VSCROLL messages indicating various types of scroll-bar manipulation.

WM_HSCROLL. This nonqueued message indicates a user is manipulating a horizontal scroll bar.

WM_CLOSE. Windows sends a window function this nonqueued message when the user has selected *Close* from the window's system menu. A program can query the user to determine whether any action, such as saving a file to disk, is needed before the program is terminated.

WM_QUERYENDSESSION. This nonqueued message indicates that the user is shutting down Windows by selecting *Close* from the MS-DOS Executive system menu. A program can request the user to verify that the program should be ended. If the window function returns a zero value from the message, Windows does not end the session.

WM_DESTROY. This nonqueued message is the last message a window function receives before the program ends. A window function can perform some last-minute cleanup while processing WM_DESTROY.

WM_QUIT. This is a queued message that never reaches the window function because it causes GetMessage to return a zero value that causes the program to exit the message loop.

Message processing

Programmers can choose to process some messages and ignore others in the window function. Messages that are ignored are generally passed on to the function Def WindowProc for default processing within Windows.

Because Windows eventually has access to messages that a window function does not process, it can send a program messages that might otherwise be regarded as pertaining to system functions—for example, mouse messages that occur in a nonclient area of the window, or system keyboard messages that affect the menu. Unless these messages are passed on to Def WindowProc, the menu and other system functions do not work properly.

A program can, however, trap some of these messages to override Windows' default processing. For example, when Windows needs to repaint the nonclient area of a window (the title bar, system-menu box, and scroll bars), it sends the window function a WM_NCPAINT (nonclient paint) message. The window function normally passes this message to DefWindowProc, which then calls routines to update the nonclient areas of the window. The program can, however, choose to process the WM_NCPAINT message and paint the nonclient area itself. A program that does this can, for example, draw its own scroll bars.

The Windows messaging system also notifies a program of important events occurring outside its window. For example, if the MS-DOS Executive were simply to end the Windows session when the user selects the *Close* option from its system menu, then applications that were still running would not have a chance to save changed files to disk. Instead, when the user selects *Close* from the last instance of the MS-DOS Executive's system menu, the MS-DOS Executive sends a WM_QUERYENDSESSION message to each currently running application. If any application responds by returning a zero value, the MS-DOS Executive does not end the Windows session.

Before responding, an application can process the WM_QUERYENDSESSION message and display a message box asking the user if a file should be saved. The message box should include three buttons labeled *Yes*, *No*, and *Cancel*. If the user answers *Yes*, the program can save the file and then return a nonzero value to the WM_QUERYENDSESSION message. If the user answers *No*, the program can return a nonzero value without saving the file. But if the user answers *Cancel*, the program should return a zero value so that the Windows session will not be ended. If a program does not process the WM_QUERYENDSESSION message, DefWindowProc returns a nonzero value.

When a user selects *Close* from the system menu of a particular instance of an application, rather than from the MS-DOS Executive's menu, Windows sends the window function a WM_CLOSE message. If the program has an unsaved file loaded, it can query the user with a message box—possibly the same one displayed when WM_QUERYENDSESSION is processed. If the user responds *Yes* to the query, the program can save the file and then call DestroyWindow. If the user responds *No*, the program can call DestroyWindow without saving the file. If the user responds *Cancel*, the window function does not call DestroyWindow and the program will not be terminated. If a program does not process WM_CLOSE messages, DefWindowProc calls DestroyWindow.

Finally, a window function can send messages to other window functions, either within the same program or in other programs, with the Windows SendMessage function. This function returns control to the calling program after the message has been processed. A program can also place messages in a program's message queue with the PostMessage function. This function returns control immediately after posting the message.

For example, when a program makes changes to the WIN.INI file (a file containing Windows initialization information), it can notify all currently running instances of these changes by sending them a WM_WININICHANGE message:

```
SendMessage (-1, WM_WININICHANGE, 0, 0L) ;
```

The -1 parameter indicates that the message is to be sent to all window functions of all currently running instances. Windows calls the window functions with the WM_WININICHANGE message and then returns control to the program that sent the message.

SAMPLE's message processing

The SAMPLE program shown in Figure 17-11 processes only four messages: WM_COMMAND, WM_SIZE, WM_PAINT, and WM_DESTROY. All other messages are passed to DefWindowProc. As is typical with most Windows programs written in C, SAMPLE uses a switch and case construction for processing messages.

The WM_COMMAND message signals the program that the user has selected a new font from the menu. SAMPLE first obtains a handle to the menu and removes the checkmark from the previously selected font:

```
hMenu = GetMenu (hWnd) ;
CheckMenuItem (hMenu, nCurrentFont, MF_UNCHECKED) ;
```

The value of *wParam* in the WM_COMMAND message is the menu ID of the newly selected font. SAMPLE saves that value in a static variable (nCurrentFont) and then places a checkmark on the new menu choice:

```
nCurrentFont = wParam ;
CheckMenuItem (hMenu, nCurrentFont, MF_CHECKED) ;
```

Because the typeface has changed, SAMPLE must repaint its display. The program does not repaint it immediately, however. Instead, it calls the InvalidateRect function:

```
InvalidateRect (hWnd, NULL, TRUE) ;
```

This causes a WM_PAINT message to be placed in the program's message queue. The NULL parameter indicates that the entire client area should be repainted. The TRUE parameter indicates that the background should be erased.

The WM_SIZE message indicates that the size of SAMPLE's client area has changed. SAMPLE simply saves the new dimensions of the client area in two static variables:

```
xClient = LOWORD (lParam) ;
yClient = HIWORD (lParam) ;
```

The LOWORD and HIWORD macros are defined in WINDOWS.H.

Windows also places a WM_PAINT message in SAMPLE's message queue when the size of the client area has changed. As is the case with WM_COMMAND, the program does not have to repaint the client area immediately, because the WM_PAINT message is in the message queue.

SAMPLE can receive a WM_PAINT message for many reasons. The first WM_PAINT message it receives results from calling UpdateWindow in the WinMain function. Later, if the current font is changed from the menu, the program itself causes a WM_PAINT message to be placed in the message queue by calling InvalidateRect. Windows also sends a window function a WM_PAINT message whenever the user changes the size of the window or when part of the window previously covered by another window is uncovered.

Programs begin processing WM_PAINT messages by calling BeginPaint:

```
BeginPaint (hWnd, &ps) ;
```

The SAMPLE program then creates a font based on the current size of the client area and the current typeface selected from the menu:

```
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]) ;
```

The font is selected into the device context (a data structure internal to Windows that describes the characteristics of the output device); the program also saves the original device-context font:

```
hFont = SelectObject (ps.hdc, hFont) ;
```

And the word *Windows* is displayed:

```
TextOut (ps.hdc, 0, 0, "Windows", 7) ;
```

The original font in the device context is then selected, and the font that was created is now deleted:

```
DeleteObject (SelectObject (ps.hdc, hFont)) ;
```

Finally, SAMPLE calls EndPaint to signal Windows that the client area is now updated and valid:

```
EndPaint (hWnd, &ps) ;
```

Although the processing of the WM_PAINT message in this program is simple, the method used is common to all Windows programs. The BeginPaint and EndPaint functions always occur in pairs, first to get information about the area that needs repainting and then to mark that area as valid.

SAMPLE will display this text even when the program is minimized to be displayed as an icon at the bottom of the screen. Although most Windows programs use a customized icon for this purpose, the window-class structure in SAMPLE indicates that the program's icon is NULL, meaning that the program is responsible for drawing its own icon. SAMPLE does not, however, make any special provisions for drawing the icon. To it, the icon is simply a small client area. As a result, SAMPLE displays the word *Windows* in its "icon," using a small font size.

Windows sends the window function the WM_DESTROY message as a result of the DestroyWindow function that DefWindowProc calls when processing a WM_CLOSE message. The standard processing involves placing a WM_QUIT message in the message queue:

```
PostQuitMessage (0) ;
```

When the GetMessage function retrieves WM_QUIT from the message queue, GetMessage returns 0. This terminates the message loop and the program.

For all other messages, SAMPLE calls Def WindowProc and exits the window function by returning the value from the call:

```
return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
```

This allows Windows to perform default processing on the messages SAMPLE ignores.

Windows' multitasking

Most operating systems or operating environments that allow multitasking use what is called a preemptive scheduler. Generally, the procedure involves use of the computer's clock to switch rapidly between programs and allow each a small time slice. When switching between programs, the operating system must preserve the machine state.

Windows is different. It is a nonpreemptive multitasking environment. Although Windows allows several programs to run simultaneously, it never switches from one program to allow another to run. It switches between programs only when the currently running program calls the GetMessage function or the related PeekMessage and WaitMessage functions.

When a Windows program calls GetMessage and the program's message queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the program with the next message. However, if the program's message queue contains only a WM_PAINT or WM_TIMER message and another program's queue contains a message other than WM_PAINT or WM_TIMER, Windows returns control to the other program, which is also waiting for its GetMessage call to return.

(Windows also switches between programs temporarily when a program uses SendMessage to send a message to a window function in another program, but control returns to the calling program after the window function has processed the message sent to it.)

To cooperate with Windows' nonpreemptive multitasking, programmers should try to perform message processing as quickly as possible. Programs can, for example, split a large amount of processing into several smaller pieces to allow other programs to run in the interval. During long processing a program can also periodically call PeekMessage to allow other programs to run.

Graphics Device Interface

Programs receive input through the Windows message system. For program output, Windows provides a device-independent interface to graphics output devices, such as the video display, printers, and plotters. This interface is called the Graphics Device Interface, or GDI.

The device context (DC)

When a Windows program needs to send output to the video screen, the printer, or another graphics output device, it must first obtain a handle to the device's device context, or DC. Windows provides a number of functions for obtaining this device-context handle:

BeginPaint. Used for obtaining a video device-context handle during processing of a WM_PAINT message. This device context applies only to the rectangular section of the client area that is invalid (needs repainting). This region is also a clipping region, meaning that a program cannot paint outside this rectangle. BeginPaint fills in the fields of a PAINTSTRUCT structure. This structure contains the coordinates of the invalid rectangle and a byte that indicates if the background of the invalid rectangle has been erased.

GetDC. Generally used for obtaining a video device-context handle during processing of messages other than WM_PAINT. The handle obtained with this function references only the client area of the window.

GetWindowDC. Used for obtaining a video device-context handle that encompasses the entire window, including the title bar, menu bar, and scroll bars. A Windows program can use this function if it is necessary to paint over areas of the window outside the client area.

CreateDC. Used for obtaining a device-context handle for the entire display or for a printer, a plotter, or other graphics output device.

CreateIC. Used for obtaining an information-context handle, which is similar to a device-context handle but can be used only for obtaining information about the output device, not for drawing.

CreateCompatibleDC. Used for obtaining a device-context handle to a memory device context compatible with a particular graphics output device. This function is generally used for transferring bitmaps to a graphics output device.

CreateMetaFile. Used for obtaining a metafile device-context handle. A metafile is a collection of GDI calls encoded in binary form.

The Windows program uses the device-context handle when calling GDI functions. In addition to drawing, the various GDI functions can change the attributes of the device context, select different drawing objects (such as pens and fonts) into the device context, and determine the characteristics of the device context.

Device-independent programming

Windows supports such a wide variety of video displays, printers, and plotters that programs cannot make assumptions about the size and resolution of the device. Furthermore, because the user can generally alter the size of a program's window, the program must be able to adjust its output appropriately. The SAMPLE program, for example, showed how the window function can use the WM_SIZE message to obtain the current size of a window to create a font that fits text within the window's client area.

Programs can also use other Windows functions to determine the physical characteristics of a device. For instance, a program can use the GetDeviceCaps function to obtain information about the device context, including the resolution of the device, its physical dimensions, and its relative pixel height and width.

Then, too, the GetTextMetrics function returns information about the current font selected in the device context. In the default device context, this is the system font. Many Windows programs base the size of their display output on the size of a system-font character.

Device-context attributes

The device context includes attributes that define how the graphics output functions work on the device. When a program first obtains a handle to a device context, Windows sets these attributes to default values, but the program can change them. Some of these device-context attributes are as follows:

Pen. Windows uses the current pen for drawing lines. The default pen produces a solid black line 1 pixel wide. A program can change the pen color, change to a dotted or dashed line, or make the pen draw a solid line wider than 1 pixel.

Brush. Windows uses the current brush (sometimes called a pattern) for filling areas. A brush is an 8-pixel-by-8-pixel bitmap. The default brush is solid white. Programs can create colored brushes, hatched brushes, and customized brushes based on bitmaps.

Background color. Windows uses the background color to fill the spaces in and between characters when drawing text and to color the open areas in hatched brushstrokes and dotted or dashed pen lines. Windows uses the background color only if the background mode (another attribute of the display context) is opaque. If the background mode is transparent, Windows leaves the background unaltered. The default background color is white.

Text color. Windows uses this color for drawing text. The default is black.

Font. Windows uses the font to determine the shape of text characters. The default is called the system font, a fixed-pitch font that also appears in menus, caption bars, and dialog boxes.

Additional device-context attributes (such as mapping modes) are described in the following sections.

Mapping modes

Most GDI drawing functions in Windows have parameters that specify the coordinates or size of an object. For instance, the Rectangle function has five parameters:

```
Rectangle (hDC, x1, y1, x2, y2) ;
```

The first parameter is the handle to the device context. The others are

- *x1*: horizontal coordinate of the upper left corner of the rectangle.
- *y1*: vertical coordinate of the upper left corner of the rectangle.
- *x2*: horizontal coordinate of the lower right corner of the rectangle.
- *y2*: vertical coordinate of the lower right corner of the rectangle.

In the Rectangle and most other GDI functions, coordinates are logical coordinates, which are not necessarily the same as the physical coordinates (pixels) of the device. To translate logical coordinates into physical coordinates, Windows uses the current mapping mode.

In actuality, the mapping mode defines a transformation of coordinates between a window, which is defined in terms of logical coordinates, and a viewport, which is defined in terms of physical coordinates. For any mapping mode, a program can define separate window and viewport origins. The logical point defined as the window origin is then mapped to the physical point defined as the viewport origin. For some mapping modes, a program can also define window and viewport extents, which determine how the logical coordinates are scaled to the physical coordinates.

Windows programs can select one of eight mapping modes. The first six are sometimes called fully constrained, because the ratio between the window and viewport extents is fixed and cannot be changed.

In MM_TEXT, the default mapping mode, coordinates on the x axis increase from left to right, and coordinates on the y axis increase from the top downward. In the other five fully constrained mapping modes, coordinates on the x axis also increase from left to right, but coordinates on the y axis increase from the bottom upward. The six fully constrained mapping modes are

- *MM_TEXT*: Logical coordinates are the same as physical coordinates.
- *MM_LOMETRIC*: Logical coordinates are in units of 0.1 millimeter.
- *MM_HIMETRIC*: Logical coordinates are in units of 0.01 millimeter.
- *MM_LOENGLISH*: Logical coordinates are in units of 0.01 inch.
- *MM_HIENGLISH*: Logical coordinates are in units of 0.001 inch.
- *MM_TWIPS*: Logical coordinates are in units of 1/1440 inch. (These units are 1/20 of a typographic point, which is approximately 1/72 inch.)

The seventh mapping mode is called partially constrained, because a program can change the window and viewport extents but Windows adjusts the values to ensure that equal horizontal and vertical logical coordinates translate to equal horizontal and vertical physical dimensions:

- *MM_ISOTROPIC*: Logical coordinates represent the same physical distance on both the x and y axes.

The MM_ISOTROPIC mapping mode is useful for drawing circles and squares. The MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, and MM_TWIPS mapping modes are also isotropic, because equal logical coordinates map to the same physical dimensions on both axes.

The final mapping mode is sometimes called unconstrained because a program is free to set different window and viewport extents on the x and y axes.

- *MM_ANISOTROPIC*: Logical coordinates are mapped to arbitrarily scaled physical coordinates.

Functions for drawing

Windows includes several functions that programs can use to draw in the client area of a window. The most common of these functions are

SetPixel. Sets a point to a particular color.

LineTo. Draws a line from the current position to a point specified in the LineTo function. The current position is defined in the device context and can be altered before the call to LineTo with the MoveTo function, which changes the current position but does not draw anything. Windows uses the current pen and the current drawing mode (*see* below) for drawing the line.

Polyline. Draws multiple lines much like a series of LineTo calls but does not alter the current position on completion.

Rectangle. Draws a filled rectangle with a border. Parameters to the Rectangle function specify the coordinates of the upper left and lower right corners of the rectangle. Windows draws the border of the rectangle with the current pen and current drawing mode defined in the device context, just as if it were using the Polyline function then Windows fills the rectangle with the current brush defined in the device context.

Ellipse. Uses the same parameters as Rectangle but draws an ellipse within the rectangular area.

RoundRect. Draws a rectangle with rounded corners. Two parameters to this function define the height and width of an ellipse that Windows uses for drawing the rounded corners.

Polygon. Draws a polygon connecting a series of points and fills the enclosed areas in either an alternate or winding mode. The winding mode causes Windows to fill every area within the polygon. The alternate mode fills every other area. For a polygon that defines a five-pointed star, for instance, the center is filled if the mode is winding but is not filled if the mode is alternate.

Arc. Draws a curved line that is part of the circumference of an ellipse.

Chord. Similar to the Arc function, but Windows connects the beginning and ending points of the arc with a straight line. The area is filled with the current brush defined in the device context.

Pie. Similar to the Arc function, but Windows draws lines from the beginning and ending points of the arc to the center of the ellipse. The area is filled with the current brush defined in the device context.

TextOut. Writes text with the current font, text color, background color, and background mode (transparent or opaque).

Windows also includes other drawing functions for filling areas, formatting text, and transferring bitmaps.

Raster operations for pens

When Windows uses a pen to write to a device context, it must first determine which pixels of the destination are to be altered by the pen (the foreground) and which pixels will not be affected (the background). With dotted and dashed pens, the background — the space between the dots or dashes — is left unaltered if the drawing mode is transparent and is filled with the background color if the drawing mode is opaque.

When Windows alters the pixels of the destination that correspond to the foreground of the pen, the most obvious result is that the color of the current pen defined in the display context is used to color the destination. But this is not the only possible result. Windows also generalizes the process by using a logical operation to combine the pixels of the pen and the pixels of the destination.

This logical operation is defined by the drawing mode attribute of the device context. This drawing mode can be set to one of 16 binary raster operations (abbreviated ROP2).

The following table shows the 16 binary raster operation codes defined in WINDOWS.H. The column headed "Resultant Destination" shows how the destination changes, depending on the bit pattern of the pen and the bit pattern of the destination before the line is drawn. The words OR, AND, XOR, and NOT are the logical operations.

Binary Raster Operation	**Resultant Destination**
R2_BLACK	0
R2_COPYPEN	pen
R2_MERGEPEN	pen OR destination
R2_MASKPEN	pen AND destination
R2_XORPEN	pen XOR destination
R2_NOTCOPYPEN	NOT pen
R2_NOTMERGEPEN	NOT (pen OR destination)
R2_NOTMASKPEN	NOT (pen AND destination)
R2_NOTXORPEN	NOT (pen XOR destination)
R2_MERGEPENNOT	pen OR (NOT destination)
R2_MASKPENNOT	pen AND (NOT destination)
R2_MERGENOTPEN	(NOT pen) OR destination
R2_MASKNOTPEN	(NOT pen) AND destination
R2_NOP	destination
R2_NOT	NOT destination
R2_WHITE	1

The default drawing mode defined in a device context is R2_COPYPEN, which simply copies the pen to the destination. However, if the pen color is blue, the destination is red, and the drawing mode is R2_MERGEPEN, then the drawn line appears as magenta, which results from combining the pen and destination colors. If the pen color is blue, the destination is red, and the drawing mode is R2_NOTMERGEPEN, then the drawn line is green, because the blue pen and the red destination are combined into magenta, which Windows then inverts to make green.

Bit-block transfers

Windows also uses logical operations when transferring a rectangular pixel pattern (a bit block) from one device context to another or from one area of a device context to another area of the same device context.

While line drawing involves a logical combination of two sets of pixels (the pen and the destination), the bit-block transfer functions perform a logical combination of three sets of pixels: a source bitmap, a destination bitmap, and the brush currently selected in the destination device context. As shown in the preceding section, there are 16 different ROP2 drawing modes for all the possible combinations of two sets of pixels. The tertiary raster operations (abbreviated ROP3) for bit-block transfers require 256 different operations for all possible combinations.

Windows defines three functions for transferring rectangular pixel patterns: BitBlt (bit-block transfer), StretchBlt (stretch-block transfer), and PatBlt (pattern-block transfer). Of these three functions, StretchBlt is the most generalized. StretchBlt transfers a bitmap from a source device context to a destination device context. Function parameters specify the origin, width, and height of the bitmap. If the source and destination widths and heights are different, Windows stretches or compresses the bitmap appropriately. Negative values of widths and heights cause Windows to draw a mirror image of the bitmap.

The BitBlt function transfers a bitmap from a source device context to a destination device context, but the width and height of the source and destination must be the same. If the source and destination device contexts have different mapping modes, Windows uses StretchBlt instead.

In both BitBlt and StretchBlt, Windows performs a bit-by-bit logical operation with the bit block in the source device context, the bit block in the destination area of the destination device context, and the brush currently selected in the destination device context. Although Windows supports all 256 possible raster operations with these three bitmaps, only a few have been given WINDOWS.H identifiers:

Raster Operation	**Resultant Destination**
BLACKNESS	0
SRCCOPY	source
SRCAND	source AND destination
SRCPAINT	source OR destination

(more)

Raster Operation	Resultant Destination
SRCINVERT	source XOR destination
SRCERASE	source AND (NOT destination)
MERGEPAINT	source OR (NOT destination)
NOTSRCCOPY	NOT source
NOTSRCERASE	NOT (source OR destination)
DSTINVERT	NOT destination
PATCOPY	pattern
MERGECOPY	source AND pattern
PATINVERT	destination XOR pattern
PATPAINT	source OR (NOT destination) OR pattern
WHITENESS	1

The PatBlt function is similar to BitBlt and StretchBlt but performs a logical operation only between the currently selected brush and a destination area of the device context. Thus, only 16 raster operations can be used with PatBlt; these are equivalent to the binary raster operations used with line drawing.

Text and fonts

Windows supports file-based text fonts in two different formats: raster and vector. The raster fonts, such as Courier, Helvetica, and Times Roman, are defined by digital representations of the bit patterns of the characters. Font files usually contain several different sizes for each typeface. The vector fonts, such as Modern, Script, and Roman, are defined by points that are connected to form the letters and can be scaled to different sizes.

When using a device such as a printer, which has built-in fonts, Windows can also use these device-based fonts.

To specify a font, a Windows program uses the CreateFont function to create a logical font—a detailed description of the desired font. When this logical font is selected into a device context, Windows finds the actual font that best fits this description. In many cases, this match is not exact. The program can then call GetTextMetrics to determine the characteristics of the actual font that the device will use to display text.

Windows supports both fixed-width and variable-width fonts, as well as such attributes as italics, underlining, and boldfacing. Programs can also justify text with the GetTextExtent call, which obtains the width of a particular text string. The program can then insert extra spaces between words with SetTextJustification or it can insert extra spaces between letters with SetTextCharacterExtra.

Metafiles

As explained earlier, a metafile is a collection of GDI function calls stored in a binary coded form. A program can create a metafile by calling CreateMetaFile and giving it either an MS-DOS filename or NULL as a parameter. If CreateMetaFile is given an MS-DOS filename, Windows creates a disk-based metafile; if the parameter is NULL, Windows creates a metafile in memory. The CreateMetaFile call returns a handle to a metafile device context. Any GDI calls that reference this device-context handle become part of the metafile.

When the program calls CloseMetaFile, Windows closes the metafile device context and returns a handle to the metafile. The program can then "play" this metafile on another device context (such as the video display) without calling the GDI functions directly.

Metafiles provide a useful way to transfer device-independent pictures between programs.

Data Sharing and Data Exchange

Windows includes a variety of methods by which programs can share and exchange data. These methods are discussed in the following sections.

Sharing local data among instances

Multiple instances of the same program can share data in the static data area of the program's data segment. Later instances of a program can thus call GetInstanceData and copy configuration options established by the user in the first instance. Multiple instances of programs can also share resources, such as dialog-box templates.

The Windows Clipboard

The Windows Clipboard is a general-purpose mechanism that allows a user to transfer data from one program to another. Programs that support the Clipboard generally include a top-level menu item called *Edit,* which invokes a pop-up menu that offers at least these three options:

- *Cut*: Copies the current selection to the Clipboard and deletes the selection from the current program file.
- *Copy*: Copies the current selection to the Clipboard without deleting the selection from the current program file.
- *Paste*: Copies the contents of the Clipboard to the current program file.

The Clipboard can hold only one item at a time. A program can transfer data to the Clipboard through the function call SetClipboardData. With this function, the program passes the Clipboard a handle to a global memory block, which then becomes the property of the Clipboard. A program can access Clipboard data through the complementary function GetClipboardData.

The Clipboard supports several formats:

- Text: ASCII text; each line ends with a carriage return and linefeed, and the text is terminated with a NULL character.
- Bitmap: A collection of bits in the GDI bitmap format.

- Metafile Picture: A structure that contains a handle to a metafile along with other information suggesting the mapping mode and aspect ratio of the picture.
- SYLK: Microsoft's Symbolic Link format.
- DIF: Software Arts' Data Interchange Format.

Programs can also use the Clipboard for storing data in private formats.

Some programs, such as the CLIPBRD program included with Windows, can also become Clipboard viewers. Such programs receive a message whenever the contents of the Clipboard change.

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is a protocol that cooperating programs can use to exchange data without user intervention. DDE makes use of the facilities in Windows that enable programs to send messages among themselves.

In DDE, the program that needs data from another program is called the client. The client sends a WM_DDE_INITIATE message either to a dedicated server program or to all currently running programs. Parameters to the WM_DDE_INITIATE message are *atoms*, which are numbers referring to text strings. A server application that has the data the client needs sends a WM_DDE_ACK message back to the client. The client can then be more specific about the data it needs by sending the server a WM_DDE_ADVISE message. The server can then pass global memory handles to the client with the WM_DDE_DATA message.

Internationalization

Windows includes several features that ease the conversion and translation of programs for international markets. Among these features are keyboard drivers appropriate for many European languages and use of the ANSI character set, which provides a richer set of accented letters than does the character set resident in the IBM PC and compatibles.

Windows also includes several functions that assist in language-independent coding. The AnsiUpper and AnsiLower functions translate characters or strings to uppercase or lowercase in the full ANSI character set, rather than the more limited ASCII character set. In addition, the AnsiNext and AnsiPrev functions allow scanning of text strings that may contain 2 or more bytes per character.

Windows programmers can also help in program translation by defining all text strings used within the program as resources contained in the resource script file. Because the resource script file also contains menu templates and dialog-box templates, it thus becomes the only file that needs alteration when a foreign-language version of the program is created.

Charles Petzold

Part E
Programming Tools

Article 18
Debugging in the MS-DOS Environment

It is axiomatic that any program will need debugging at some time in its development cycle, and programs written to run under MS-DOS are no exception. This article provides an introduction to the debugging tools and techniques available to the serious programmer developing code in the MS-DOS environment. Space does not permit a thorough investigation of the philosophy, psychology, and science of debugging computer programs; instead, a brief and practical discussion of the basic debugging approaches is presented, along with some rules-of-thumb for choosing the best approach. Nor are the details of every single utility and command included in this article; these are described in detail in the reference sections of this volume. The commands and utility programs that are most useful for debugging are discussed and illustrated with examples and case histories that also serve as models for the various debugging methods.

The reader of this article is assumed to be a programmer with sufficient experience to understand an assembly-language program. The reader is also assumed to be familiar with MS-DOS—terms like FCB and PSP are not explained. A reader without this background in MS-DOS need not be deterred, however; these terms are thoroughly explained elsewhere in this book. Besides assembly language, examples in this article are written in Microsoft QuickBASIC and Microsoft C. A detailed knowledge of these languages is not required; the examples are short and straightforward.

The reader should also keep in mind that the examples given here are real but not necessarily realistic. To avoid the tedium that accompanies debugging, the examples have been designed to reveal their bugs fairly quickly. All the methods and techniques shown are accurate in detail but not always in scale. Most of the debugging examples presented here would require one-half to one hour of work. It is possible for real debugging sessions to last for hours or days, especially if the wrong approach or tool is chosen. One of the purposes of this article is to help the programmer choose the correct tool and, thus, to reduce the tedium.

The Programs

There are more than a dozen listings in this article. Some of them are correct and others contain errors for use in illustrating debugging techniques. Many of the programs serve as examples in multiple sections of the article. The following summary of the programs (Table 18-1) is given to avoid confusion and to provide a common location to consult for explanations of the programs.

Table 18-1. Summary of Example Programs.

Name:	EXP.BAS
Figure:	18-1
Status:	Incorrect — do not use.
Purpose:	Computes EXP(*x*) (the exponential of *x*) to a specified precision using an infinite series.

Compiling:	QB EXP;
	LINK EXP;

Parameters:	Prompts for value for *x* and a convergence criterion. Enter zero to quit.

Name:	EXP.BAS
Figure:	18-3
Status:	Correct version of Figure 18-1.
Purpose:	Computes EXP(*x*) (the exponential of *x*) to a specified precision using an infinite series.

Compiling:	QB EXP;
	LINK EXP;

Parameters:	Prompts for value for *x* and a convergence criterion. Enter zero to quit.

Name:	COMMSCOP.ASM
Figure:	18-4
Status:	Correct.
Purpose:	Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. COMMSCOP is a terminate-and-stay-resident (TSR) program.

Compiling:	MASM COMMSCOP;
	LINK COMMSCOP;
	EXE2BIN COMMSCOP.EXE COMMSCOP.COM
	DEL COMMSCOP.EXE

Parameters:	Installed by entering *COMMSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START
02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS

(more)

DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name: COMMSCMD.C
Figure: 18-5
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls.
C version.
COMPILING: MSC COMMSCMD;
LINK COMMSCMD;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name: COMMSCMD.BAS
Figure: 18-6
Status: Correct.
Purpose: Controls the COMMSCOP program by issuing Interrupt 60H calls.
QuickBASIC version.
Compiling: QB COMMSCMD;
LINK COMMSCMD USERLIB;
Parameters: Commands are issued by
COMMSCMD [[*cmd*][,*port*]]
where: *cmd* is the command to be executed:
STOP Stop trace
START Flush buffer and start trace
RESUME Resume a stopped trace
port is the COM port (1 = COM1, 2 = COM2)
If *cmd* is omitted, STOP is assumed; if *port* is omitted, 1 is assumed.

Name COMMDUMP.BAS
Figure: 18-7
Status: Correct.
Purpose: Produces a formatted dump of the communications trace buffer.

(more)

Compiling: QB COMMDUMP;
LINK COMMDUMP USERLIB;

Parameters: No parameters. When COMMDUMP is invoked, it formats and dumps the entire buffer.

Name: TESTCOMM.ASM

Figure: 18-9

Status: Incorrect — do not use.

Purpose: Provides test data for the COMMSCOP routine.

Compiling: MASM TESTCOMM;
LINK TESTCOMM;

Parameters: No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name: TESTCOMM.ASM

Figure: 18-10

Status: Correct version of Figure 18-9.

Purpose: Provides test data for the COMMSCOP routine.

Compiling: MASM TESTCOMM;
LINK TESTCOMM;

Parameters: No parameters. TESTCOMM reads data from the keyboard and writes to COM1 and reads COM1 data and displays it on the screen. Ctrl-C cancels.

Name: BADSCOP.ASM

Figure: 18-11

Status: Incorrect version of Figure 18-4 — do not use.

Purpose: Monitors the activity on a specified COM port and places a copy of all transmitted and received data in a RAM buffer. Each entry in the buffer is tagged to indicate whether the byte was sent by or received by the application program under test. Control is provided to start, stop, and resume tracing by means of a control interrupt. When tracing is stopped and resumed, a marker is left in the buffer. BADSCOP is a terminate-and-stay-resident (TSR) program.

Compiling: MASM BADSCOP;
LINK BADSCOP;
EXE2BIN BADSCOP.EXE BADSCOP.COM
DEL BADSCOP.EXE

Parameters: Installed by entering *BADSCOP*; no parameters for installation. The TSR is controlled by passing parameter data in registers with an Interrupt 60H call. The registers can have the following values:

AH:	Command:
00H	STOP
01H	FLUSH AND START

(more)

02H	RESUME TRACE
03H	RETURN TRACE BUFFER ADDRESS
DX:	COM port:
00H	COM1
01H	COM2

Interrupt 60H returns the following in response to function 3:

CX	Buffer count in bytes
DX	Segment address of buffer
BX	Offset address of buffer

Name:	UPPERCAS.C
Figure:	18-13
Status:	Incorrect — do not use.
Purpose:	Converts a fixed string to uppercase and prints it.

Compiling:	MSC /Zi UPPERCAS;
	LINK UPPERCAS /CO;

Parameters:	No parameters.

Name:	UPPERCAS.C
Figure:	18-14
Status:	Correct version of Figure 18-13.
Purpose:	Converts a fixed string to uppercase and prints it.

Compiling:	MSC /Zi UPPERCAS;
	LINK UPPERCAS /CO;

Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-16
Status:	Incorrect — do not use.
Purpose:	Displays a table of all displayable characters.

Compiling:	MSC /Zi ASCTBL;
	LINK ASCTBL /CO;

Parameters:	No parameters.

Name:	ASCTBL.C
Figure:	18-17
Status:	Correct version of Figure 18-16.
Purpose:	Displays a table of all displayable characters.

Compiling:	MSC /Zi ASCTBL;
	LINK ASCTBL /CO;

Parameters:	No parameters.

Debugging Tools and Techniques

MS-DOS provides a wide variety of tools to aid in the debugging process. Some are intended specifically for debugging. For example, the DEBUG program is delivered with MS-DOS and provides basic debugging aid; the more sophisticated SYMDEB is supplied with MASM, Microsoft's macro assembler; CodeView, a debugger for high-order languages, is supplied with Microsoft C, Microsoft Pascal, and Microsoft FORTRAN. Others are general MS-DOS services and features that are also useful in the debugging process.

Debugging, like programming, has aspects of both an art and a craft. The craft — the mechanical details of using the tools — is discussed both here and elsewhere in this volume, but the main subject of this article is the *art* of debugging — the choice of the correct tool, the best techniques to use in various situations, the methods of extracting the clues to the problem from a recalcitrant program.

Debugging a program is a form of puzzle solving. As with most intellectual detective work, the following rule applies:

Gather enough information and the solution will be obvious.

The craft of debugging involves gathering the data; the art lies in deciding which data to gather and in noticing when the solution has become obvious.

The methods of gathering data for debugging, listed in order of increasing difficulty and tediousness, fall into four major categories:

- Inspection and observation
- Instrumentation
- Use of software debugging monitors (DEBUG, SYMDEB, and CodeView)
- Use of hardware debugging aids

As mentioned above, part of the art of debugging is knowing which method to use. This is one of the most difficult aspects of debugging — so difficult, in fact, that even programmers with years of experience make mistakes. Many programmers have spent hours single-stepping through a program with DEBUG only to discover that the cause of the problem would have been obvious if they had given the program's output even a cursory check. The only universal rule for choosing the correct debugging method is

Try them all, starting with the simplest.

Inspection and observation

Inspection and observation is the oldest and, usually, the best method of program debugging. It is also the basis for all the other methods. The first step with this method, as with the others, is to gather all the pertinent materials. Program listings, file layouts, report layouts, and program design materials (such as algorithm descriptions and flowcharts) are all extremely valuable in the debugging process.

Desk-checking

Before a programmer can determine what a program is doing wrong, he or she must know the correct operation of the program. There was a time, when computers were rare and expensive resources, that programmers were encouraged not to run their programs until the programs had been thoroughly desk-checked. The desk-checking process involves sitting down with a listing, a hand calculator, and some sample data. The programmer then "plays computer," executing each line of the program manually and writing down on paper the results of each program step. This process is extremely slow and tedious. When the desk-checking is completed, however, the programmer not only has found most of the bugs in the program but also has become intimately familiar with the execution of the program and the values of the program variables at each step.

The advent of inexpensive yet powerful personal computers, combined with the rising cost of programmer time, has made complete desk-checking nearly obsolete. It is now cheaper to run the program and let the computer find the errors. However, the usefulness of the desk-checking technique remains. Many programmers find it helpful to manually execute those sections of a program that they suspect are causing trouble. Even if they don't find errors in the code, the insight they gain into the workings of the code and the values of the variables at each step can be invaluable when applying other debugging techniques.

The inspection-and-observation methodology

The basic technique of the inspection-and-observation method is simple: After gathering all the required materials, run the program and observe. Observe very carefully; events that seem insignificant may be the very clues needed to discover where the program is going astray. As the program executes, note whether each section performs correctly. Does the program clear the screen when it should? Does it ask for input when it should? Does it produce the correct results? Observable events are the debugger's milestones in the execution of the program. If the program clears the screen but writes purple asterisks instead of requesting input, then the problem lies somewhere after the program issues the Clear Screen command but before it writes the input prompt on the screen. At this point, the program listing and design data become important. Inspect the listing and examine the area after the last successful milestone and before the missing milestone. Look for a logic error in the code that could explain the observed data.

If the program produces printed reports, they may also be useful. Watch the screen and listen to the printer. Clues can sometimes be found in the order in which things happen. The light on the disk drive can be another indication of activity. See how disk activity coordinates with screen and printer events. Try to identify each section of the program from these clues. Then use this information to localize the inspection of the listing to isolate the erroneous code.

The values of data given in reports and on the screen can also give clues to what's going wrong. Examining the data and reconstructing the values used to compute it sometimes leads to inferences about data problems. Perhaps a variable was misspelled in the code or perhaps a data file is in the wrong format or has been corrupted. With this information, the bug can often be isolated. However, a very thorough knowledge of the program and its algorithms is required. *See* Desk-checking above.

MS-DOS provides four commands and filters that are useful in the collection and examination of data for debugging: TYPE, PRINT, FIND, and DEBUG. All these commands display the data in a file in some way. If the data is ASCII (displayable) characters, TYPE and PRINT can be used, with some help from FIND. Binary files can be examined and modified with the DEBUG utility. *See* USER COMMANDS: FIND; PRINT; TYPE; PROGRAMMING UTILITIES: DEBUG.

The TYPE command provides the simplest way to display ASCII data files. This method can be used to examine both input and output files. Checking the input files may uncover some bad (or unexpected) data that causes the program to malfunction; examining the output files will show whether calculations are being performed correctly and may help pinpoint the erroneous calculations if they are not.

The FIND utility is useful in locating specific data in a file. Using FIND is more accurate and definitely less tedious than examining the file manually using the TYPE command. The /N switch causes FIND to also display the relative line number of the matching line—information that is most useful in debugging.

Sometimes the data is too complex to be examined on the screen and printed copy is needed. The PRINT command will produce hard copy of an ASCII file as will the TYPE command if its output is redirected to the printer with the >PRN command-line parameter after the filename.

Not all data files contain pure ASCII data, and displaying such non-ASCII files requires a different approach. The TYPE command can be used, but nonprintable characters will produce garbage on the screen. This technique can still prove useful if the file has a large amount of ASCII data or if the records are regular and the ASCII information always appears at the same location, but no information can be gained about non-ASCII numeric data in such files. Note also that the entire file might not be displayed using TYPE because if TYPE encounters a byte containing 1AH (Control-Z), it assumes it has reached the end of the file and stops.

Clearly, a more useful tool for examining non-ASCII files would be a program that dumps the file in hexadecimal, with an appropriate translation of all displayable characters. Such programs exist in the public domain (through bulletin-board services, for instance) and, in any event, are not difficult to write. MS-DOS does not include a stand-alone file-dumping program among its standard commands and utilities, but the DEBUG program can be used, with minor inconvenience, to display files. This program is discussed in detail later in this article; for now, simply follow these instructions to use DEBUG as a file dumper. To load DEBUG and the program to be debugged, use the form

DEBUG [*drive*:][*path*]*filename.ext*

DEBUG will display a hyphen as a prompt. To see the contents of the file, enter *D* (the DEBUG Display Memory command) and press Enter. DEBUG will display the first 128 (80H) bytes of the file in hexadecimal and will also show any displayable characters.

To see the rest of the file, simply continue entering *D* until the desired area is found. Hard copy of the contents of the display can be made by using the PrtSc key (or Ctrl-PrtSc to print continuously). Note that the offset addresses for the bytes in the file begin at the value in the program's CS:IP registers, which can be viewed by using the Debug R (Display or Modify Registers) command. To obtain the true offsets, subtract CS:IP from the address shown.

The essence of the inspection-and-observation method is careful and thoughtful observation. The computer and the operating system can provide tools to aid in the collection of data, but the most important tool is the programmer's mind. By applying the logical skills they already possess to the observed data, programmers can usually avoid the more complex forms of debugging.

Instrumentation

Debugging by instrumentation is a traditional method that has been popular since programs were holes punched in cards. In general, this method consists of adding something to the program, either internally or externally, to report on the progress of program execution. Programmers call this added mechanism instrumentation because of its resemblance to the measuring instruments used in science and engineering. Instrumentation can be software, hardware, or a combination of both; it can be internal to the program or external to it. Internal instrumentation is always software, but external instrumentation may be either hardware or software.

Internal instrumentation

Internal instrumentation usually consists of display or print statements placed at strategic locations. Other signals to the user can be used if they are available. For instance, the system beeper can be sounded at key locations, perhaps in a coded sequence of beeps; if the device being debugged has lights that can be accessed by the program, these lights can be flashed at important locations in the program. Beeping and flashing do not, however, possess the information content usually required for debugging, so display or print statements are preferred.

The use of display or print statements to display key data and milestones on the screen or printer requires careful planning. First, apply the techniques of inspection and observation described in the previous section to determine the most probable points of failure. Then, if there is some doubt about what path execution is taking through the code, embed messages of the following types after key decision points:

BEGINNING SORT PHASE
ENDING PRINCIPAL CALCULATION
PROCESSING RECORD *XX*

A second way to use display or print statement instrumentation is to embed statements that display the data and interim values used for calculations. This technique can be extremely useful in finding problems related to the data being processed. Consider the QuickBASIC program in Figure 18-1 as an example. The program has no syntax errors and compiles cleanly, but it sometimes produces an incorrect answer.

```
'  EXP.BAS -- COMPUTE EXPONENTIAL WITH INFINITE SERIES
'
' ***********************************************************************
' *                                                                     *
' *  EXP                                                                *
' *                                                                     *
' *  This routine computes EXP(x) using the following infinite series:  *
' *                                                                     *
' *                       x    x^2   x^3   x^4   x^5                    *
' *         EXP(x) = 1 + --- + --- + --- + --- + --- + ...              *
' *                       1!    2!    3!    4!    5!                    *
' *                                                                     *
' *                                                                     *
' *  The program requests a value for x and a value for the convergence *
' *  criterion, C.  The program will continue evaluating the terms of   *
' *  the series until the difference between two terms is less than C.  *
' *                                                                     *
' *  The result of the calculation and the number of terms required to  *
' *  converge are printed.  The program will repeat until an x of 0 is  *
' *  entered.                                                           *
' *                                                                     *
' ***********************************************************************

'
'  Initialize program variables
'
INITIALIZE:
        TERMS = 1
        FACT = 1
        LAST = 1.E35
        DELTA = 1.E34
        EX = 1
'
'  Input user data
'
        INPUT "Enter number:  "; X
        IF X = 0 THEN END
        INPUT "Enter convergence criterion (.0001 for 4 places):  "; C

'
'  Compute exponential until difference of last 2 terms is < C
'
        WHILE ABS(LAST - DELTA) >= C
            LAST = DELTA
            FACT = FACT * TERMS
            DELTA = X^TERMS / FACT
            EX = EX + DELTA
            TERMS = TERMS + 1
        WEND
```

Figure 18-1. A routine to compute exponentials. *(more)*

```
'
'  Display answer and number of terms required to converge
'
        PRINT EX
        PRINT TERMS; "elements required to converge"
        PRINT

        GOTO INITIALIZE
```

Figure 18-1. Continued.

The purpose of the EXP.BAS program is to compute the exponential of a given number to a specified precision using an infinite series. The program computes the value of each term in the infinite series and adds it to a running total. To keep from executing forever, the program checks the difference between the last two elements computed and stops when this difference is less than the convergence criterion entered by the user.

When the program is run for several values, the following results are observed:

```
Enter number:  ? 1
Enter convergence criterion (.0001 for 4 places):  ? .0001
 2.718282
 10 elements required to converge

Enter number:  ? 1.5
Enter convergence criterion (.0001 for 4 places):  ? .0001
 4.481686
 11 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 5
 3 elements required to converge

Enter number:  ? 2.5
Enter convergence criterion (.0001 for 4 places):  ? .0001
 12.18249
 15 elements required to converge

Enter number:  ? 3
Enter convergence criterion (.0001 for 4 places):  ? .0001
 13
 4 elements required to converge

Enter number:  ? 0
```

Some of these numbers are incorrect. Table 18-2 shows the computed values and the correct values.

Table 18-2. The Computed Values Generated by EXP.BAS and the Expected Values.

x	Computed	Correct
1.0	2.718282	2.718282
1.5	4.481686	4.481689
2.0	5	7.389056
2.5	12.18249	12.18249
3.0	13	20.08554

Applying the methods from the first section of this article and observing the data quickly reveals a pattern. With the exception of 1, all whole numbers give incorrect results, but all numbers with fractions give results that are correct to the specified convergence criterion. Examination of the listing shows no obvious reason for this. The answer is there, but only an exceptionally intuitive numeric analyst would see it. Because no answer is obvious, the next step is to validate the only information available—that whole numbers produce errors and fractional ones do not. Repeating the first experiment with 2 and a number very close to 2 yields the following results:

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.38167
 13 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 5
 3 elements required to converge

Enter number:  ? 0
```

The outcome is the same—repeating the experiment with a number as near to 2 as the convergence criterion permits (1.9999) produces the same result. The error is indeed caused by the fact that the number is an integer.

Because no intuitive way can be found to solve the mystery by inspection, the programmer must turn to the next method in the hierarchy, instrumentation. The problem has something to do with the calculation of the terms of the series. Therefore, the section of the program that performs this calculation should be instrumented by placing PRINT statements inside the WHILE loop (Figure 18-2) to display all the intermediate values of the calculation.

```
     WHILE ABS(LAST - DELTA) >= C
           LAST = DELTA
           FACT = FACT * TERMS
           DELTA = X ^ TERMS / FACT
```

Figure 18-2. Instrumenting the WHILE loop.

(more)

```
          EX = EX + DELTA
          PRINT "TERMS="; TERMS; "EX="; EX; "FACT="; FACT; "DELTA="; DELTA;
          PRINT "LAST="; LAST
          TERMS = TERMS + 1
    WEND
```

Figure 18-2. Continued.

The print statements used in this WHILE loop are typical of the type used for instrumentation. The program makes no attempt at fancy formatting. The print statements simply identify each value with its variable name, allowing easy correlation of the data and the code in the listing. Repeating the experiment with 1.999 and 2 yields

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
TERMS= 1 EX= 2.999 FACT= 1 DELTA= 1.999 LAST= 1E+34
TERMS= 2 EX= 4.997001 FACT= 2 DELTA= 1.998 LAST= 1.999
TERMS= 3 EX= 6.328335 FACT= 6 DELTA= 1.331334 LAST= 1.998
TERMS= 4 EX= 6.993669 FACT= 24 DELTA= .6653343 LAST= 1.331334
TERMS= 5 EX= 7.25967 FACT= 120 DELTA= .2660006 LAST= .6653343
TERMS= 6 EX= 7.348292 FACT= 720 DELTA= 8.862254E-02 LAST= .2660006
TERMS= 7 EX= 7.373601 FACT= 5040 DELTA= 2.530806E-02 LAST= 8.862254E-02
TERMS= 8 EX= 7.379924 FACT= 40320 DELTA= 6.323853E-03 LAST= 2.530806E-02
TERMS= 9 EX= 7.381329 FACT= 362880 DELTA= 1.404598E-03 LAST= 6.323853E-03
TERMS= 10 EX= 7.38161 FACT= 3628800 DELTA= 2.807791E-04 LAST= 1.404598E-03
TERMS= 11 EX= 7.381661 FACT= 3.99168E+07 DELTA= 5.102522E-05 LAST= 2.807791E-04
TERMS= 12 EX= 7.38167 FACT= 4.790016E+08 DELTA= 8.499951E-06 LAST= 5.102522E-05
 7.38167
 13 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
TERMS= 1 EX= 3 FACT= 1 DELTA= 2 LAST= 1E+34
TERMS= 2 EX= 5 FACT= 2 DELTA= 2 LAST= 2
 5
 3 elements required to converge
```

Examination of the instrumentation printout for the two cases shows a drastically different pattern. The fractional number went through 13 iterations following the expected pattern; the whole number, however, quit on the third step. The loop is terminating prematurely. Why? Look at the values calculated for *DELTA* and *LAST* on the last complete step. They are the same, giving a difference of zero. Because this difference will always be less than the convergence criterion, the loop will always terminate early. A moment's reflection shows why. The numerator of the fraction for each term but the first in the infinite series is a power of the number entered; the denominator is a factorial, a product formed by multiplying successive integers. Because $n! = n*(n-1)!$, when an integer is raised to a power equal to itself and divided by the factorial of that integer the result will always be the same as the preceding term. That is what has happened here.

Now that the cause of the problem is found, it must be fixed. How can this problem be prevented? In this case, the problem is caused by a logic error. The programmer misread (or miswrote!) the algorithm and assumed that the criterion for termination was that the difference between the last two terms be less than the specified value. This is incorrect. Actually, the termination criterion should be that the difference between the forming EXP(*x*) and the last term be less than the criterion. To simplify, the last term itself must be less than the value specified. The correct program listing, including the new WHILE loop, is shown in Figure 18-3.

```
'  EXP.BAS -- COMPUTE EXPONENTIAL WITH INFINITE SERIES
'
' ***********************************************************************
' *                                                                     *
' *  EXP                                                                *
' *                                                                     *
' *  This routine computes EXP(x) using the following infinite series:  *
' *                                                                     *
' *                       x     x^2   x^3   x^4   x^5                    *
' *         EXP(x) = 1 + --- + --- + --- + --- + --- + ...              *
' *                       1!    2!    3!    4!    5!                    *
' *                                                                     *
' *                                                                     *
' *  The program requests a value for x and a value for the convergence *
' *  criterion, C.  The program will continue evaluating the terms of   *
' *  the series until the amount added with a term is less than C.     *
' *                                                                     *
' *  The result of the calculation and the number of terms required to  *
' *  converge are printed.  The program will repeat until an x of 0 is  *
' *  entered.                                                           *
' *                                                                     *
' ***********************************************************************

'
'  Initialize program variables
'
INITIALIZE:
        TERMS = 1
        FACT = 1
        DELTA = 1.E35
        EX = 1
'
'  Input user data
'
        INPUT "Enter number:  "; X
        IF X = 0 THEN END
        INPUT "Enter convergence criterion (.0001 for 4 places):  "; C

'
'  Compute exponential until difference of last 2 terms is < C
'
```

Figure 18-3. Corrected exponential calculation routine. *(more)*

```
        WHILE DELTA > C
            FACT = FACT * TERMS
            DELTA = X^TERMS / FACT
            EX = EX + DELTA
            TERMS = TERMS + 1
        WEND

'
'   Display answer and number of terms required to converge
'
        PRINT EX
        PRINT TERMS; "elements required to converge"
        PRINT

        GOTO INITIALIZE
```

Figure 18-3. Continued.

The program now produces the correct results within the limits of the specified accuracy:

```
Enter number:  ? 1.999
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.381661
 12 elements required to converge

Enter number:  ? 2
Enter convergence criterion (.0001 for 4 places):  ? .0001
 7.389047
 12 elements required to converge

Enter number:  ? 0
```

This example illustrates how easy it is to use internal instrumentation in high-order languages. Because these languages usually have simple formatted output commands, they require very little work to instrument. When these output commands are not available, however, more work may be required. For instance, if the program being debugged is in assembly language, it is possible that the code required to format and print internal data will be longer than the program being debugged. For this reason, internal instrumentation is rarely used on small and moderate assembly programs. However, large assembly programs and systems often already have print formatting routines built into them; in these cases, internal instrumentation may be as easy as with high-order languages.

External instrumentation

Sometimes it is difficult to use internal instrumentation with a program. If, for instance, the problem is timing related, adding print statements could cloud the problem or, worse yet, make it go away completely. This leaves the programmer in the frustrating position of having the problem only when the cause can't be seen and not having the problem when it can. A solution to this type of problem can sometimes be found by moving the instrumentation outside the program itself. There are two types of external instrumentation: hardware and software.

Hardware instrumentation consists of whatever logic analyzers, oscilloscopes, meters, lights, bells, or gongs are appropriate to the hardware and software under test. Hardware instrumentation is difficult to set up and tedious to use. It is, therefore, usually reserved for those problems directly associated with hardware. Such problems often arise when new software is being run on new hardware and no one is quite sure where the bugs are. Because most programmers reading this book are developing software on tried-and-true personal computer hardware and because most of those programmers are unlikely to have a logic analyzer costing several thousand dollars, we will skip over the use of hardware instrumentation for software debugging. If a logic analyzer must be used, the programmer should remember that the debugging philosophy and techniques discussed in this article can still be applied effectively.

MS-DOS provides a feature that is very useful in building external instrumentation software: the TSR, or terminate-and-stay-resident routine. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities. This feature of the operating system allows the programmer to build a monitoring routine that is, in essence, a part of the operating system and outside the application program. The TSR is loaded as a normal program, but instead of leaving the system when it is done, it remains intact in memory. The operating system provides no way to reexecute the program after it terminates, so most TSRs are interrupt driven.

Because TSRs exist outside the application program, they can be used to build external instrumentation devices. This independence allows them to perform monitoring functions without disturbing the logic flow of the application program. The only areas where interference is likely are those where the TSR and the program must use common resources. These conflicts typically involve timing but can also involve other resources, such as I/O devices, disk files, and MS-DOS resources, including environment space. Some of these problems are addressed in the next example.

The TSR type of external instrumentation software can prove useful in analyzing serial communications. Such an instrumentation program monitors the serial communication line and records all data. To detect protocol or timing problems, the program tags the recorded data so that transmitted data can be differentiated from received data. Hardware devices exist that plug into the serial port and perform both the monitoring and tagging function, but they are expensive and not always handy. Fortunately, this inexpensive piece of software instrumentation will serve in many cases.

Software interrupt calls are made with the INT instruction. Although their service routines must obey similar rules, these interrupts should not be confused with hardware interrupts caused by external hardware events. Software interrupts in MS-DOS are used by an application program to communicate with the operating system and, by extension in IBM systems, with the ROM BIOS. For example, on IBM PCs and compatibles, application programs can use software Interrupt 14H to communicate with the ROM BIOS serial port driver. The ROM BIOS routine, in turn, manages the hardware interrupts from the actual serial device. Thus, Interrupt 14H does not communicate directly with the hardware. All the programs in this article deal with software interrupts to the ROM BIOS and MS-DOS.

A program to trace the serial data flow must have access to the serial data, so such a program must replace the vector for Interrupt 14H with one that points to itself. The routine can then record all the serial data and pass it along through the serial port. Because the goal is to minimize the effect of this monitoring on the timing of the data, the method used for recording the data should be fast. This requirement rules out writing to a disk file, because unexpected delays can be introduced (and because doing disk I/O from an interrupt service routine under MS-DOS is difficult, if not impossible). Printing the data on paper is clearly too slow, and data displayed on the screen is too ephemeral. Thus, about the only thing that can be done with the data is to write it to RAM. Luckily, memory has become cheap and most personal computers have plenty.

Writing a routine that monitors and records serial data is not enough, however. The data must still flow through the serial port to and from the external serial device. Thus, the monitor program can have only temporary custody of the data and must pass it on to the serial interrupt service routine in the ROM BIOS. This is accomplished by using MS-DOS function calls to extract the address of the serial interrupt handler before the new vector is installed in its place. The process of intercepting interrupts and then passing the data on is known as "daisy-chaining" interrupt handlers. So long as such intercepting programs are careful to maintain the data and conditions upon entrance for subsequent routines (that is, so long as routines are well behaved; *see* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS), several interrupt handlers can be daisy-chained together with no detriment to processing. (Woe be unto the person who breaks the daisy chain — the results are annoying at best and unpredictable at worst.)

The serial monitoring program, as described so far, correctly collects and stores serial data and then passes it on to the serial port. This may be intellectually satisfying, but it is not of much use in the real world. Some way must be provided to control the program — to start collection, to stop collection, to pause and resume collection. Also, once data is collected, a control function must be provided that returns the number of bytes collected and their starting location, so that the data can be examined.

From all this, it is clear that a serial communications monitoring instrument must

1. Replace the Interrupt 14H vector with one pointing to itself.
2. Save the address of the old interrupt handler.
3. Collect the serial data, tag it as transmitted or received, and store it in RAM.
4. Pass the data on, in a completely transparent manner, to the old interrupt handler.
5. Provide some way to control data collection.

A program that meets all these criteria is shown in Figure 18-4. The COMMSCOP program has three major parts:

Procedure	Purpose
COMMSCOPE	Monitoring and tagging
CONTROL	External control
VECTOR_INIT	Interrupt vector initialization

The *COMMSCOPE* procedure provides the new Interrupt 14H handler that intercepts the serial I/O interrupts. The *CONTROL* procedure provides the external control needed to make the system work. The *VECTOR_INIT* procedure gets the old interrupt handler address, installs *COMMSCOPE* as the new interrupt handler, and installs the interrupt handler for the control interrupt.

```
        TITLE   COMMSCOP -- COMMUNICATIONS TRACE UTILITY
; ***********************************************************************
; *                                                                     *
; *  COMMSCOP --                                                        *
; *     THIS PROGRAM MONITORS THE ACTIVITY ON A SPECIFIED COMM PORT     *
; *     AND PLACES A COPY OF ALL COMM ACTIVITY IN A RAM BUFFER.  EACH   *
; *     ENTRY IN THE BUFFER IS TAGGED TO INDICATE WHETHER THE BYTE      *
; *     WAS SENT BY OR RECEIVED BY THE SYSTEM.                          *
; *                                                                     *
; *     COMMSCOP IS INSTALLED BY ENTERING                               *
; *                                                                     *
; *                     COMMSCOP                                        *
; *                                                                     *
; *     THIS WILL INSTALL COMMSCOP AND SET UP A 64K BUFFER TO BE USED   *
; *     FOR DATA LOGGING.  REMEMBER THAT 2 BYTES ARE REQUIRED FOR       *
; *     EACH COMM BYTE, SO THE BUFFER IS ONLY 32K EVENTS LONG, OR ABOUT *
; *     30 SECONDS OF CONTINUOUS 9600 BAUD DATA.  IN THE REAL WORLD,    *
; *     ASYNC DATA IS RARELY CONTINUOUS, SO THE BUFFER WILL PROBABLY    *
; *     HOLD MORE THAN 30 SECONDS WORTH OF DATA.                        *
; *                                                                     *
; *     WHEN INSTALLED, COMMSCOP INTERCEPTS ALL INT 14H CALLS.  IF THE  *
; *     PROGRAM HAS BEEN ACTIVATED AND THE INT IS EITHER SEND OR RE-    *
; *     CEIVE DATA, A COPY OF THE DATA BYTE, PROPERLY TAGGED, IS PLACED *
; *     IN THE BUFFER.  IN ANY CASE, DATA IS PASSED ON TO THE REAL      *
; *     INT 14H HANDLER.                                                *
; *                                                                     *
; *     COMMSCOP IS INVOKED BY ISSUING AN INT 60H CALL.  THE INT HAS    *
; *     THE FOLLOWING CALLING SEQUENCE:                                 *
; *                                                                     *
; *             AH -- COMMAND                                           *
; *                   0 -- STOP TRACING, PLACE STOP MARK IN BUFFER      *
; *                   1 -- FLUSH BUFFER AND START TRACE                 *
; *                   2 -- RESUME TRACE                                 *
; *                   3 -- RETURN COMM BUFFER ADDRESSES                 *
; *             DX -- COMM PORT (ONLY USED WITH AH = 1 or 2)            *
; *                   0 -- COM1                                         *
; *                   1 -- COM2                                         *
```

Figure 18-4. Communications trace utility. *(more)*

```
; *                                                                         *
; *     THE FOLLOWING DATA IS RETURNED IN RESPONSE TO AH = 3:               *
; *                                                                         *
; *             CX -- BUFFER COUNT IN BYTES                                 *
; *             DX -- SEGMENT ADDRESS OF THE START OF THE BUFFER            *
; *             BX -- OFFSET ADDRESS OF THE START OF THE BUFFER             *
; *                                                                         *
; *     THE COMM BUFFER IS FILLED WITH 2-BYTE DATA ENTRIES OF THE           *
; *     FOLLOWING FORM:                                                     *
; *                                                                         *
; *             BYTE 0 -- CONTROL                                           *
; *                  BIT 0 -- ON FOR RECEIVED DATA, OFF FOR TRANS.          *
; *                  BIT 7 -- STOP MARK -- INDICATES COLLECTION WAS         *
; *                              INTERRUPTED AND RESUMED.                   *
; *             BYTE 1 -- 8-BIT DATA                                        *
; *                                                                         *
; ***************************************************************************

CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
        ORG     100H                    ;TO MAKE A COMM FILE

INITIALIZE:
        JMP     VECTOR_INIT             ;JUMP TO THE INITIALIZATION
                                        ; ROUTINE WHICH, TO SAVE SPACE,
                                        ; IS IN THE COMM BUFFER

;
;  SYSTEM VARIABLES
;
OLD_COMM_INT    DD      ?               ;ADDRESS OF REAL COMM INT
COUNT           DW      0               ;BUFFER COUNT
COMMSCOPE_INT   EQU     60H             ;COMMSCOPE CONTROL INT
STATUS          DB      0               ;PROCESSING STATUS
                                        ; 0 -- OFF
                                        ; 1 -- ON
PORT            DB      0               ;COMM PORT BEING TRACED
BUFPNTR         DW      VECTOR_INIT     ;NEXT BUFFER LOCATION

        SUBTTL  DATA INTERRUPT HANDLER
PAGE
; ***************************************************************************
; *                                                                         *
; *  COMMSCOPE                                                              *
; *     THIS PROCEDURE INTERCEPTS ALL INT 14H CALLS AND LOGS THE DATA        *
; *     IF APPROPRIATE.                                                     *
; *                                                                         *
; ***************************************************************************
COMMSCOPE       PROC    NEAR

        TEST    CS:STATUS,1             ;ARE WE ON?
        JZ      OLD_JUMP                ; NO, SIMPLY JUMP TO OLD HANDLER
```

Figure 18-4. Continued. *(more)*

```
        CMP     AH,00H                  ;SKIP SETUP CALLS
        JE      OLD_JUMP                ; .

        CMP     AH,03H                  ;SKIP STATUS REQUESTS
        JAE     OLD_JUMP                ; .

        CMP     AH,02H                  ;IS THIS A READ REQUEST?
        JE      GET_READ                ; YES, GO PROCESS

;
;  DATA WRITE REQUEST -- SAVE IF APPROPRIATE
;
        CMP     DL,CS:PORT              ;IS WRITE FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .
        MOV     BX,BUFPNTR              ;GET ADDR OF NEXT BUFFER LOC
        MOV     [BX],BYTE PTR 0         ;MARK AS TRANSMITTED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     WRITE_DONE              ;ZERO MEANS BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF
WRITE_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        JMP     OLD_JUMP                ;PASS REQUEST ON TO BIOS ROUTINE
;
;  PROCESS A READ DATA REQUEST AND WRITE TO BUFFER IF APPROPRIATE
;
GET_READ:
        CMP     DL,CS:PORT              ;IS READ FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .

        PUSHF                           ;FAKE INT 14H CALL
        CLI                             ; .
        CALL    OLD_COMM_INT            ;PASS REQUEST ON TO BIOS
        TEST    AH,80H                  ;VALID READ?
        JNZ     READ_DONE               ; NO, SKIP BUFFER UPDATE
```

Figure 18-4. Continued. *(more)*

```
        MOV     BX,BUFPNTR              ;GET ADDR OF NEXT BUFFER LOC
        MOV     [BX],BYTE PTR 1         ;MARK AS RECEIVED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     READ_DONE               ;ZERO MEANS BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF
READ_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        IRET

;
;  JUMP TO COMM BIOS ROUTINE
;
OLD_JUMP:
        JMP     CS:OLD_COMM_INT

COMMSCOPE ENDP

        SUBTTL  CONTROL INTERRUPT HANDLER
PAGE
; ***********************************************************************
; *                                                                     *
; *  CONTROL                                                            *
; *     THIS ROUTINE PROCESSES CONTROL REQUESTS.                        *
; *                                                                     *
; ***********************************************************************

CONTROL PROC    NEAR
        CMP     AH,00H                  ;STOP REQUEST?
        JNE     CNTL_START              ; NO, CHECK START
        PUSH    DS                      ;SAVE REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET DS FOR OUR ROUTINE
        POP     DS
        MOV     STATUS,0                ;TURN PROCESSING OFF
        MOV     BX,BUFPNTR              ;PLACE STOP MARK IN BUFFER
        MOV     [BX],BYTE PTR 80H       ; .
        MOV     [BX+1],BYTE PTR 0FFH    ; .
        INC     BX                      ;INCREMENT BUFFER POINTER
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ; .
        INC     COUNT                   ;INCREMENT COUNT
        INC     COUNT                   ; .
        POP     BX                      ;RESTORE REGISTERS
        POP     DS                      ; .
        JMP     CONTROL_DONE
```

Figure 18-4. Continued.

(more)

```
CNTL_START:
        CMP     AH,01H                  ;START REQUEST?
        JNE     CNTL_RESUME             ; NO, CHECK RESUME
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:BUFPNTR,OFFSET VECTOR_INIT ;RESET BUFFER TO START
        MOV     CS:COUNT,0              ;ZERO COUNT
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_RESUME:
        CMP     AH,02H                  ;RESUME REQUEST?
        JNE     CNTL_STATUS             ; NO, CHECK STATUS
        CMP     CS:BUFPNTR,0            ;END OF BUFFER CONDITION?
        JE      CONTROL_DONE            ; YES, DO NOTHING
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_STATUS:
        CMP     AH,03H                  ;RETURN STATUS REQUEST?
        JNE     CONTROL_DONE            ; NO, ERROR -- DO NOTHING
        MOV     CX,CS:COUNT             ;RETURN COUNT
        PUSH    CS                      ;RETURN SEGMENT ADDR OF BUFFER
        POP     DX                      ; .
        MOV     BX,OFFSET VECTOR_INIT   ;RETURN OFFSET ADDR OF BUFFER

CONTROL_DONE:
        IRET

CONTROL ENDP


        SUBTTL     INITIALIZE INTERRUPT VECTORS
PAGE
; ************************************************************************
; *                                                                      *
; *  VECTOR_INIT                                                         *
; *     THIS PROCEDURE INITIALIZES THE INTERRUPT VECTORS AND THEN       *
; *     EXITS VIA THE MS-DOS TERMINATE-AND-STAY-RESIDENT FUNCTION.      *
; *     A BUFFER OF 64K IS RETAINED.  THE FIRST AVAILABLE BYTE          *
; *     IN THE BUFFER IS THE OFFSET OF VECTOR_INIT.                     *
; *                                                                      *
; ************************************************************************

        EVEN                            ;ASSURE BUFFER ON EVEN BOUNDARY
VECTOR_INIT     PROC     NEAR
;
;  GET ADDRESS OF COMM VECTOR (INT 14H)
;
        MOV     AH,35H
```

Figure 18-4. Continued.

(more)

```
        MOV     AL,14H
        INT     21H
;
;  SAVE OLD COMM INT ADDRESS
;
        MOV     WORD PTR OLD_COMM_INT,BX
        MOV     AX,ES
        MOV     WORD PTR OLD_COMM_INT[2],AX
;
;  SET UP COMM INT TO POINT TO OUR ROUTINE
;
        MOV     DX,OFFSET COMMSCOPE
        MOV     AH,25H
        MOV     AL,14H
        INT     21H
;
;  INSTALL CONTROL ROUTINE INT
;
        MOV     DX,OFFSET CONTROL
        MOV     AH,25H
        MOV     AL,COMMSCOPE_INT
        INT     21H
;
;  SET LENGTH TO 64K, EXIT AND STAY RESIDENT
;
        MOV     AX,3100H                  ;TERM AND STAY RES COMMAND
        MOV     DX,1000H                  ;64K RESERVED
        INT     21H                       ;DONE
VECTOR_INIT ENDP
CSEG    ENDS
        END     INITIALIZE
```

Figure 18-4. Continued.

The first executable statement of the program is a jump to the *VECTOR_INIT* procedure. The vector initialization code is needed only during installation; after initialization of the vectors, the code can be discarded. In this case, the area where this code resides will become the start of the trace buffer; therefore, it makes sense to put the initialization code at the end of the program where it can be overlaid by the trace buffer. The jump at the start of the program is required because the rules for making .COM files require that the entry point be the first instruction of the program.

The vector initialization routine uses Interrupt 21H Function 35H (Get Interrupt Vector) to get the address of the current Interrupt 14H service routine. The segment and offset address (returned in the ES:BX registers) is stored in the doubleword at *OLD_COMM_INT.* Interrupt 21H Function 25H (Set Interrupt Vector) is then used to vector all Interrupt 14H calls to *COMMSCOPE.* Another Function 25H call sets Interrupt 60H to vector to the *CONTROL* routine. This interrupt, which provides the means to control and interrogate the *COMMSCOPE* routine, was chosen because it is unused by MS-DOS and because some IBM technical materials list 60H through 66H as being available for user interrupts. (If, for some reason, Interrupt 60H is not available, simply change the equated symbol *COMMSCOPE_INT* to an available interrupt.)

When the vector initialization process is complete, the routine exits and stays resident by using Interrupt 21H Function 31H (Terminate and Stay Resident). As part of the termination process, the routine requests 1000H paragraphs, or 64 KB, of storage. A little over 500 bytes of this storage area is used for the code; the rest is available for trace data. If the serial port is running at 2400 baud, a solid stream of data will fill this buffer in about two minutes. However, a solid 32 KB block of data is unusual in asynchronous communications and, in reality, the buffer will usually contain many minutes worth of data. Note that the buffer-handling routines in *COMMSCOPE* require that the buffer be aligned on an even byte boundary, so *VECTOR_INIT* is preceded by the EVEN directive.

The interrupt service routine, *COMMSCOPE*, receives all Interrupt 14H calls. First *COMMSCOPE* checks its own status. If it has not been activated, it immediately passes control to the real service routine. If the tracer is active, *COMMSCOPE* examines the Interrupt 14H function in AH. Setup and status requests (AH = 0 and AH = 3) do not affect tracing, so they are passed on directly to the the real service routine. If the Interrupt 14H call is a write-data request (AH = 1), *COMMSCOPE* moves the byte marking the data as transmitted and the data byte itself to the current buffer location and increments both the byte count and the buffer pointer by 2. If the buffer pointer goes to zero, the buffer has wrapped; data collection is turned off and cannot be turned on again without clearing the trace buffer. Because the buffer, which starts at *VECTOR_INIT*, is always on an even byte boundary, there is no danger of the first byte of the data pair forcing a wrap. After the transmitted data is added to the buffer, *COMMSCOPE* passes control to the real service routine.

A read-data request (AH = 2) must be handled a little differently. In this case, the data to be collected is not yet available. In order to get it, *COMMSCOPE* must pass control to the real service routine and then intercept the results on the way back. The code at *GET_READ* fakes an interrupt to the service routine by pushing the flags onto the stack so that the service routine's IRET will pop them off again. *COMMSCOPE* then calls the service routine and, when it returns, retrieves the incoming serial data character from AL. If the incoming data byte is valid (bit 7 of AH is zero), the byte marking the data as received and the data byte itself are placed in the trace buffer, and both the byte count and the buffer pointer are incremented by 2. The buffer-wrap condition is detected and handled in the same manner as with transmitted data. Because the real service routine has already been called, *COMMSCOPE* exits as if it were the service routine by issuing an IRET.

The *CONTROL* procedure provides the mechanism for external control of the trace procedure. The routine is entered whenever an Interrupt 60H is executed. Commands are sent through the AH register and can cause the routine to STOP (AH = 0), START/FLUSH (AH = 1), RESUME (AH = 2), or RETURN STATUS (AH = 3). This routine also sets the communications port to be traced. The required information is provided in DX using the same format as the Interrupt 14H routine. The port information is used only with START and RESUME requests. The RETURN STATUS command returns data in registers: the byte count (CX), the segment address of the buffer (DX), and the offset of the first byte in the buffer (BX).

The COMMSCOP program is assembled using the Microsoft Macro Assembler (MASM), linked using the Microsoft Object Linker (LINK), and then converted to a .COM file using EXE2BIN (*see* PROGRAMMING UTILITIES):

```
C>MASM COMMSCOP;  <Enter>
C>LINK COMMSCOP;  <Enter>
C>EXE2BIN COMMSCOP.EXE COMMSCOP.COM  <Enter>
C>DEL COMMSCOP.EXE  <Enter>
```

The linker will display the message *Warning: no stack segment*; this message can be ignored because the rules for making a .COM file forbid a separate stack segment.

The program is installed by simply typing *COMMSCOP.* Tracing can then be started and stopped using Interrupt 60H. MS-DOS does not allow resident routines to be removed, so COMMSCOP will be in the system until the system is restarted. Also note that, because COMMSCOP is well behaved, nothing disastrous will happen if multiple copies of it are accidentally installed. As each new copy is installed, it chains to the previous copy. When Interrupt 14H is intercepted, the new routine dutifully passes the data on to the previous routine, which repeats the process until the real service routine is reached. The data is added to the trace buffer of each copy, giving multiple, redundant copies of the same data. Because Interrupt 60H is not chained, only the last copy's buffer can be accessed. Thus, the other copies simply waste 64 KB each.

Two techniques can be used to start or stop a trace. The first is to issue Interrupt 60H calls at strategic locations within the program being debugged. With assembly-language programs, this is easy. The appropriate registers are loaded and an INT 60H instruction is executed. Issuing this INT instruction is not much more difficult with higher-order Microsoft languages — both QuickBASIC and C provide a library routine called INT86 that allows registers to be loaded and INT instructions to be executed. (In QuickBASIC, the INT86 library routine is included in the File USERLIB.OBJ; in Microsoft C, it is included in the file DOS.H.) Embedded Interrupt 60H calls can be convenient because they limit tracing to those areas where processing is suspect. Because COMMSCOP marks the buffer each time the trace is stopped and resumed, the separate pieces of a trace are easy to differentiate.

The second technique is to write a simple routine to start or stop the trace outside the program being debugged. The example in Figure 18-5, COMMSCMD, is a Microsoft C program that can perform these functions using the INT86 library function to issue Interrupt 60H calls.

```
/***********************************************************************
*                                                                      *
*  COMMSCMD                                                            *
*                                                                      *
*  This routine controls the COMMSCOP program that has been in-        *
*  stalled as a resident routine.  The operation performed is de-      *
*  termined by the command line.  The COMMSCMD program is invoked      *
*  as follows:                                                         *
*                                                                      *
*                COMMSCMD [[cmd][ port]]                               *
*                                                                      *
```

Figure 18-5. A serial-trace control routine written in C. *(more)*

```
*  where cmd is the command to be executed                              *
*            STOP   -- stop trace                                       *
*            START  -- flush trace buffer and start trace              *
*            RESUME -- resume a stopped trace                           *
*        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)      *
*                                                                       *
*  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is        *
*  assumed.                                                             *
*                                                                       *
************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#define COMMCMD 0x60

main(argc, argv)
int argc;
char *argv[];
{
    int cmd, port, result;
    static char commands[3] [10] = {"STOPPED", "STARTED", "RESUMED"};
    union REGS inregs, outregs;

    cmd = 0;
    port = 0;

    if (argc > 1)
        {
        if (0 == stricmp(argv[1], "STOP"))
            cmd = 0;
        else if (0 == stricmp(argv[1], "START"))
            cmd = 1;
        else if (0 == stricmp(argv[1], "RESUME"))
            cmd = 2;
        }

    if (argc == 3)
        {
        port = atoi(argv[2]);
        if (port > 0)
            port = port - 1;
        }

    inregs.h.ah = cmd;
    inregs.x.dx = port;
    result = int86(COMMCMD, &inregs, &outregs);


    printf("\nCommunications tracing %s for port COM%1d:\n",
            commands[cmd], port + 1);
}
```

Figure 18-5. Continued.

COMMSCMD is passed arguments in the command line. The first argument is the command to be performed: STOP, START, or RESUME. If no command is specified, STOP is assumed. The second argument is the port number: 1 (for COM1) or 2 (for COM2). If no port number is specified, 1 is assumed.

The COMMSCMD program uses a simple IF filter to determine the function to be performed. The program tests the number of arguments in the command line to see if a port has been specified. If the argument count (*argc*) is 3 (one for the command name, one for the command, and one for the port number), the port number argument is retrieved and converted to an integer. The Interrupt 60H routine expects port numbers to be specified in the same manner as for Interrupt 14H, so the port number is decremented if it is not already zero. The AH register is loaded with the command (*cmd*), the DX register is loaded with the port number (*port*), and the INT86 library function is then used to execute an Interrupt 60H call. When the interrupt returns, COMMSCMD displays a message showing the function and port.

The same function can be performed by the QuickBASIC program in Figure 18-6.

```
' ***********************************************************************
' *                                                                     *
' *  COMMSCMD                                                           *
' *                                                                     *
' *  This routine controls the COMMSCOP program that has been in-       *
' *  stalled as a resident routine.  The operation performed is de-     *
' *  termined by the command line.  The COMMSCMD program is invoked     *
' *  as follows:                                                        *
' *                                                                     *
' *             COMMSCMD [[cmd][,port]]                                 *
' *                                                                     *
' *  where cmd is the command to be executed                            *
' *             STOP   -- stop trace                                    *
' *             START  -- flush trace buffer and start trace            *
' *             RESUME -- resume a stopped trace                        *
' *        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)    *
' *                                                                     *
' *  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is      *
' *  assumed.                                                           *
' *                                                                     *
' ***********************************************************************

        '
        '  Establish system constants and variables
        '
        DEFINT A-Z

        DIM INREG(7), OUTREG(7)         'Define register arrays
```

Figure 18-6. A QuickBASIC version of COMMSCMD. *(more)*

```
        RAX = 0                         'Establish values for 8086
        RBX = 1                         '  registers
        RCX = 2                         '  .
        RDX = 3                         '  .
        RBP = 4                         '  .
        RSI = 5                         '  .
        RDI = 6                         '  .
        RFL = 7                         '  .

        DIM TEXT$(2)

        TEXT$(0) = "STOPPED"
        TEXT$(1) = "STARTED"
        TEXT$(2) = "RESUMED"

        '
        '  Process command-line tail
        '
        C$ = COMMAND$                   'Get command-line data

        IF LEN(C$) = 0 THEN             'If no command line specified
            CMD = 0                     'Set CMD to STOP
            PORT = 0                    'Set PORT to COM1
            GOTO SENDCMD
        END IF

        COMMA = INSTR(C$, ",")          'Extract operands
        IF COMMA = 0 THEN
            CMDTXT$ = C$
            PORT = 0
        ELSE
            CMDTXT$ = LEFT$(C$, COMMA - 1)
            PORT = VAL(MID$(C$, COMMA + 1)) - 1
        END IF

        IF PORT < 0 THEN PORT = 0

        IF CMDTXT$ = "STOP" THEN
            CMD = 0
        ELSEIF CMDTXT$ = "START" THEN
            CMD = 1
        ELSEIF CMDTXT$ = "RESUME" THEN
            CMD = 2
        ELSE
            CMD = 0
        END IF

        '
        '  Send command to COMMSCOP routine
        '
SENDCMD:
        INREG(RAX) = 256 * CMD
```

Figure 18-6. Continued. *(more)*

```
        INREG(RDX) = PORT
        CALL INT86(&H60, VARPTR(INREG(0)), VARPTR(OUTREG(0)))
        '
        '  Notify user that action is complete
        '
        PRINT : PRINT
        PRINT "Communications tracing "; TEXT$(CMD);
        IF CMD <> 0 THEN
            PRINT " for port COM"; MID$(STR$(PORT + 1), 2); ":"
        ELSE
            PRINT
        END IF

        END
```

Figure 18-6. Continued.

Both versions of COMMSCMD accept their commands from the command tail; both are invoked with a STOP, START, or RESUME command and a serial port number (1 or 2). If the operands are omitted, STOP and COM1 are assumed.

After data has been collected and safely placed in the trace buffer, it must be read before it can be useful. Interrupt 60H provides a function (AH = 3) that returns the buffer address and the number of bytes in the buffer. The QuickBASIC routine in Figure 18-7 uses this function to get the address of the data and then formats the data on the screen.

```
' *********************************************************************
' *                                                                   *
' *  COMMDUMP                                                         *
' *                                                                   *
' *  This routine dumps the contents of the COMMSCOP trace buffer to  *
' *  the screen in a formatted manner.  Received data is shown in     *
' *  reverse video.  Where possible, the ASCII character for the byte *
' *  is shown; otherwise a dot is shown.  The value of the byte is    *
' *  displayed in hex below the character.  Points where tracing was  *
' *  stopped are shown by a solid bar.                                *
' *                                                                   *
' *********************************************************************

        '
        '  Establish system constants and variables
        '
        DEFINT A-Z

        DIM INREG(7), OUTREG(7)         'Define register arrays

        RAX = 0                         'Establish values for 8086
        RBX = 1                         ' registers
        RCX = 2                         '  .
        RDX = 3                         '  .
```

Figure 18-7. Formatted dump routine for serial-trace buffer. *(more)*

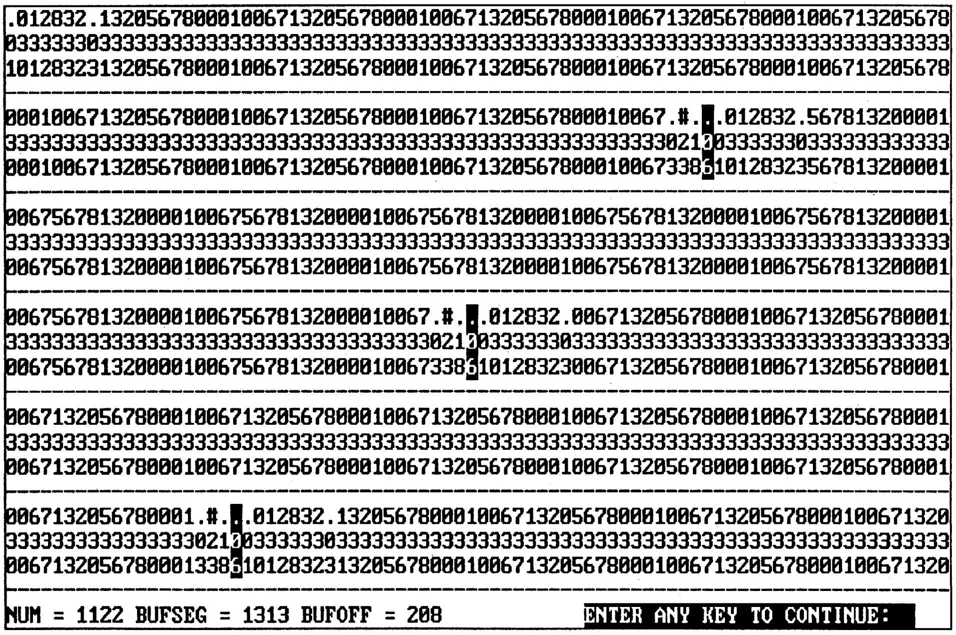
```
RBP = 4                         '  .
RSI = 5                         '  .
RDI = 6                         '  .
RFL = 7                         '  .

'
'  Interrogate COMMSCOP to obtain addresses and count of data in
'  trace buffer
'
INREG(RAX) = &H0300             'Request address data and count
CALL INT86(&H60, VARPTR(INREG(0)), VARPTR(OUTREG(0)))

NUM = OUTREG(RCX)               'Number of bytes in buffer
BUFSEG = OUTREG(RDX)            'Buffer segment address
BUFOFF = OUTREG(RBX)            'Offset of buffer start

IF NUM = 0 THEN END

'
'  Set screen up and display control data
'
CLS
KEY OFF
LOCATE 25, 1
PRINT "NUM ="; NUM;"BUFSEG = "; HEX$(BUFSEG); " BUFOFF = ";
PRINT HEX$(BUFOFF);
LOCATE 4, 1
PRINT STRING$(80,"-")
DEF SEG = BUFSEG

'
'  Set up display control variables
'
DLINE = 1
DCOL = 1
DSHOWN = 0

'
'  Fetch and display each character in buffer
'
FOR I= BUFOFF TO BUFOFF+NUM-2 STEP 2
    STAT = PEEK(I)
    DAT = PEEK(I + 1)

    IF (STAT AND 1) = 0 THEN
        COLOR 7, 0
    ELSE
        COLOR 0, 7
    END IF

    RLINE = (DLINE-1) * 4 + 1
```

Figure 18-7. Continued. *(more)*

```
        IF (STAT AND &H80) = 0 THEN
            LOCATE RLINE, DCOL
            C$ = CHR$(DAT)
            IF DAT < 32 THEN C$ = "."
            PRINT C$;
            H$ = RIGHT$("00" + HEX$(DAT), 2)
            LOCATE RLINE + 1, DCOL
            PRINT LEFT$(H$, 1);
            LOCATE RLINE + 2, DCOL
            PRINT RIGHT$(H$, 1);
        ELSE
            LOCATE RLINE, DCOL
            PRINT CHR$(178);
            LOCATE RLINE + 1, DCOL
            PRINT CHR$(178);
            LOCATE RLINE + 2, DCOL
            PRINT CHR$(178);
        END IF

        DCOL = DCOL + 1
        IF DCOL > 80 THEN
            COLOR 7, 0
            DCOL = 1
            DLINE = DLINE + 1
            SHOWN = SHOWN + 1
            IF SHOWN = 6 THEN
                LOCATE 25, 50
                COLOR 0, 7
                PRINT "ENTER ANY KEY TO CONTINUE:  ";
                WHILE LEN(INKEY$) = 0
                WEND
                COLOR 7, 0
                LOCATE 25, 50
                PRINT SPACE$(29);
                SHOWN = 0
            END IF
            IF DLINE > 6 THEN
                LOCATE 24, 1
                PRINT : PRINT : PRINT : PRINT
                LOCATE 24, 1
                PRINT STRING$(80, "-");
                DLINE = 6
            ELSE
                LOCATE DLINE * 4, 1
                PRINT STRING$(80, "-");
            END IF
        END IF

    NEXT I

    END
```

Figure 18-7. Continued.

COMMDUMP is a simple routine. Like most debugging aids, it lacks needless frills. When it is executed, COMMDUMP displays the data in the trace buffer on the screen in the format shown in Figure 18-8.



```
.012832.132056780001006713205678000100671320567800010067132056780001006713205678
03333330333333333333333333333333333333333333333333333333333333333333333333333333
10128323132056780001006713205678000100671320567800010067132056780001006713205678
--------------------------------------------------------------------------------
0001006713205678000100671320567800010067132056780001006.#...012832.567813200001
```

```
.012832.132056780001006713205678000100671320567800010067132056780001006713205678
03333330333333333333333333333333333333333333333333333333333333333333333333333333
10128323132056780001006713205678000100671320567800010067132056780001006713205678
--------------------------------------------------------------------------------
00010067132056780001006713205678000100671320567800010067.#...012832.567813200001
33333333333333333333333333333333333333333333333333333333021003333330333333333333
00010067132056780001006713205678000100671320567800010067338610128323567813200001
--------------------------------------------------------------------------------
00675678132000010067567813200001006756781320000100675678132000010067567813200001
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00675678132000010067567813200001006756781320000100675678132000010067567813200001
--------------------------------------------------------------------------------
006756781320000100675678132000010067.#...012832.00671320567800010067132056780001
33333333333333333333333333333333333302100333333033333333333333333333333333333333
00675678132000010067567813200001006733861012832300671320567800010067132056780001
--------------------------------------------------------------------------------
00671320567800010067132056780001006713205678000100671320567800010067132056780001
33333333333333333333333333333333333333333333333333333333333333333333333333333333
00671320567800010067132056780001006713205678000100671320567800010067132056780001
--------------------------------------------------------------------------------
0067132056780001.#...012832.1320567800010067132056780001006713205678000100671320
33333333333333330210033333303333333333333333333333333333333333333333333333333333
00671320567800013386101283231320567800010067132056780001006713205678000100671320
--------------------------------------------------------------------------------
NUM = 1122 BUFSEG = 1313 BUFOFF = 208          ENTER ANY KEY TO CONTINUE:
```

Figure 18-8. Formatted trace dump routine output.

Note that the data for each byte is presented in two forms. If the byte is greater than 1FH, the ASCII character represented by that number is shown; otherwise, a dot is shown. Directly below each character is the hexadecimal representation of the data. The display shows received data in reverse video and transmitted data in normal video. The mark placed in the buffer when collection is stopped and resumed is represented on the screen as a vertical bar one character wide. The display pauses when the screen is full and waits for a key to be pressed.

Data collected and displayed in this way can be invaluable to the programmer trying to debug a program involving a communications protocol. The example shown above is part of an ordered exchange of sales data for a system using blocked transmissions and ACK/NAK protocol. Like all debugging, finding bugs in such a system requires the collection of large amounts of data. With no data, the causes of problems can be almost impossible to find; with sufficiently large amounts of data, the solutions are obvious.

Several things could be done to the COMMSCOP program to increase its usefulness. For instance, there are six unused bits in the tag accompanying each data byte in the trace buffer. These could be used to record the status of the modem control bits, to place timer ticks in the buffer, or to coordinate the data with some outside event. (Such changes to COMMSCOP would require a more complicated COMMDUMP routine to display them.)

Software debugging monitors

Debugging monitors provide the next level of sophistication in the hierarchy of debugging methods. These monitors are coresident in memory with the application being debugged and provide a controlled testing environment—that is, they allow the programmer to control the execution of the program and to monitor the results. They even allow some problems to be fixed directly and the result reexecuted immediately, without the need to reassemble or recompile.

These monitors are analogous to the TSR serial monitor from the previous section. The debugging monitors, however, do not reside permanently in memory and are controlled interactively from the keyboard during the execution of the program under test. Although this level of control is more flexible than instrumentation, it is also more intrusive into program execution. While the debugging monitor sits and waits for input from the keyboard, the application program is also idle. For programs that must run in real time or must respond to external stimuli, long delays can be fatal. Careful planning and a thorough knowledge of the internal workings of the program are required to debug in such an environment.

Other problems with debugging monitors arise from the nature of the monitors themselves. They are programs, no different from the application program being debugged and are therefore limited to those things that can be done with software. For instance, they can break (stop execution to allow investigation of program status) when a specific instruction address is executed (because this can be done with software), but they cannot break when a data address is referenced (because this would require special hardware). Because these monitors reside in RAM, as do the application program and MS-DOS, they are susceptible to damage from a program running wild. Some trial and error is usually involved in locating the problem causing this kind of damage; breakpoints won't work here because the problem kills the monitor (and usually MS-DOS also).

Microsoft provides three debugging monitors, each with greater capabilities than its predecessor. In order of increasing sophistication, these three monitors are

Monitor	Description
DEBUG	A basic debugging monitor with the ability to load files, modify memory and registers, execute programs, set simple breakpoints, trace execution, modify disk files, and enter assembly-language statements into memory.
SYMDEB	A more advanced debugging monitor incorporating all the features of DEBUG plus more sophisticated data display, support for graphics programs, support for the Intel 80186/80286 microprocessors and the Intel 80287 math coprocessor, improved breakpoints, improved tracing, recognition of symbols from the program being debugged, and limited source-line display.
CodeView	The most sophisticated debugging monitor, incorporating the functionality of SYMDEB (with some differences in the details) plus windows, full source-line support, mouse support, and generally more sophistication on all functions.

Although all these debugging monitors will be discussed here, this section is not intended to be a tutorial on all the commands and options of the monitors—those are presented elsewhere in this volume and in the manuals accompanying the monitors. *See* PROGRAMMING UTILITIES: DEBUG; SYMDEB; CODEVIEW. Rather, this section uses case histories and sample programs to illustrate the techniques for solving various types of common debugging problems. The case histories have been chosen to show a wide range of problems, from simple to extremely complex.

DEBUG

Although DEBUG is the least sophisticated of the software debugging monitors, it is quite useful with moderately complex programs and is an effective tool for learning basic techniques.

Basic techniques

The first sample program is written in assembly language. It is a test program that performs serial input and output and was used to debug COMMSCOP, the serial-trace TSR presented earlier. The routine reads from the keyboard and writes to COM1 by means of Interrupt 14H. It also accepts incoming serial data and displays it on the screen. This process continues until Ctrl-C is pressed on the keyboard. A serial terminal is attached to COM1 to serve as a data source. Figure 18-9 shows the erroneous program.

```
        TITLE TESTCOMM - TEST COMMSCOP ROUTINE
; ************************************************************************
; *                                                                      *
; *  TESTCOMM                                                            *
; *     THIS ROUTINE PROVIDES DATA FOR THE COMMSCOP ROUTINE.  IT READS   *
; *     CHARACTERS FROM THE KEYBOARD AND WRITES THEM TO COM1 USING       *
; *     INT 14H.  DATA IS ALSO READ FROM INT 14H AND DISPLAYED ON THE    *
; *     SCREEN.  THE ROUTINE RETURNS TO MS-DOS WHEN Ctrl-C IS PRESSED    *
; *     ON THE KEYBOARD.                                                 *
; *                                                                      *
; ************************************************************************

SSEG    SEGMENT PARA STACK 'STACK'
        DW      128 DUP(?)
SSEG    ENDS

CSEG    SEGMENT
        ASSUME  CS:CSEG,SS:SSEG
BEGIN   PROC    FAR
        PUSH    DS                      ;SET UP FOR RET TO MS-DOS
        XOR     AX,AX                   ; .
        PUSH    AX                      ; .
```

Figure 18-9. Incorrect serial test routine. *(more)*

```
MAINLOOP:
        MOV     AH,6                    ;USE MS-DOS CALL TO CHECK FOR
        MOV     DL,0FFH                 ; KEYBOARD ACTIVITY
        INT     21                      ; IF NO CHARACTER, JUMP TO
        JZ      TESTCOMM                ; COMM ACTIVITY TEST

        CMP     AL,03                   ;WAS CHARACTER A Ctrl-C?
        JNE     SENDCOMM                ; NO, SEND IT TO SERIAL PORT
        RET                             ; YES, RETURN TO MS-DOS

SENDCOMM:
        MOV     AH,01                   ;USE INT 14H WRITE FUNCTION TO
        MOV     DX,0                    ; SEND DATA TO SERIAL PORT
        INT     14H                     ; .

TESTCOMM:
        MOV     AH,3                    ;GET SERIAL PORT STATUS
        MOV     DX,0                    ; .
        INT     14H                     ; .
        AND     AH,1                    ;ANY DATA WAITING?
        JZ      MAINLOOP                ; NO, GO BACK TO KEYBOARD TEST
        MOV     AH,2                    ;READ SERIAL DATA
        MOV     DX,0                    ; .
        INT     14H                     ; .
        MOV     AH,6                    ;WRITE SERIAL DATA TO SCREEN
        INT     21H                     ; .
        JMP     MAINLOOP                ;CONTINUE
BEGIN   ENDP
CSEG    ENDS
        END     BEGIN
```

Figure 18-9. Continued.

When executed, this program produces a constant stream of zeros from the serial port. Incoming serial data is not echoed on the screen, but the cursor moves as if it were. Further, the Ctrl-C keystroke is not recognized, so the only way to stop the program is to restart the system.

An examination of the listing should reveal the errors that cause these problems, but things do not always happen that way. For the purposes of this case study, assume that the listing was no help. Instrumentation is more difficult for assembly-language programs than for programs written in higher-order languages, so in this case it is advantageous to go directly to a debugging monitor. The monitor for this example is DEBUG.

The first step in using DEBUG is not to invoke the monitor; rather, it is to gather all pertinent listings, link maps, and program design documentation. In this case, the program is so short that a link map will not be needed; all the design documentation that exists is in the program comments.

Now begin DEBUG by typing

```
C>DEBUG TESTCOMM.EXE  <Enter>
```

The filename must be fully qualified; DEBUG makes no assumptions about the extension. Any type of file can be examined with DEBUG, but only files with an extension of .COM, .EXE, or .HEX are actually loaded and made ready for execution. Since TESTCOMM is a .EXE file, DEBUG loads it and prepares it for execution in a manner compatible with the MS-DOS loader. Type the Display or Modify Registers command, R.

```
-R  <Enter>
AX=0000  BX=0000  CX=0131  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0000   NV UP EI PL NZ NA PO NC
1ACD:0000 1E            PUSH  DS
```

Notice that the SS and CS registers have been loaded to their correct values and that SP points to the bottom of the stack. DS and ES point to an address 100H bytes (10H paragraphs) before the stack segment. (This is because the system sets these registers to point to the program segment prefix [PSP] when a .EXE program is loaded.) Normally, the program code would be responsible for loading the correct value of DS, but this example does not use the data segment, so the program doesn't bother. The register display also shows the instruction at the current value of CS:IP, 1ACD:0000H. The instruction pointer was set to this address because the END statement in the source program specified the procedure *BEGIN* as the entry point and that procedure begins at CS:IP. Note that the instruction displayed below the register information has not yet been executed. This condition is true for all register displays in DEBUG — IP always points to the *next* instruction to be executed, so the instruction at IP has not been executed.

From the symptoms observed during program execution, it is clear that the keyboard data is not reaching the serial port. The failure could be in the keyboard read routine or in the serial port write routine. This code is compact and fairly linear, so the easiest way to find out what is going on is to trace through the first few instructions of the program. Executing five instructions with the Trace Program Execution command, T, will do this.

```
-T5  <Enter>

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FE  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0001   NV UP EI PL NZ NA PO NC
1ACD:0001 33C0          XOR   AX,AX

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FE  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0003   NV UP EI PL ZR NA PE NC
1ACD:0003 50            PUSH  AX

AX=0000  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0004   NV UP EI PL ZR NA PE NC
1ACD:0004 B406          MOV   AH,06

AX=0600  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0006   NV UP EI PL ZR NA PE NC
1ACD:0006 B2FF          MOV   DL,FF

AX=0600  BX=0000  CX=0131  DX=00FF  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0008   NV UP EI PL ZR NA PE NC
1ACD:0008 CD15          INT   15
```

The Trace command shows the contents of the registers as each instruction is executed. The register contents are *after* the execution of the instruction listed above the registers and the instruction shown with the registers is the *next* instruction to be executed. The first register display in this example represents the state of affairs after the execution of the PUSH DS instruction, as indicated by SP. The first three instructions set up the stack so that the far return issued at the end of the program will pass control to the PSP for termination. The next two instructions set the registers for a Direct Console I/O MS-DOS call (AH = 060, DL = HFFH for input). After these registers are set up, the program should execute the MS-DOS call INT 21H. However, the next instruction to be executed is INT 15H. This is the reason the keyboard data is not being read. The code requests INT 21, not 21H. This mistake is a common one. The assembler's default radix is decimal, so it converted 21 into 15H. This error can be corrected in memory from within DEBUG and, because the instruction hasn't executed yet, the fix can be tested immediately. To make the correction, use the Assemble Machine Instructions command, A.

```
-A 8   <Enter>
1ACD:0008 int 21   <Enter>
1ACD:000A   <Enter>
```

The *A 8* code instructs DEBUG to begin assembling at CS:0008H. DEBUG prompts with the address and waits for an instruction to be entered. The letter *H* is not needed after the 21 this time because DEBUG assumes all numbers entered with the Assemble command are in hexadecimal form. In general, any valid 8086/8087/8088 assembly-language statement can be entered this way and translated into executable machine code. *See* PROGRAMMING UTILITIES: DEBUG: A. Within its restrictions, the Assemble command is a handy way of making changes. The Enter Data command, E, could also have been used to change the 15H to a 21H, but the Assemble command is safer, especially for complex instructions. After the new instruction has been entered, press Enter again to stop the assembly process.

There is a danger associated with making changes in memory during debugging: The memory copy of the program is temporary; the changes exist only in memory and when DEBUG exits, they are lost. Changes made to .EXE and .HEX files cannot be written back to disk. To avoid forgetting the changes, write them down. When DEBUG exits, edit the source file *immediately.* Changes made to other files can be written back to disk with DEBUG's Write File or Sectors command, W.

To be sure that the change was made correctly, use the Disassemble (Unassemble) Program command, U, to show the instructions starting at CS:0004H.

```
-U 4   <Enter>
1ACD:0004 B406          MOV    AH,06
1ACD:0006 B2FF          MOV    DL,FF
1ACD:0008 CD21          INT    21
1ACD:000A 740C          JZ     0018
1ACD:000C 3C03          CMP    AL,03
1ACD:000E 7501          JNZ    0011
1ACD:0010 CB            RETF
```

```
1ACD:0011 B401          MOV    AH,01
1ACD:0013 BA0000        MOV    DX,0000
1ACD:0016 CD14          INT    14
1ACD:0018 B403          MOV    AH,03
1ACD:001A BA0000        MOV    DX0000
1ACD:001D CD14          INT    14
1ACD:001F 80E401        AND    AH,01
1ACD:0022 74E0          JZ     0004
```

The change has been correctly made. Now, to test the change, start the program to see if characters make it out the serial port. The problem of data from the serial port not making it to the screen remains, however, so instead of simply starting the program, set a breakpoint at the location in the program that handles incoming serial data (CS:0024H). This technique allows the output section of the code to be tested separately. The breakpoint is set using the Go command, G.

```
-G 24  <Enter>

AX=0130  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=0024   NV UP EI PL NZ NA PO NC
1ACD:0024 B402          MOV    AH,02
-U  <Enter>
1ACD:0024 B402          MOV    AH,02
1ACD:0026 BA0000        MOV    DX,0000
1ACD:0029 CD14          INT    14
1ACD:002B B406          MOV    AH,06
1ACD:002D CD21          INT    21
1ACD:002F EBD3          JMP    0004
1ACD:0031 0000          ADD    [BX+SI],AL
1ACD:0033 0000          ADD    [BX+SI],AL
1ACD:0035 0000          ADD    [BX+SI],AL
1ACD:0037 0000          ADD    [BX+SI],AL
1ACD:0039 0000          ADD    [BX+SI],AL
1ACD:003B 0000          ADD    [BX+SI],AL
1ACD:003D 0000          ADD    [BX+SI],AL
1ACD:003F 0000          ADD    [BX+SI],AL
1ACD:0041 0000          ADD    [BX+SI],AL
1ACD:0043 0000          ADD    [BX+SI],AL
```

As stated earlier, the serial port is attached to a serial terminal. After execution of the program is started with the Go command, all keys typed on the keyboard are displayed correctly on the terminal, thus confirming the fix made to the INT 21H instruction. To test serial input, a key must be pressed on the terminal, causing the breakpoint at CS:0024H to be executed.

The fact that location CS:0024H was reached indicates that Interrupt 14H is detecting the presence of an input character. To test if the character is now making it to the screen, a breakpoint is needed after the write to the screen. The Disassemble command shows the instructions starting at the current IP value. The program ends at CS:002FH; the instructions shown after that are whatever happened to be in memory when the program was loaded. A good place to set the next breakpoint is CS:002FH, just after the Interrupt 21H call.

```
-G 2f  <Enter>

AX=0600  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=002F   NV UP EI PL NZ NA PO NC
1ACD:002F EBD3          JMP    0004
```

DEBUG shows that the breakpoint was reached and the character did not print (it should have been on the line after *-G 2f*), so something must be wrong with the Interrupt 21H call. A breakpoint just before the MS-DOS call at CS:002DH should reveal the cause of the problem.

```
-G 2d  <Enter>

AX=0662  BX=0000  CX=0131  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1AAD  ES=1AAD  SS=1ABD  CS=1ACD  IP=002D   NV UP EI PL NZ NA PO NC
1ACD:002D CD21          INT    21
```

The key that was entered on the serial terminal (b) is in AL, where it was returned by Interrupt 14H. Unfortunately, it is not in DL, where it is expected by the Direct Console I/O function (06H) of the MS-DOS command. The MS-DOS function was simply printing a null (00H) and then moving the cursor. An instruction (MOV DL,AL) is missing.

Fixing this problem requires the insertion of a line of code, which is usually difficult to do inside DEBUG. The Move (Copy) Data command, M, can be used to move the code located below the point where the insertion is to be made down 2 bytes, but this will probably throw any subsequent addressing off. It is usually easier to exit DEBUG, edit the source file, and then reassemble. In this case, however, because the instruction to be added is near the last instruction, a patch can easily be made by entering only three instructions: the new one and the two it destroys.

```
-A 2d  <Enter>
1ACD:002D mov dl,al  <Enter>
1ACD:002F int 21  <Enter>
1ACD:0031 jmp 4  <Enter>
1ACD:0033  <Enter>
-U 2b  <Enter>
1ACD:002B B406          MOV    AH,06
1ACD:002D 88C2          MOV    DL,AL
1ACD:002F CD21          INT    21
1ACD:0031 EBD1          JMP    0004
1ACD:0033 0000          ADD    [BX+SI],AL
1ACD:0035 0000          ADD    [BX+SI],AL
1ACD:0037 0000          ADD    [BX+SI],AL
1ACD:0039 0000          ADD    [BX+SI],AL
1ACD:003B 0000          ADD    [BX+SI],AL
1ACD:003D 0000          ADD    [BX+SI],AL
1ACD:003F 0000          ADD    [BX+SI],AL
1ACD:0041 0000          ADD    [BX+SI],AL
1ACD:0043 0000          ADD    [BX+SI],AL
1ACD:0045 0000          ADD    [BX+SI],AL
1ACD:0047 0000          ADD    [BX+SI],AL
1ACD:0049 0000          ADD    [BX+SI],AL
```

The new line of code has been inserted and verified with the Disassemble command. The fix is ready to test. The Trace command could be used to single-step through the program to verify execution. A word of warning is in order, however: The DEBUG Trace command should never be used to trace an Interrupt 21H call. Once the trace enters the MS-DOS call, it will wander around for a while and then lock the machine, requiring a restart. Avoid this problem either by setting a breakpoint just beyond the Interrupt 21H call or by using the Proceed Through Loop or Subroutine command, P. The Proceed command operates in a similar manner to the Trace command but does not trace loops, calls, and interrupts.

Because the fix is fairly certain, use the Go command in its simple form with no breakpoints. The program will execute without further intervention from DEBUG.

```
-G  <Enter>
lasdfgh
Program terminated normally
-Q  <Enter>
```

The *lasdfgh* text entered on the serial terminal is displayed correctly. When a Ctrl-C is entered from the keyboard, the program terminates properly and DEBUG displays the message *Program terminated normally.* Now exit DEBUG with the Quit command, Q.

The source code of TESTCOMM should be edited immediately so that it reflects the two changes made temporarily under DEBUG. Figure 18-10 shows the corrected listing.

```
     TITLE TESTCOMM - TEST COMMSCOP ROUTINE
; ***********************************************************************
; *                                                                     *
; *  TESTCOMM                                                           *
; *     THIS ROUTINE PROVIDES DATA FOR THE COMMSCOP ROUTINE.  IT READS  *
; *  CHARACTERS FROM THE KEYBOARD AND WRITES THEM TO COM1 USING         *
; *  INT 14H.  DATA IS ALSO READ FROM INT 14H AND DISPLAYED ON THE      *
; *  SCREEN.  THE ROUTINE RETURNS TO MS-DOS WHEN Ctrl-C IS PRESSED      *
; *  ON THE KEYBOARD.                                                   *
; *                                                                     *
; ***********************************************************************

SSEG    SEGMENT PARA STACK 'STACK'
        DW      128 DUP(?)
SSEG    ENDS

CSEG    SEGMENT
        ASSUME  CS:CSEG,SS:SSEG
BEGIN   PROC    FAR
        PUSH    DS                      ;SET UP FOR RET TO MS-DOS
        XOR     AX,AX                   ; .
        PUSH    AX                      ; .
```

Figure 18-10. Correct serial test routine. *(more)*

```
MAINLOOP:
        MOV     AH,6                    ;USE DOS CALL TO CHECK FOR
        MOV     DL,0FFH                 ; KEYBOARD ACTIVITY
        INT     21H                     ; IF NO CHARACTER, JUMP TO
        JZ      TESTCOMM                ; COMM ACTIVITY TEST

        CMP     AL,03                   ;WAS CHARACTER A Ctrl-C?
        JNE     SENDCOMM                ; NO, SEND IT TO SERIAL PORT
        RET                             ; YES, RETURN TO MS-DOS

SENDCOMM:
        MOV     AH,01                   ;USE INT 14H WRITE FUNCTION TO
        MOV     DX,0                    ; SEND DATA TO SERIAL PORT
        INT     14H                     ; .

TESTCOMM:
        MOV     AH,3                    ;GET SERIAL PORT STATUS
        MOV     DX,0                    ; .
        INT     14H                     ; .
        AND     AH,1                    ;ANY DATA WAITING?
        JZ      MAINLOOP                ; NO, GO BACK TO KEYBOARD TEST
        MOV     AH,2                    ;READ SERIAL DATA
        MOV     DX,0                    ; .
        INT     14H                     ; .
        MOV     AH,6                    ;WRITE SERIAL DATA TO SCREEN
        MOV     DL,AL                   ; .
        INT     21H                     ; .
        JMP     MAINLOOP                ;CONTINUE

BEGIN   ENDP
CSEG    ENDS
        END     BEGIN
```

Figure 18-10. Continued.

DEBUG has a rich set of commands and features. The preceding case study shows the more common ones in their most straightforward aspect. Some of the other commands and some useful techniques are described below. *See* PROGRAMMING UTILITIES: DEBUG.

Establishing initial conditions

When a program is loaded for testing, four areas may require initialization:

- Registers
- Data areas
- Default file-control blocks (FCBs)
- Command tail

These areas may also require changes during testing, especially when the programmer is working around bugs or establishing different test conditions.

Registers. Registers are ordinarily set when the program is loaded. The values in them depend on whether a .EXE, .COM, or .HEX file was loaded. Generally, the segment registers, the IP register, and the SP register are set to appropriate values; with the exception of AX, BX, and CX, the rest of the registers are set to zero. BX and CX contain the length of the loaded file. By MS-DOS convention, when a program is loaded, the contents of AL and AH indicate the validity of the drive specifiers in the first and second DEBUG command-line parameters, respectively. Each register contains zero if the corresponding drive was valid, 01H if the drive was valid and wildcards were used, or 0FFH if the drive was invalid.

To change the value of any register, use an alternate form of the Register command. Enter R followed by the two-letter register name. Only 16-bit registers can be changed, so use the X form of the general-purpose registers:

```
-R AX  <Enter>
```

DEBUG will respond with the current contents of the register and prompt for a new value. Either enter a new hexadecimal value or press Enter to keep the current value:

```
AX 0000
:FFFF  <Enter>
```

In this example, the new value of AX is FFFFH.

When changing registers, exercise caution modifying the segment registers. These registers control the execution of the program and should be changed only after careful and thoughtful consideration.

The Register command can also be used to modify the CPU flags.

Data areas. Initializing or changing data areas is easy, and several methods are provided. The Fill Memory command, F, can be used to initialize areas of RAM. For instance,

```
-F 0 L400 0  <Enter>
```

fills DS:0000H through DS:03FFH with zero. (The absence of a segment override causes the Fill command to use its default segment, DS.) Entering

```
-F CS:100 200 1B "[Hello" 0D  <Enter>
```

fills CS:0100H through CS:0200H with many repetitions of the string 1B 5B 48 65 6C 6C 6F 0D. (Note that an address range was specified, not a length.)

When the wholesale changing of memory is not appropriate, the Enter command can be used to edit a small number of locations. The Enter command has two forms: One enters a list of bytes into the specified memory location; the other prompts with the contents of each location and waits for input. Either form can be used as appropriate.

Default file-control blocks and the command tail. The setting of the default FCBs and of the command tail are related functions. When DEBUG is entered, the first parameter following the command DEBUG is the name of the file to be loaded into memory for debugging. If the next two parameters are filenames, FCBs for these files are formatted at

DS:005CH and DS:006CH in the PSP. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management. If either parameter contains a pathname, the corresponding FCB will contain only a valid drive number; the filename field will not be valid. All filenames and switches following the name of the file to be debugged are considered the command tail and are saved in memory starting at DS:0081H. The length of the command tail is in DS:0080H. For example, entering

```
C>DEBUG COMMDUMP.EXE FILE1.DAT FILE2.DAT  <Enter>
```

results in the first FCB (5CH), the second FCB (6CH), and the command tail (81H) being loaded as follows:

```
-D 50  <Enter>
42C9:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 46 49 4C   .!...........FIL
42C9:0060  45 31 20 20 20 44 41 54-00 00 00 00 00 46 49 4C   E1   DAT.....FIL
42C9:0070  45 32 20 20 20 44 41 54-00 00 00 00 00 00 00 00   E2   DAT........
42C9:0080  15 20 66 69 6C 65 31 2E-64 61 74 20 66 69 6C 65   . file1.dat file
42C9:0090  32 2E 64 61 74 20 0D 74-20 66 69 6C 65 32 2E 64   2.dat .t file2.d
42C9:00A0  61 74 20 0D 00 00 00 00-00 00 00 00 00 00 00 00   at .............
42C9:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
42C9:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

In this example, location DS:005CH contains an unopened FCB for file FILE1.DAT on the current drive. Location DS:006CH contains an unopened FCB for FILE2.DAT on the current drive. (The second FCB cannot be used where it is and must be moved to another location before the first FCB is opened.) Location DS:0080H contains the length of the command tail, 15H (21) bytes. The next 21 bytes are the command tail prepared by DEBUG; they correspond exactly to what the command tail would be if the program had been loaded by COMMAND.COM instead of by DEBUG.

The default FCBs and the command tail can also be set after the program has been loaded, by using the Name File or Command-Tail Parameters command, N. DEBUG treats the string of characters that follow the Name command as the command tail: If the first two parameters are filenames, they become the first and second FCBs, respectively. The Name command also places the string at DS:0081H, with the length of the string at DS:0080H. Entering the DEBUG command

```
-N FILE1.DAT FILE2.DAT  <Enter>
```

produces the same results as specifying the filenames in the command line. When employed in this manner, the Name command is useful for initializing command-tail data that was not in the command line or for changing the command-tail data to test different aspects of a program. (If files are named in this manner, they are not validated until the Load File or Sectors command, L, is used.) Note that the data following the Name command need not be filenames; it can be any parameters, data, or switches that the application program expects to see.

More on breakpoints

The case study at the beginning of this section used breakpoints in their simplest form: Only a single breakpoint was specified at a time and the execution address was considered to be the current IP. The Go command is also capable of setting multiple breakpoints and of beginning execution at any address in memory. The more general form of the Go command is

G[=*address*] [*address* [*address*...]]

If Go is used with no operands, execution begins at the current value of CS:IP and no breakpoints are set. If the *=address* operand is used, DEBUG sets IP to the address specified and execution then begins at the new CS:IP. The other optional addresses are breakpoints. When execution reaches one of these breakpoints, DEBUG stops and displays the system's registers. As many as 10 breakpoints can be set on one Go command, and they can be in any order.

The breakpoint addresses must be on instruction boundaries because DEBUG replaces the instruction at each breakpoint address with an INT 03H instruction (0CCH). DEBUG saves the replaced instructions internally. When any breakpoint is reached, DEBUG stops execution and restores the instructions at *all* the breakpoints; if no breakpoint is reached, the instructions are not restored and the Load command must be used to reload the original program.

The multiple-breakpoint feature of the Go command allows the tracing of program execution when branches exist in the code. When a program contains, for instance, a conditional jump on the zero flag, a breakpoint can be placed in each of the two possible branches. When the branch is reached, one of the two breakpoints will be encountered shortly thereafter. When DEBUG displays the breakpoint, the programmer knows which branch was taken. Moving through a program with breakpoints at key locations is faster than using the Trace command to execute each and every instruction.

Multiple breakpoints can also be used to home in on a bad piece of code. This technique is particularly useful in those nasty situations when there are no symptoms except that the system locks up and must be restarted. When debugging a problem such as this, set breakpoints at each of the major sections of the program and then note those breakpoints that are executed successfully, continuing until the system locks up. The problem lies somewhere between the last successful breakpoint and the next breakpoint set. Now repeat the processes, setting breakpoints between the last breakpoint and the one that was never reached. By progressively narrowing the gap between breakpoints, the exact offending instruction can be isolated.

Some general comments about the Go command and breakpoints:

- After a program has reached completion and returned to MS-DOS, it must be reloaded with the Load command before it can be executed again. (DEBUG intercepts this return and displays *Program terminated normally.*)
- Because DEBUG replaces program instructions with an INT 03H instruction to form breakpoints, the break address must be on an instruction boundary. If it is not, the INT 03H will be stuck in the middle of an instruction, causing strange and sometimes entertaining results.

- Breakpoints cannot be set in data, because data is not executed.
- The target program's SS:SP registers must point to a valid stack that has at least 6 bytes of stack space available. When the Go command is executed, it pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.
- Finally, and obviously, breakpoints cannot be set in read-only memory (the ROM BIOS, for instance).

Using the Write commands

After a program has been debugged, fixed, and tested with DEBUG, the temptation exists to write the patched program directly back to the disk as a .COM file. This action is sometimes legitimate, but only rarely. The technique will be explained in a moment, but first a sermon:

DON'T DO IT.

One of the greatest sadnesses in a programmer's life comes when, after a program has been running wonderfully, enhancements are made to the source code and the recompiled program suddenly has bugs in it that haven't been seen for months. Always make any debugging patches permanent in the source file immediately.

Unless, of course, the source code is not available. This is the only time saving a patched program is permissible. For example, sometimes commercial programs require patching because the program does not quite fit the hardware it must run on or because bugs have been found in the program. The source of these patches is sometimes word-of-mouth, sometimes a bulletin-board service, and sometimes the program's manufacturer.

Even when legitimate reasons exist to save patched code, precautions should be taken. Be very careful, meticulous, and alert as the patches are applied. Understand each step before undertaking it. Most important of all, always have a backup of the original unpatched program safely on a floppy disk.

Use the Write command to write the program image to disk. A starting address can optionally be specified; otherwise the write starts at CS:0100H. The name of the file will be either the name specified in the last Name command or the name of the program from the DEBUG command line if the Name command has not been used. The number of bytes to be written is in BX and CX, with the most significant half in BX. These registers will have been loaded correctly when the program was loaded, but they should be checked if the program has executed since it was loaded.

The .EXE and .HEX file types cannot be written to disk with the Write command. The command performs no formatting and only writes the binary image of memory to the disk file. Thus, all programs written with Write must be .COM files. The image of a .EXE or .HEX file can still be written as a .COM file provided no segment fixups are required and provided the other rules for a .COM file are followed. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. (A segment fixup is a segment address that must be provided by the loader when the program is originally loaded. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules.) If a .EXE file containing a segment fixup is written as a .COM file, the new file will execute correctly only when loaded at exactly the same address as the original file, and this is difficult to ensure for programs running under MS-DOS.

If it is necessary to patch a .EXE or .HEX file and the exact addresses relative to the start of the file are known, use the following procedure:

1. Rename (or better yet, copy) the file to an extension other than .EXE or .HEX.
2. Load the program image into memory by placing the new name on DEBUG's command line. Note that the loaded file is an image of the disk file and is not executable.
3. Modify the program image in memory, but *never* try to execute the program. Results would be unpredictable and the program image could be damaged.
4. Write the modified image back to disk using a simple *w*. No other action is needed, because the original load will have set the filename and the correct length in BX and CX.
5. Rename the file to a name with the correct .EXE or .HEX extension. The new name need not be the same as the original, but it should have the same extension.

The same technique can be used to load, modify, and save data files. Simply make sure that the file does not have an extension of .COM, .EXE, or .HEX. The data file will be loaded at address CS:0100H. (DEBUG treats the file much the same as a .COM file.) After patching the data (the Enter command works best), use the Write command to write it back to the disk.

SYMDEB

SYMDEB is an extension of DEBUG; virtually all the DEBUG commands and techniques still work as expected. The major new feature, and the source of the name SYMDEB, is symbolic debugging: SYMDEB can use all public labels in a program for reference, instead of using hexadecimal offset addresses. In addition, SYMDEB allows the use of line numbers for reference in compatible high-order languages; source-line display within SYMDEB is also possible for these languages. Currently, the languages supporting these options are Microsoft FORTRAN versions 3.0 and later, Microsoft Pascal versions 3.0 and later, and Microsoft C versions 2.0 and later. Versions 4.0 and earlier of the Microsoft Macro Assembler (MASM) do not generate the data needed for line-number display and source-line debugging.

In addition to symbolic debugging, SYMDEB has added several other new features and has expanded existing DEBUG features:

- Breakpoints have been made more sophisticated with the addition of "sticky" breakpoints. Unlike the breakpoints set with the Go command, sticky breakpoints remain attached to the program throughout a SYMDEB session until they are explicitly removed. Specific commands are supplied for listing, removing, enabling, and disabling sticky breakpoints.
- DEBUG's Display Memory command, D, has been extended so that data can be displayed in different formats.

- Full redirection is supported.
- A stack trace feature has been added.
- Terminate-and-stay-resident programs are supported.
- A shell escape command has been added to allow the execution of MS-DOS commands and programs without leaving SYMDEB and the debugging session.

These additions allow more sophisticated debugging techniques to be used and, in some cases, also simplify locating problems. To see the advantages of using symbols and sticky breakpoints in debugging, consider a type of program that is one of the most difficult to debug—the TSR.

Debugging TSRs with SYMDEB

Terminate-and-stay-resident routines can be difficult to debug. They exist in two worlds and can have bugs associated with each. At the outset, they are usually simple programs that perform some initialization task and then exit. At this point, they are transformed into another type of beast entirely—resident routines that are more a part of the operating system than of any application program. Each form of the program must be debugged separately, using different techniques.

The TSR routine used for this case study is the same one created previously to serve as external instrumentation to trace serial communications. The program was called COMMSCOP, but to avoid confusion of that working program with the broken one presented here, the name has been changed to BADSCOP. BADSCOP was assembled and linked in the usual manner and then converted to a .COM file using EXE2BIN. When it was installed, it returned normally, but at the first attempt to issue an Interrupt 14H, the system locked up completely. Warm booting was not sufficient to restore it, and a power-on cold boot was required to get the system working again.

Figure 18-11 is a listing of BADSCOP. The only difference from COMMSCOP, aside from the errors, is the addition of two PUBLIC statements to make all the procedure names and the important data names available to SYMDEB.

```
        TITLE   BADSCOP - BAD VERSION OF COMMUNICATIONS TRACE UTILITY
; ***********************************************************************
; *                                                                     *
; *  BADSCOP -                                                          *
; *     THIS PROGRAM MONITORS THE ACTIVITY ON A SPECIFIED COMM PORT     *
; *     AND PLACES A COPY OF ALL COMM ACTIVITY IN A RAM BUFFER.  EACH   *
; *     ENTRY IN THE BUFFER IS TAGGED TO INDICATE WHETHER THE BYTE      *
; *     WAS SENT BY OR RECEIVED BY THE SYSTEM.                          *
; *                                                                     *
; *     BADSCOP IS INSTALLED BY ENTERING                                *
; *                                                                     *
; *                     BADSCOP                                         *
; *                                                                     *
```

Figure 18-11. An incorrect version of the serial trace utility. *(more)*

```
; *  THIS WILL INSTALL BADSCOP AND SET UP A 64K BUFFER TO BE USED        *
; *  FOR DATA LOGGING.  REMEMBER THAT 2 BYTES ARE REQUIRED FOR           *
; *  EACH COMM BYTE, SO THE BUFFER IS ONLY 32K EVENTS LONG, OR ABOUT     *
; *  30 SECONDS OF CONTINUOUS 9600 BAUD DATA.  IN THE REAL WORLD,        *
; *  ASYNC DATA IS RARELY CONTINUOUS, SO THE BUFFER WILL PROBABLY        *
; *  HOLD MORE THAN 30 SECONDS WORTH OF DATA.                            *
; *                                                                      *
; *  WHEN INSTALLED, BADSCOP INTERCEPTS ALL INT 14H CALLS.  IF THE       *
; *  PROGRAM HAS BEEN ACTIVATED AND THE INT IS EITHER SEND OR RE-        *
; *  CEIVE DATA, A COPY OF THE DATA BYTE, PROPERLY TAGGED, IS PLACED     *
; *  IN THE BUFFER.  IN ANY CASE, DATA IS PASSED ON TO THE REAL          *
; *  INT 14H HANDLER.                                                    *
; *                                                                      *
; *  BADSCOP IS INVOKED BY ISSUING AN INT 60H CALL.  THE INT HAS         *
; *  THE FOLLOWING CALLING SEQUENCE:                                     *
; *                                                                      *
; *       AH - COMMAND                                                   *
; *            0 - STOP TRACING, PLACE STOP MARK IN BUFFER               *
; *            1 - FLUSH BUFFER AND START TRACE                          *
; *            2 - RESUME TRACE                                          *
; *            3 - RETURN COMM BUFFER ADDRESSES                          *
; *       DX - COMM PORT (ONLY USED WITH AH = 1 or 2)                    *
; *            0 - COM1                                                  *
; *            1 - COM2                                                  *
; *                                                                      *
; *  THE FOLLOWING DATA IS RETURNED IN RESPONSE TO AH = 3:               *
; *                                                                      *
; *       CX - BUFFER COUNT IN BYTES                                     *
; *       DX - SEGMENT ADDRESS OF THE START OF THE BUFFER                *
; *       BX - OFFSET ADDRESS OF THE START OF THE BUFFER                 *
; *                                                                      *
; *  THE COMM BUFFER IS FILLED WITH 2-BYTE DATA ENTRIES OF THE           *
; *  FOLLOWING FORM:                                                     *
; *                                                                      *
; *       BYTE 0 - CONTROL                                               *
; *            BIT 0 - ON FOR RECEIVED DATA, OFF FOR TRANS.              *
; *            BIT 7 - STOP MARK - INDICATES COLLECTION WAS              *
; *                  INTERRUPTED AND RESUMED.                            *
; *       BYTE 1 - 8-BIT DATA                                            *
; *                                                                      *
; ************************************************************************

        PUBLIC  INITIALIZE,CONTROL,VECTOR_INIT,COMMSCOPE
        PUBLIC  OLD_COMM_INT,COUNT,STATUS,PORT,BUFPNTR

CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
        ORG     100H                    ;TO MAKE A COM FILE
```

Figure 18-11. Continued.

(more)

```
INITIALIZE:
        JMP     VECTOR_INIT             ;JUMP TO THE INITIALIZATION
                                        ; ROUTINE WHICH, TO SAVE SPACE,
                                        ; IS IN THE COMM BUFFER


;
;  SYSTEM VARIABLES
;
OLD_COMM_INT    DD      ?               ;ADDRESS OF REAL COMM INT
COUNT           DW      0               ;BUFFER COUNT
COMMSCOPE_INT   EQU     60H             ;COMMSCOPE CONTROL INT
STATUS          DB      0               ;PROCESSING STATUS
                                        ; 0 - OFF
                                        ; 1 - ON
PORT            DB      0               ;COMM PORT BEING TRACED
BUFPNTR         DW      VECTOR_INIT     ;NEXT BUFFER LOCATION

        SUBTTL  DATA INTERRUPT HANDLER
PAGE
; ***********************************************************************
; *                                                                     *
; *   COMMSCOPE                                                         *
; *   THIS PROCEDURE INTERCEPTS ALL INT 14H CALLS AND LOGS THE DATA     *
; *   IF APPROPRIATE.                                                   *
; *                                                                     *
; ***********************************************************************
COMMSCOPE       PROC    NEAR

        TEST    CS:STATUS,1             ;ARE WE ON?
        JZ      OLD_JUMP                ; NO, SIMPLY JUMP TO OLD HANDLER

        CMP     AH,00H                  ;SKIP SETUP CALLS
        JE      OLD_JUMP                ; .

        CMP     AH,03H                  ;SKIP STATUS REQUESTS
        JAE     OLD_JUMP                ; .

        CMP     AH,02H                  ;IS THIS A READ REQUEST?
        JE      GET_READ                ; YES, GO PROCESS

;
;  DATA WRITE REQUEST - SAVE IF APPROPRIATE
;
        CMP     DL,CS:PORT              ;IS WRITE FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .
        MOV     BX,BUFPNTR              ;GET ADDRESS OF NEXT BUFFER LOCATION
```

Figure 18-11. Continued. *(more)*

```
        MOV     [BX],BYTE PTR 0         ;MARK AS TRANSMITTED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     WRITE_DONE              ;ZERO INDICATES BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF - BUFFER FULL
WRITE_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        JMP     OLD_JUMP                ;PASS REQUEST ON TO BIOS ROUTINE
;
;  PROCESS A READ DATA REQUEST AND WRITE TO BUFFER IF APPROPRIATE
;
GET_READ:
        CMP     DL,CS:PORT              ;IS READ FOR PORT BEING TRACED?
        JNE     OLD_JUMP                ; NO, JUST PASS IT THROUGH

        PUSH    DS                      ;SAVE CALLER'S REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET UP DS FOR OUR PROGRAM
        POP     DS                      ; .

        PUSHF                           ;FAKE INT 14H CALL
        CLI                             ; .
        CALL    OLD_COMM_INT            ;PASS REQUEST ON TO BIOS
        TEST    AH,80H                  ;VALID READ?
        JNZ     READ_DONE               ; NO, SKIP BUFFER UPDATE

        MOV     BX,BUFPNTR              ;GET ADDRESS OF NEXT BUFFER LOCATION
        MOV     [BX],BYTE PTR 1         ;MARK AS RECEIVED BYTE
        MOV     [BX+1],AL               ;SAVE DATA IN BUFFER
        INC     COUNT                   ;INCREMENT BUFFER BYTE COUNT
        INC     COUNT                   ; .
        INC     BX                      ;POINT TO NEXT LOCATION
        INC     BX                      ; .
        MOV     BUFPNTR,BX              ;SAVE NEW POINTER
        JNZ     READ_DONE               ;ZERO INDICATES BUFFER HAS WRAPPED

        MOV     STATUS,0                ;TURN COLLECTION OFF - BUFFER FULL
READ_DONE:
        POP     BX                      ;RESTORE CALLER'S REGISTERS
        POP     DS                      ; .
        IRET

;
;  JUMP TO COMM BIOS ROUTINE
;
OLD_JUMP:
        JMP     OLD_COMM_INT

COMMSCOPE ENDP
```

Figure 18-11. Continued.

(more)

```
        SUBTTL  CONTROL INTERRUPT HANDLER
PAGE
; ***********************************************************************
; *                                                                     *
; *  CONTROL                                                            *
; *     THIS ROUTINE PROCESSES CONTROL REQUESTS.                        *
; *                                                                     *
; ***********************************************************************

CONTROL PROC    NEAR
        CMP     AH,00H                  ;STOP REQUEST?
        JNE     CNTL_START              ; NO, CHECK START
        PUSH    DS                      ;SAVE REGISTERS
        PUSH    BX                      ; .
        PUSH    CS                      ;SET DS FOR OUR ROUTINE
        POP     DS
        MOV     STATUS,0                ;TURN PROCESSING OFF
        MOV     BX,BUFPNTR              ;PLACE STOP MARK IN BUFFER
        MOV     [BX],BYTE PTR 80H       ; .
        MOV     [BX+1],BYTE PTR 0FFH    ; .
        INC     COUNT                   ;INCREMENT COUNT
        INC     COUNT                   ; .
        POP     BX                      ;RESTORE REGISTERS
        POP     DS                      ; .
        JMP     CONTROL_DONE

CNTL_START:
        CMP     AH,01H                  ;START REQUEST?
        JNE     CNTL_RESUME             ; NO, CHECK RESUME
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:BUFPNTR,OFFSET VECTOR_INIT   ;RESET BUFFER TO START
        MOV     CS:COUNT,0              ;ZERO COUNT
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_RESUME:
        CMP     AH,02H                  ;RESUME REQUEST?
        JNE     CNTL_STATUS             ; NO, CHECK STATUS
        CMP     CS:BUFPNTR,0            ;END OF BUFFER CONDITION?
        JE      CONTROL_DONE            ; YES, DO NOTHING
        MOV     CS:PORT,DL              ;SAVE PORT TO TRACE
        MOV     CS:STATUS,1             ;START LOGGING
        JMP     CONTROL_DONE

CNTL_STATUS:
        CMP     AH,03H                  ;RETURN STATUS REQUEST?
        JNE     CONTROL_DONE            ; NO, ERROR - DO NOTHING
        MOV     CX,CS:COUNT             ;RETURN COUNT
        PUSH    CS                      ;RETURN SEGMENT ADDR OF BUFFER
        POP     DX                      ; .
        MOV     BX,OFFSET VECTOR_INIT   ;RETURN OFFSET ADDR OF BUFFER
```

Figure 18-11. Continued. *(more)*

```
CONTROL_DONE:
        IRET

CONTROL ENDP


        SUBTTL  INITIALIZE INTERRUPT VECTORS
PAGE
; *********************************************************************
; *                                                                   *
; *  VECTOR_INIT                                                      *
; *  THIS PROCEDURE INITIALIZES THE INTERRUPT VECTORS AND THEN        *
; *  EXITS VIA THE MS-DOS TERMINATE-AND-STAY-RESIDENT FUNCTION.       *
; *  A BUFFER OF 64K IS RETAINED.  THE FIRST AVAILABLE BYTE           *
; *  IN THE BUFFER IS THE OFFSET OF VECTOR_INIT.                      *
; *                                                                   *
; *********************************************************************

        EVEN                            ;ASSURE BUFFER ON EVEN BOUNDARY
VECTOR_INIT     PROC    NEAR
;
;  GET ADDRESS OF COMM VECTOR (INT 14H)
;
        MOV     AH,35H
        MOV     AL,14H
        INT     21H
;
;  SAVE OLD COMM INT ADDRESS
;
        MOV     WORD PTR OLD_COMM_INT,BX
        MOV     AX,ES
        MOV     WORD PTR OLD_COMM_INT[2],AX
;
;  SET UP COMM INT TO POINT TO OUR ROUTINE
;
        MOV     DX,OFFSET COMMSCOPE
        MOV     AH,25H
        MOV     AL,14H
        INT     21H
;
;  INSTALL CONTROL ROUTINE INT
;
        MOV     DX,OFFSET CONTROL
        MOV     AH,25H
        MOV     AL,COMMSCOPE_INT
        INT     21H
;
;  SET LENGTH TO 64K, EXIT AND STAY RESIDENT
;
        MOV     AX,3100H                ;TERM AND STAY RES COMMAND
        MOV     DX,1000H                ;64K RESERVED
        INT     21H                     ;DONE
```

Figure 18-11. Continued. *(more)*

```
VECTOR_INIT ENDP

CSEG    ENDS
        END     INITIALIZE
```

Figure 18-11. Continued.

In order to use the symbolic debugging features of SYMDEB, a symbol file must be built in a specific format. The SYMDEB utility MAPSYM performs this function, using the contents of the .MAP file built by LINK. MAPSYM is easy to use because it has only two parameters: the .MAP file and the /L switch (which triggers verbose mode). The symbol table for BADSCOP is built as follows:

```
C>MAPSYM BADSCOP  <Enter>
```

This operation produces a symbol file called BADSCOP.SYM.

Armed with the .SYM file and the usual collection of listing and design notes, the programmer can begin the debugging process using SYMDEB.

The first task is to discover if the BADSCOP TSR is installing correctly. To test this, run the .COM file under SYMDEB by typing

```
C>SYMDEB BADSCOP.SYM BADSCOP.COM  <Enter>
```

Note the order in which operands are passed to SYMDEB — it is not the order that would be expected. All switches (none were used here) must immediately follow the word *SYMDEB*. These switches must be followed in turn by the fully qualified names of any symbol files (in this case, BADSCOP.SYM). Only then is the name of the file to be debugged given. If BADSCOP expected any parameters in the command tail, they would be last. This potential need for command-tail data is the reason the name of the file to be debugged follows the name of the symbol file. SYMDEB knows that the first non-.SYM file it encounters is the file to be loaded; the parameters that follow the filename may be of any form and number.

When SYMDEB begins, it displays

```
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [80286]
```

The debugger identifies itself and then notes the type of CPU it is running on — in this case, an Intel 80286. The Display or Modify Registers command, R, gives the same display that DEBUG gives, with one exception.

```
-R  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=0100   NV UP EI PL NZ NA PO NC
CSEG:INITIALIZE:
1FD0:0100 E90701        JMP  VECTOR_INIT
```

The instruction at CS:IP, JMP, is now preceded by the information that the instruction is at label *INITIALIZE* within segment *CSEG*. An examination of Figure 18-11 shows that this is indeed the case.

To check that all the symbols requested with the PUBLIC statement are present, use the X?* form of the Examine Symbol Map command.

```
-X?*  <Enter>

CSEG: (1FD0)
0100 INITIALIZE    0103 OLD_COMM_INT 0107 COUNT        0109 STATUS
010A PORT          010B BUFPNTR      010D COMMSCOPE    018F CONTROL
020A VECTOR_INIT
```

The display shows that the value of *CSEG* (1FD0H) matches the current value of CS. The offset values shown for the procedure names and data names match the numbers from an assembled listing. Because this is a .COM file, there is only one segment. If there had been other segments — a data segment, for instance — they would have been shown with their values and associated labels and offsets.

The purpose of this test is to determine whether the problems this program is having are caused by an incorrect installation. First, use the Trace Program Execution command, T, to trace through the first few steps.

```
-T7  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020A   NV UP EI PL NZ NA PO NC
CSEG:VECTOR_INIT:
1FD0:020A B435          MOV  AH,35                         ;'5'
AX=3500  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020C   NV UP EI PL NZ NA PO NC
1FD0:020C B014          MOV  AL,14
AX=3514  BX=0000  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1FD0  SS=1FD0  CS=1FD0  IP=020E   NV UP EI PL NZ NA PO NC
1FD0:020E CD21          INT  21  ;Get Interrupt Vector
AX=3514  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0210   NV UP EI PL NZ NA PO NC
1FD0:0210 891E0301      MOV  [OLD_COMM_INT],BX                DS:0103=0000
AX=3514  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0214   NV UP EI PL NZ NA PO NC
1FD0:0214 8CC0          MOV  AX,ES
AX=1567  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0216   NV UP EI PL NZ NA PO NC
1FD0:0216 A30501        MOV  [OLD_COMM_INT+02 (0105)],AX      DS:0105=0000
AX=1567  BX=1375  CX=0133  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0219   NV UP EI PL NZ NA PO NC
1FD0:0219 BA0D01        MOV  DX,010D
```

This part of the program uses Interrupt 21H Function 35H to obtain the current vector for Interrupt 14H. Note that, unlike DEBUG, SYMDEB coasts right through an Interrupt 21H call with no problems. It not only knows enough not to make the call but also displays the type of function call being made, based on the value in AH.

To make sure that the correct vector for the old Interrupt 14H handler has been stored, use the Display Doublewords command, DD, in conjunction with a symbol name.

```
-DD OLD_COMM_INT L1  <Enter>
1FD0:01030  1567:1375
```

This is the correct vector address (1567:1375H). Now trace through the next part of the program, which establishes the new vectors for interrupts.

```
-T8  <Enter>
AX=1567  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=021C   NV UP EI PL NZ NA PO NC
1FD0:021C B425          MOV  AH,25                        ;'%'
AX=2567  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=021E   NV UP EI PL NZ NA PO NC
1FD0:021E B014          MOV  AL,14
AX=2514  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0220   NV UP EI PL NZ NA PO NC
1FD0:0220 CD21          INT  21  ;Set Vector
AX=2514  BX=1375  CX=0133  DX=010D  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0222   NV UP EI PL NZ NA PO NC
1FD0:0222 BA8F01        MOV  DX,018F
AX=2514  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0225   NV UP EI PL NZ NA PO NC
1FD0:0225 B425          MOV  AH,25                        ;'%'
AX=2514  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0227   NV UP EI PL NZ NA PO NC
1FD0:0227 B060          MOV  AL,60                        ;'`'
AX=2560  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=0229   NV UP EI PL NZ NA PO NC
1FD0:0229 CD21          INT  21  ;Set Vector
AX=2560  BX=1375  CX=0133  DX=018F  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1FD0  ES=1567  SS=1FD0  CS=1FD0  IP=022B   NV UP EI PL NZ NA PO NC
1FD0:022B B80031        MOV  AX,3100
```

Examination of these trace steps shows that all went normally. The new Interrupt 14H vector has been established at *COMMSCOPE*; the vector for the new Interrupt 60H has also been correctly installed. Use the Go command, G, to allow the program to continue to termination and then use the Quit command, Q, to exit SYMDEB.

```
-G  <Enter>

Program terminated and stayed resident (0)
-Q  <Enter>
```

SYMDEB displays the information that the program terminated with a completion code of zero and stayed resident. This is as it should be, and the conclusion is that the installation portion of this TSR is running properly. The problem must be in the real-time execution of the program.

Debugging the resident portion of a TSR is complicated but not especially difficult. A simple program is used to exercise the TSR, and it is this program that is debugged. As this driver program exercises the TSR, the tracing process continues into the resident routine.

Because symbol tables exist for the TSR, symbolic debugging can be used to follow its execution.

The driver program will be TESTCOMM, shown in Figure 18-10. To make the program more easily usable by SYMDEB, one line has been added before the first SEGMENT statement:

```
PUBLIC      BEGIN,MAINLOOP,SENDCOMM,TESTCOMM
```

Using the .MAP file produced by LINK, the MAPSYM routine creates TESTCOMM.SYM. TESTCOMM can now be invoked with two symbol files:

```
C>SYMDEB TESTCOMM.SYM BADSCOP.SYM TESTCOMM.EXE  <Enter>
```

SYMDEB will load both symbol files and then load TESTCOMM.EXE. Because the name of the TESTCOMM.SYM file matches the name of the program being loaded, SYMDEB makes TESTCOMM.SYM the active symbol file.

Use the Register command to show that the test program was properly loaded.

```
-R  <Enter>
AX=0000  BX=0000  CX=0133  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=390E  IP=0000   NV UP EI PL NZ NA PO NC
CSEG:BEGIN:
390E:0000 1E            PUSH     DS
```

Then use the Examine Symbol Map command to determine whether the symbol files were loaded correctly. The form X* lists all the symbol maps and their segments; the form X?* lists all the symbols for the current symbol map and segment.

```
-X*  <Enter>
[38FE TESTCOMM]
     [390E CSEG]
 0000 BADSCOP
      0000 CSEG
-X?*  <Enter>

CSEG: (390E)
0000 BEGIN    0004 MAINLOOP 0011 SENDCOMM 0018 TESTCOMM
```

The current symbol map and segment are shown in square brackets. The symbol map for BADSCOP is also present but not selected. Note that there are no values associated with BADSCOP in the listing produced by the X?* command, because all the symbols currently available to SYMDEB are shown and only the symbols in TESTCOMM's *CSEG* are available (that is, TESTCOMM.SYM is the only active symbol file).

Recall that the BADSCOP TSR loaded normally but locked the system up at the first attempt to issue an Interrupt 14H. This behavior indicates that the problem is associated with an Interrupt 14H call. TESTCOMM repeatedly makes the system fail, but which of the Interrupt 14H calls within TESTCOMM is causing the trouble is not known. The most straightforward approach would be to put a breakpoint just before each Interrupt 14H instruction. Use the Disassemble (Unassemble) command, U, to find the location of all Interrupt 14H calls.

```
-U MAINLOOP L19  <Enter>
CSEG:MAINLOOP:
390E:0004 B406          MOV  AH,06
390E:0006 B2FF          MOV  DL,FF
390E:0008 CD21          INT  21
390E:000A 740C          JZ   TESTCOMM
390E:000C 3C03          CMP  AL,03
390E:000E 7501          JNZ  SENDCOMM
390E:0010 CB            RETF
CSEG:SENDCOMM:
390E:0011 B401          MOV  AH,01
390E:0013 BA0000        MOV  DX,BADSCOP!CSEG
390E:0016 CD14          INT  14
CSEG:TESTCOMM:
390E:0018 B403          MOV  AH,03
390E:001A BA0000        MOV  DX,BADSCOP!CSEG
390E:001D CD14          INT  14
390E:001F 80E401        AND  AH,01
390E:0022 74E0          JZ   MAINLOOP
390E:0024 B402          MOV  AH,02
390E:0026 BA0000        MOV  DX,BADSCOP!CSEG
390E:0029 CD14          INT  14
390E:002B B406          MOV  AH,06
390E:002D 8AD0          MOV  DL,AL
390E:002F CD21          INT  21
390E:0031 EBD1          JMP  MAINLOOP
```

The Disassemble request starts at *MAINLOOP* and acts on the next 25 (19H) instructions. SYMDEB displays symbol names instead of numbers whenever it can. However, it does get confused from time to time, so a grain of salt might be needed when reading the disassembly. Notice, for instance, the MOV DX,0 instructions at offsets 13H, 1AH, and 26H. SYMDEB has decided that what is being moved is not zero, but BADSCOP!CSEG. (The ! identifies a mapname in the same way a : defines a segment.) In this case, SYMDEB searched its map tables for an address of zero and found one at *CSEG* in BADSCOP. This segment has the address of zero because it has not been initialized.

Ignoring the name confusions, the disassembly clearly shows the three INT 14H instructions at offsets 16H, 1DH, and 29H. Use the Set Breakpoints command, BP, to set a sticky, or permanent, breakpoint at each of these locations. In this way, any Interrupt 14H call issued by TESTCOMM will be intercepted before it executes. Use the List Breakpoints command, BL, to verify the breakpoints.

```
-BP 16  <Enter>
-BP 1D  <Enter>
-BP 29  <Enter>
-BL  <Enter>
0 e 390E:0016 [CSEG:SENDCOMM+05 (0016)]
1 e 390E:001D [CSEG:TESTCOMM+05 (001D)]
2 e 390E:0029 [CSEG:TESTCOMM+11 (0029)]
```

The List Breakpoints command shows that breakpoint 0 is enabled and set to *SENDCOMM+05*, or CS:0016H. Likewise, breakpoint 1 is at CS:001DH and breakpoint 2 is at CS:0029H. It is important to trap on an Interrupt 14H so that the subsequent actions of the Interrupt 14H service routine can be traced. Now allow the program to execute until it encounters a breakpoint.

```
-G  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=390E  IP=001D   NV UP EI PL ZR NA PE NC
390E:001D CD14           INT     14                           ;BR1
```

The first Interrupt 14H encountered is the one at the second breakpoint, breakpoint 1, as can be seen from the address at which execution broke. Also, SYMDEB was kind enough to include the comment *;BR1* on the disassembled line, indicating that this is Break Request 1. The instruction at this location is a request for serial port status (AH = 3) and the registers are loaded correctly. Execution can now be passed to the TSR by simply executing the current instruction. (Remember that the instruction displayed at a breakpoint has not yet been executed.)

```
-T  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=010D   NV UP DI PL ZR NA PE NC
1FD0:010D 2EF606090101   TEST       Byte Ptr CS:[0109],01            CS:0109=00
```

The single Trace command has moved execution into the TSR. Note that the Interrupt 14H has changed the value of CS and jumped to location 10DH off the new CS. This location contains the first instruction of the *COMMSCOPE* procedure in the TSR. SYMDEB does not know that a different segment is being executed and must be instructed to use a different map table. Use the Open Symbol Map command, XO, to do this, instructing SYMDEB to set the active map table to BADSCOP!.

```
-XO BADSCOP!  <Enter>
-X?*  <Enter>

CSEG: (0000)
0100 INITIALIZE   0103 OLD_COMM_INT 0107 COUNT        0109 STATUS
010A PORT         010B BUFPNTR      010D COMMSCOPE    018F CONTROL
020A VECTOR_INIT
```

The X?* command shows that the BADSCOP symbols are now the current map. They are not usable, however, because the value of *CSEG*—zero—needs to be changed to the current CS register. To correct this, use the SYMDEB Set Symbol Value command, Z. This command can set any symbol in the current map table to any value; the value can be a number, another symbol, or the contents of a register. In this case, set the value of *CSEG* in BADSCOP! to the current contents of the CS register.

```
-Z CSEG CS  <Enter>
-X*  <Enter>
 38FE TESTCOMM
      390E CSEG
[0000 BADSCOP]
      [1FD0 CSEG]
```

The X* command confirms that BADSCOP! is now the selected symbol map and that the *CSEG* within it has the value 1FD0H. The *CSEG* segment in TESTCOMM is an entirely different entity and still has its correct value, which will be valid when the TSR returns.

With the symbols set, the debugging can begin by tracing the first few instructions. Because *COMMSCOPE* is not currently active, the routine should quickly pass the processing on to the old interrupt handler.

```
-T5  <Enter>
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=0113   NV UP DI PL ZR NA PE NC
1FD0:0113 7476           JZ   COMMSCOPE+7E (018B)
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=1FD0  IP=018B   NV UP DI PL ZR NA PE NC
1FD0:018B FF2E0301       JMP  FAR [0103]                        DS:0103=0000
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0000   NV UP DI PL ZR NA PE NC
0000:0000 381E6715       CMP  [1567],BL                         DS:1567=00
AX=0300  BX=0000  CX=0133  DX=0000  SP=00F6  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0004   NV UP DI PL ZR NA PE NC
0000:0004 BC2CE1         MOV  SP,E12C
AX=0300  BX=0000  CX=0133  DX=0000  SP=E12C  BP=0000  SI=0000  DI=0000
DS=38EE  ES=38EE  SS=38FE  CS=0000  IP=0007   NV UP DI PL ZR NA PE NC
0000:0007 2F             DAS
```

STATUS is tested with a mask of 01H at CS:010DH; the test sets the zero flag, indicating that tracing is disabled. The JZ to *COMMSCOPE+7E* (CS:018BH) is taken. At this address is a far jump to the old Interrupt 14H handler at 1567:1375H. The jump is taken and then disaster strikes. Instead of going to the correct address, processing is suddenly at 0000:0000H. Any wild jump is dangerous, but a far jump into low memory is exceptionally so. This explains the system's locking up and requiring a cold boot to recover.

Now that the bug has been caught in the act, it should be a simple matter to determine what went wrong. When the BADSCOP TSR installed itself, it was seen to place the correct offset address at 0103H. Yet whenever the resident portion of the TSR tries to use the value at that address, it finds all zeros. The initialization routine placed the address at the symbol *OLD_COMM_INT* (1FD0:0103H). If that location is examined, the following is found:

```
-DD OLD_COMM_INT L1  <Enter>
1FD0:0103  1567:1375
```

This is the correct address. Why, then, did the programs find zero there? Use the Display Doublewords command to look at the same memory location again, this time using the specific address 0103H rather than a program symbol.

```
-DD 103 L1  <Enter>
38EE:0103  0000:0000
```

The dump of *OLD_COMM_INT* looked at 1FD0:0103H, but the simple dump looked at 38EE:0103H. The explanation is clear when the values of the registers just before the far jump are examined. The CS register contains 1FD0H and the DS register contains 38EEH.

This is the problem—there is a missing CS override on the indirect jump command. When the TSR installed itself, CS and DS were the same because it was a .COM file. When the TSR is entered as the result of an interrupt call, only CS is set; DS remains what it was in the calling program. Without an override, the CPU assumed that the address of the destination of the far call was located at offset 103H from the DS register. This offset, unfortunately, contained zeros, and the program locked up the system.

The problem is now easily corrected. Exit SYMDEB with the Quit command and edit the program source so that the offending line reads

```
OLD_JUMP:
        JMP     CS:OLD_COMM_INT
```

Debugging C programs with SYMDEB

One of SYMDEB's finest features is the ability to debug with source-line data from programs written in Microsoft C, Pascal, and FORTRAN. The actual lines of C or FORTRAN can be included in the debugging display, and the addresses for breakpoints show which line of code the breakpoints are in. Combined with symbolic debugging, these features provide a powerful tool that can significantly reduce debugging time for programs written in a supported language.

The following rather complicated case illustrates SYMDEB at its best. The program BADSCOP from the previous example was not completely debugged. Although the patch to the BADSCOP code at *OLD_JUMP:* did correct the disastrous problem that caused the system to lock up, running the program in a realistic test situation reveals that a subtle problem still remains that might be in either BADSCOP or one of the support programs.

Before we investigate the problem, a quick review of the programs in the COMMSCOP system is in order. At the heart of the system is the Interrupt 14H intercept program COMMSCOP. When executed, this program installs itself as a TSR and intercepts all Interrupt 14H calls. (The incorrect version of the COMMSCOP program is called BADSCOP.) The installed COMMSCOP TSR passes all Interrupt 14H calls on to the real service routine in the ROM BIOS until it is commanded to start tracing. The COMMSCMD routine controls tracing. This control routine can request that COMMSCOP start, stop, or resume tracing for a specific serial port. These commands are facilitated through Interrupt 60H, which is recognized by the COMMSCOP TSR as a command request. When tracing is started, the trace buffer is emptied by zeroing the trace count and setting the buffer pointer to the first buffer location. When tracing is stopped by COMMSCMD's STOP command, a marker is placed in the buffer to indicate the end of a trace segment. Tracing can be resumed with COMMSCMD's RESUME command. Resuming a trace preserves collected data and places new trace data after the marker in the trace buffer. The RESUME command differs from the START command in that the buffer is not emptied.

Now the problem: When the serial data tracing is started with COMMSCMD (*see* Figure 18-5), data is collected normally. When COMMSCMD issues a STOP command and the data is displayed with COMMDUMP (*see* Figure 18-7), the data appears normal. The traced data ends with a stop mark just as it should. However, the RESUME command of

COMMSCMD causes the stop mark to be overwritten with collected data. After this, whenever COMMDUMP displays data an extra byte appears at the end of the data. The problem could be with either BADSCOP or COMMSCMD. SYMDEB has the facilities to debug both the routines at once.

The first step in the debugging process is, as usual, to gather all the listings and design documentation. As a part of this process, the symbol tables needed for SYMDEB must be prepared. The process of preparing a symbol table for BADSCOP has already been explained; however, preparing the SYMDEB input and supporting listings for a C program is slightly more complicated.

First, when the C program is compiled, three switches must be specified. (C switches are case sensitive and must be entered exactly as shown.)

```
C>MSC /Fc /Zd /Od COMMSCMD;  <Enter>
```

The /Zd switch produces an object file containing line-number information that corresponds to the line numbers of the source file. The /Od switch disables optimization that involves complex code rearrangement; localized optimization, peephole optimization, and other simple forms of optimization are still performed. The /Od switch is not required, but code rearrangement can make the resulting object code more difficult to debug.

The /Fc switch invokes a feature of C that is especially important for debugging with SYMDEB: a listing that contains the C source lines and the generated assembler code intermixed. The file is a .COD file; the command line shown above would produce the file COMMSCMD.COD. Figure 18-12 shows the contents of COMMSCMD.COD.

```
;       Static Name Aliases
;
;       $S142_commands  EQU     commands
        TITLE   commscmd
;       NAME    commscmd.C

        .287
_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
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
EXTRN   _int86:NEAR
EXTRN   _printf:NEAR
EXTRN   _stricmp:NEAR
EXTRN   _atoi:NEAR
EXTRN   __chkstk:NEAR
_DATA      SEGMENT
```

Figure 18-12. COMMSCMD.COD. *(more)*

```
$SG148  DB      'STOP',  00h
$SG151  DB      'START',  00h
$SG154  DB      'RESUME',  00h
$SG157  DB      0ah, 'Communications tracing %s for port COM%1d:',  0ah,  00h
$S142_commands  DB      'STOPPED',  00h
        ORG     $+2
        DB      'STARTED',  00h
        ORG     $+2
        DB      'RESUMED',  00h
        ORG     $+2
_DATA      ENDS
_TEXT      SEGMENT
;¦*** /***********************************************************************
;¦***  *                                                                     *
;¦***  *  COMMSCMD                                                           *
;¦***  *                                                                     *
;¦***  *  This routine controls the COMMSCOP program that has been in-       *
;¦***  *  stalled as a resident routine.  The operation performed is de-     *
;¦***  *  termined by the command line.  The COMMSCMD program is invoked     *
;¦***  *  as follows:                                                        *
;¦***  *                                                                     *
;¦***  *                COMMSCMD [[cmd][ port]]                              *
;¦***  *                                                                     *
;¦***  *  where cmd is the command to be executed                            *
;¦***  *            STOP   -- stop trace                                      *
;¦***  *            START  -- flush trace buffer and start trace             *
;¦***  *            RESUME -- resume a stopped trace                         *
;¦***  *        port is the COMM port to be traced (1=COM1, 2=COM2, etc.)    *
;¦***  *                                                                     *
;¦***  *  If cmd is omitted, STOP is assumed.  If port is omitted, 1 is      *
;¦***  *  assumed.                                                           *
;¦***  *                                                                     *
;¦***  ***********************************************************************/
;¦***
;¦*** #include <stdlib.h>
;¦*** #include <stdio.h>
;¦*** #include <dos.h>
;¦*** #define COMMSCMD 0x60
;¦***
;¦*** main(argc, argv)
;¦*** int argc;
; Line 29
        PUBLIC _main
_main   PROC NEAR
        *** 000000     55                      push   bp
        *** 000001     8b ec                   mov    bp,sp
        *** 000003     b8 22 00                mov    ax,34
        *** 000006     e8 00 00                call   __chkstk
        *** 000009     57                      push   di
        *** 00000a     56                      push   si
```

Figure 18-12. Continued. *(more)*

```
;¦*** char *argv[];
;¦*** {
; Line 31
;       argc = 4
;       argv = 6
;       cmd = -4
;       port = -6
;       result = -2
;       inregs = -34
;       outregs = -20
;¦***     int cmd, port, result;
;¦***     static char commands[3] [10] = {"STOPPED", "STARTED", "RESUMED"};
;¦***     union REGS inregs, outregs;
;¦***
;¦***     cmd = 0;
; Line 36
        *** 00000b     c7 46 fc 00 00          mov     WORD PTR [bp-4],0    ;cmd
;¦***     port = 0;
; Line 37
        *** 000010     c7 46 fa 00 00          mov     WORD PTR [bp-6],0    ;port
;¦***
;¦***     if (argc > 1)
; Line 39
        *** 000015     83 7e 04 01             cmp     WORD PTR [bp+4],1    ;argc
        *** 000019     7f 03                   jg      $JCC25
        *** 00001b     e9 5d 00                jmp     $I145
                                    $JCC25:
;¦***      {
; Line 40
;¦***      if (0 == stricmp(argv[1], "STOP"))
; Line 41
        *** 00001e     b8 00 00                mov     ax,OFFSET DGROUP:$SG148
        *** 000021     50                      push    ax
        *** 000022     8b 5e 06                mov     bx,[bp+6]            ;argv
        *** 000025     ff 77 02                push    WORD PTR [bx+2]
        *** 000028     e8 00 00                call    _stricmp
        *** 00002b     83 c4 04                add     sp,4
        *** 00002e     3d 00 00                cmp     ax,0
        *** 000031     74 03                   je      $JCC49
        *** 000033     e9 08 00                jmp     $I147
                                    $JCC49:
;¦***              cmd = 0;
; Line 42
        *** 000036      c7 46 fc 00 00          mov     WORD PTR [bp-4],0   ;cmd
;¦***          else if (0 == stricmp(argv[1], "START"))
```

Figure 18-12. Continued. *(more)*

```
; Line 43
        *** 00003b      e9 3d 00                  jmp     $I149
                                       $I147:
        *** 00003e      b8 05 00                  mov     ax,OFFSET DGROUP:$SG151
        *** 000041      50                        push    ax
        *** 000042      8b 5e 06                  mov     bx,[bp+6]       ;argv
        *** 000045      ff 77 02                  push    WORD PTR [bx+2]
        *** 000048      e8 00 00                  call    _stricmp
        *** 00004b      83 c4 04                  add     sp,4
        *** 00004e      3d 00 00                  cmp     ax,0
        *** 000051      74 03                     je      $JCC81
        *** 000053      e9 08 00                  jmp     $I150
                                      $JCC81:
;¦***            cmd = 1;
; Line 44
        *** 000056      c7 46 fc 01 00            mov     WORD PTR [bp-4],1       ;cmd
;¦***         else if (0 == stricmp(argv[1], "RESUME"))
; Line 45
        *** 00005b      e9 1d 00                  jmp     $I152
                                       $I150:
        *** 00005e      b8 0b 00                  mov     ax,OFFSET DGROUP:$SG154
        *** 000061      50                        push    ax
        *** 000062      8b 5e 06                  mov     bx,[bp+6]       ;argv
        *** 000065      ff 77 02                  push    WORD PTR [bx+2]
        *** 000068      e8 00 00                  call    _stricmp
        *** 00006b      83 c4 04                  add     sp,4
        *** 00006e      3d 00 00                  cmp     ax,0
        *** 000071      74 03                     je      $JCC113
        *** 000073      e9 05 00                  jmp     $I153
                                     $JCC113:
;¦***            cmd = 2;
; Line 46
        *** 000076      c7 46 fc 02 00            mov     WORD PTR [bp-4],2       ;cmd
;¦***         }
; Line 47
                                       $I153:
                                       $I152:
                                       $I149:
;¦***
;¦***     if (argc == 3)
; Line 49
                                       $I145:
        *** 00007b      83 7e 04 03               cmp     WORD PTR [bp+4],3       ;argc
        *** 00007f      74 03                     je      $JCC127
        *** 000081      e9 1b 00                  jmp     $I155
                                     $JCC127:
;¦***         {
; Line 50
;¦***         port = atoi(argv[2]);
```

Figure 18-12. Continued. *(more)*

```
; Line 51
        *** 000084     8b 5e 06                  mov     bx,[bp+6]             ;argv
        *** 000087     ff 77 04                  push    WORD PTR [bx+4]
        *** 00008a     e8 00 00                  call    _atoi
        *** 00008d     83 c4 02                  add     sp,2
        *** 000090     89 46 fa                  mov     [bp-6],ax             ;port
;¦***          if (port > 0)
; Line 52
        *** 000093     83 7e fa 00               cmp     WORD PTR [bp-6],0     ;port
        *** 000097     7f 03                     jg      $JCC151
        *** 000099     e9 03 00                  jmp     $I156
                                    $JCC151:
;¦***               port = port-1;
; Line 53
        *** 00009c     ff 4e fa                  dec     WORD PTR [bp-6]       ;port
;¦***          }
; Line 54
                                      $I156:
;¦***
;¦***      inregs.h.ah = cmd;
; Line 56
                                      $I155:
        *** 00009f     8a 46 fc                  mov     al,[bp-4]             ;cmd
        *** 0000a2     88 46 df                  mov     [bp-33],al
;¦***      inregs.x.dx = port;
; Line 57
        *** 0000a5     8b 46 fa                  mov     ax,[bp-6]             ;port
        *** 0000a8     89 46 e4                  mov     [bp-28],ax
;¦***      result = int86(COMMCMD, &inregs, &outregs);
; Line 58
        *** 0000ab     8d 46 ec                  lea     ax,[bp-20]            ;outregs
        *** 0000ae     50                        push    ax
        *** 0000af     8d 46 de                  lea     ax,[bp-34]            ;inregs
        *** 0000b2     50                        push    ax
        *** 0000b3     b8 60 00                  mov     ax,96
        *** 0000b6     50                        push    ax
        *** 0000b7     e8 00 00                  call    _int86
        *** 0000ba     83 c4 06                  add     sp,6
        *** 0000bd     89 46 fe                  mov     [bp-2],ax             ;result
;¦***
;¦***
;¦***      printf("\nCommunications tracing %s for port COM%1d:\n",
;¦***              commands[cmd], port + 1);
; Line 62
        *** 0000c0     8b 46 fa                  mov     ax,[bp-6]             ;port
        *** 0000c3     40                        inc     ax
        *** 0000c4     50                        push    ax
        *** 0000c5     8b 46 fc                  mov     ax,[bp-4]             ;cmd
        *** 0000c8     8b c8                     mov     cx,ax
        *** 0000ca     d1 e0                     shl     ax,1
        *** 0000cc     d1 e0                     shl     ax,1
        *** 0000ce     03 c1                     add     ax,cx
        *** 0000d0     d1 e0                     shl     ax,1
```

Figure 18-12. Continued. *(more)*

```
        *** 0000d2      05 40 00                    add     ax,OFFSET DGROUP:$S142_commands
        *** 0000d5      50                          push    ax
        *** 0000d6      b8 12 00                    mov     ax,OFFSET DGROUP:$SG157
        *** 0000d9      50                          push    ax
        *** 0000da      e8 00 00                    call    _printf
        *** 0000dd      83 c4 06                    add     sp,6
;|*** }
; Line 63
                                    $EX138:
        *** 0000e0      5e                          pop     si
        *** 0000e1      5f                          pop     di
        *** 0000e2      8b e5                       mov     sp,bp
        *** 0000e4      5d                          pop     bp
        *** 0000e5      c3                          ret

_main     ENDP
_TEXT     ENDS
END
```

Figure 18-12. Continued.

After the C program is compiled, it must be linked using the /LI switch to indicate that the line number information is to be maintained:

```
C>LINK COMMSCMD /MAP /LI;  <Enter>
```

The /MAP switch is still required to generate a map file of public names for use in building the symbol file, which is created in the usual manner:

```
C>MAPSYM COMMSCMD  <Enter>
```

Everything needed to debug COMMSCMD and BADSCOP is now available. The first test is an attempt to start tracing. To invoke SYMDEB, type

```
C>SYMDEB COMMSCMD.SYM BADSCOP.SYM COMMSCMD.EXE START 1  <Enter>
```

SYMDEB first loads the symbol files for COMMSCMD and BADSCOP and then loads the .EXE file for COMMSCMD. BADSCOP is already in memory, having been loaded by simply running it. (It then stays resident.) The last two entries in the command line load the command tail for COMMSCMD with a start request for COM1. SYMDEB responds with

```
Microsoft (R) Symbolic Debug Utility  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [80286]
```

Use the Register and Examine Symbol Map commands to display the initial register values and symbol table information.

```
-R  <Enter>
AX=0000  BX=0000  CX=1928  DX=0000  SP=0800  BP=0000  SI=0000  DI=0000
DS=2CA0  ES=2CA0  SS=2E85  CS=2CB0  IP=010F   NV UP EI PL NZ NA PO NC
_TEXT:__astart:
2CB0:010F B430           MOV  AH,30                         ;'0'
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       0000 CSEG
-X?*  <Enter>
9876 __acrtused    9876 __acrtmsg
_TEXT: (2CB0)
0010 _main         00F6 _atoi
00F9 __chkstk      010F __astart      01AB __cintDIV     01AE __amsg_exit
01B9 _int86        023A _printf       0270 _strcmpi      0270 _stricmp
02C2 __stbuf       0361 __ftbuf       03E7 __catox       043C __nullcheck
0458 __cinit       0507 _exit         051E __exit        054A __ctermsub
0572 __dosret0     057A __dosretax    0586 __maperror    05BA __NMSG_TEXT
05EA __NMSG_WRITE  0613 __output      0E22 __setargv     0F07 __setenvp
0F6D __flsbuf      1098 __fassign     1098 __cropzeros   1098 __positive
1098 __forcdecpt   1098 __cfltcvt     109B _fflush       1103 _isatty
1125 __myalloc     1167 _strlen       1182 _ultoa        118C __fptrap
1192 _flushall     11C3 _free         11C3 __nfree       11D1 _malloc
11D1 __nmalloc     1217 _write        12F1 __cltoasub    12FD __cxtoa
1351 __amalloc     1432 __amexpand    146C __amlink      148E __amallocbrk
14AD _brkctl
DGROUP: (2E08)
0094 STKHQQ        0096 __asizds      0098 __atopsp
009A __abrktb      00EA __abrktbe     00EA __abrkp       00EC __iob
018C __iob2        0204 __lastiob     0212 __aintdiv     0216 __fac
021E _errno        0220 __umaskval    0222 __pspadr      0224 __psp
0226 __osmajor     0226 __dosvermajor 0227 __osminor     0227 __dosverminor
0228 __oserr       0228 __doserrno    022A __osfile      023E ___argc
0240 ___argv       0242 _environ      0244 __child       0246 __csigtab
0278 __cflush      027A __asegds      0286 __aseg1       0288 __asegn
028A __asegr       028C __amblksiz    0292 __fpinit      03A8 _edata
03D0 __bufout      05D0 __bufin       07D0 _end
```

The Register command shows that the first instruction to be executed will be at symbol *__astart* in the _TEXT segment. (Note that C puts a single underscore in front of all public library and routine names; a double underscore indicates routines for C's internal use.) The Examine Symbol Map command reveals that the symbol map COMMSCMD! has two segments, *_TEXT* and *DGROUP*, with *_TEXT* currently selected. The segment in BADSCOP!, *CSEG*, has no value assigned to it because SYMDEB doesn't know where it is; one of the debugging tasks is to determine the location of *CSEG*.

C places initialization and preamble code at the front of its object modules. This code can be skipped during debugging, so this example begins at the label *_main*. Examination of the code at this label using the Disassemble command reveals the following:

```
-U _main  <Enter>
commscmd.C
29:  int argc;
_TEXT:_main:
2CB0:0010 55                PUSH     BP
2CB0:0011 8BEC              MOV      BP,SP
2CB0:0013 B82200            MOV      AX,0022
2CB0:0016 E8E000            CALL     __chkstk
2CB0:0019 57                PUSH     DI
```

This disassembly shows the way source-line information is displayed. These instructions are generated by line 29 of COMMSCMD.C. When the disassembly is compared with the listing in Figure 18-12, the same instructions are seen. However, their addresses are different. The addresses in the disassembly are relative to the start of the segment *_TEXT*, but the addresses in the listing are relative to the start of *_main*. SYMDEB allows address references to be made relative to a symbol, so breakpoints can be set as displacements from *_main* and the addresses shown in the listing can be used.

Because the location of the problem being debugged is not known, breakpoints must be placed strategically throughout COMMSCMD to trace the execution of the program. Use the Set Breakpoints command to set the breakpoints.

```
-BP _main+1e  <Enter>
-BP _main+36  <Enter>
-BP _main+56  <Enter>
-BP _main+76  <Enter>
-BP _main+7b  <Enter>
-BP _main+9c  <Enter>
-BP _main+b7  <Enter>
-BP _main+e5  <Enter>
-BL  <Enter>
0 e 2CB0:002E [_TEXT:_main+1E (002E)] commscmd.C:41
1 e 2CB0:0046 [_TEXT:_main+36 (0046)] commscmd.C:42
2 e 2CB0:0066 [_TEXT:_main+56 (0066)] commscmd.C:44
3 e 2CB0:0086 [_TEXT:_main+76 (0086)] commscmd.C:46
4 e 2CB0:008B [_TEXT:_main+7B (008B)] commscmd.C:49
5 e 2CB0:00AC [_TEXT:_main+9C (00AC)] commscmd.C:53
6 e 2CB0:00C7 [_TEXT:_main+B7 (00C7)] commscmd.C:58
7 e 2CB0:00F5 [_TEXT:_main+E5 (00F5)] commscmd.C:63
```

The List Breakpoints command shows the breakpoint addresses in three ways: first the absolute segment:offset address, then the displacement from the label *_main*, and finally the line number in COMMSCMD.C.

The first part of the COMMSCMD program decodes the arguments and sets the appropriate values for *cmd* and *port*. If there are no arguments, this decoding is skipped; if there are arguments, the decoding begins at line 41, so the first breakpoint is set there. If the criterion of line 41 is met (the first argument is STOP), then line 42 is executed. The second breakpoint is set there. Reaching the second breakpoint means that a STOP command was properly decoded. If the command was not STOP, execution continues at line 43. If this test is passed, line 44 is executed. This is the location of the third breakpoint. If the test at line 44 fails but the one at line 45 is passed, then the breakpoint at line 46 is executed. Whether or not one of the tests passes, execution ends up at line 49. At this point, the program tests for the presence of a second operand. If there is a second operand, execution traps at line 53, where the program decrements the port number to put it in the proper form for the Interrupt 60H handler. Execution will then always stop in line 58, just before the call to *_int86.* (*_int86* is a library routine that loads registers and executes INT instructions.)

When the program is run with *START1* in the command tail, it gives the following results:

```
-G  <Enter>
AX=0022  BX=0F82  CX=0019  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:               if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600         MOV        AX,0036                     ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0066   NV UP EI PL ZR NA PE NC
44:                      cmd = 1;
2CB0:0066 C746FC0100     MOV        Word Ptr [BP-04],0001       ;BR2  SS:0FA0=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:           if (argc == 3)
2CB0:008B 837E0403       CMP       Word Ptr [BP+04],+03          ;BR4 SS:0FA8=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
5             port = port-1;
2CB0:00AC FF4EFA         DEC        Word Ptr [BP-06]            ;BR5  SS:0F9E=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F78  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00         CALL      _int86                        ;BR6
```

The first break occurs at line 41, indicating that one or more arguments were present in the command line. The next break is at line 44, where the program sets the *cmd* code for Interrupt 60H to 1, the correct value for a start request. The next break occurs at line 49, where the program checks the number of arguments. If this number is 3, then there is a second argument in the command line. (Remember that, in C, the first argument is the name of the routine, so an argument count of 3 actually means that there are 2 arguments present.) The number of arguments is at BP+04, or SS:0FA8H, and it is indeed 3. Therefore, the next break is at line 53. The program decrements the current value of *port*, leaving a value of 0, which is what Interrupt 60H expects to see for COM1.

Continuing execution causes a break just before the call to *_int86.* To validate that the Interrupt 60H call is being made correctly, set a breakpoint just before the INT 60H instruction is issued. Unfortunately, no listing of *_int86* is available, so no alternative exists but to trace the execution of the routine until the INT instruction is issued. The details of the processing are of no interest to this debugging session, so they can be ignored until an INT 60H is seen. (The trace offers a great deal of information about how C interfaces with subroutines. Studying the trace would be educational but is beyond the scope of this example.)

```
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F76  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01B9   NV UP EI PL ZR NA PE NC
_TEXT:_int86:
2CB0:01B9 55            PUSH    BP
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F74  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BA   NV UP EI PL ZR NA PE NC
2CB0:01BA 8BEC          MOV     BP,SP
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F74  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BC   NV UP EI PL ZR NA PE NC
2CB0:01BC 56            PUSH    SI
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F72  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BD   NV UP EI PL ZR NA PE NC
2CB0:01BD 57            PUSH    DI
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F70  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01BE   NV UP EI PL ZR NA PE NC
2CB0:01BE 83EC0A        SUB     SP,+0A
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C1   NV UP EI PL NZ AC PE NC
2CB0:01C1 C646F6CD      MOV     Byte Ptr [BP-0A],CD               SS:0F6A=BE
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C5   NV UP EI PL NZ AC PE NC
2CB0:01C5 8B4604        MOV     AX,[BP+04]                        SS:0F78=0060
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01C8   NV UP EI PL NZ AC PE NC
2CB0:01C8 8846F7        MOV     [BP-09],AL                        SS:0F6B=01
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CB   NV UP EI PL NZ AC PE NC
2CB0:01CB 3C25          CMP     AL,25                         ;'%'
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CD   NV UP EI PL NZ AC PO NC
2CB0:01CD 740A          JZ      _int86+20 (01D9)
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01CF   NV UP EI PL NZ AC PO NC
2CB0:01CF 3C26          CMP     AL,26                         ;'&'
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D1   NV UP EI PL NZ AC PE NC
2CB0:01D1 7406          JZ      _int86+20 (01D9)
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D3   NV UP EI PL NZ AC PE NC
2CB0:01D3 C646F8CB      MOV     Byte Ptr [BP-08],CB               SS:0F6C=B0
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01D7   NV UP EI PL NZ AC PE NC
```

(more)

```
2CB0:01D7 EB0C          JMP     _int86+2C (01E5)
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01E5   NV UP EI PL NZ AC PE NC
2CB0:01E5 8C56F4        MOV     [BP-0C],SS                        SS:0F68=0F74
-T 5  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01E8   NV UP EI PL NZ AC PE NC
2CB0:01E8 8D46F6        LEA     AX,[BP-0A]                        SS:0F6A=60CD
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01EB   NV UP EI PL NZ AC PE NC
2CB0:01EB 8946F2        MOV     [BP-0E],AX                        SS:0F66=0060
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01EE   NV UP EI PL NZ AC PE NC
2CB0:01EE 8B7E06        MOV     DI,[BP+06]                        SS:0F7A=0F82
AX=0F6A  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F1   NV UP EI PL NZ AC PE NC
2CB0:01F1 8B05          MOV     AX,[DI]                           DS:0F82=0100
AX=0100  BX=00D0  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F3   NV UP EI PL NZ AC PE NC
2CB0:01F3 8B5D02        MOV     BX,[DI+02]                        DS:0F84=0000
-T 5  <Enter>
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F6   NV UP EI PL NZ AC PE NC
2CB0:01F6 8B4D04        MOV     CX,[DI+04]                        DS:0F86=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01F9   NV UP EI PL NZ AC PE NC
2CB0:01F9 8B5506        MOV     DX,[DI+06]                        DS:0F88=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0089  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01FC   NV UP EI PL NZ AC PE NC
2CB0:01FC 8B7508        MOV     SI,[DI+08]                        DS:0F8A=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=0F82
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=01FF   NV UP EI PL NZ AC PE NC
2CB0:01FF 8B7D0A        MOV     DI,[DI+0A]                        DS:0F8C=0000
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0202   NV UP EI PL NZ AC PE NC
2CB0:0202 55            PUSH     BP
-T 5  <Enter>
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F64  BP=0F74  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0203   NV UP EI PL NZ AC PE NC
2CB0:0203 83ED0E        SUB     BP,+0E
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F64  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0206   NV UP EI PL NZ AC PE NC
2CB0:0206 FF5E00        CALL     FAR [BP+00]                       SS:0F66=0F6A
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F60  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6A   NV UP EI PL NZ AC PE NC
2E08:0F6A CD60          INT     60
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PE NC
1313:0190 80FC00        CMP     AH,00
AX=0100  BX=0000  CX=0000  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=0000
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL NZ NA PO NC
1313:0193 7521          JNZ     01B6
```

When the Interrupt 60H call is encountered at offset 0F6AH, the values passed to it can be checked. AH contains 1 and DX contains 0—the correct values for START COM1.

In order to use the symbols for BADSCOP, use the Open Symbol Map command, XO, to switch to the correct symbol map. Then, because the value of *CSEG* is not defined in the map, use the Set Symbol Value command to set *CSEG* to the current value of CS. (CS was changed to the correct value for BADSCOP when the program executed the INT 60H instruction.)

```
-XO BADSCOP!  <Enter>
-Z CSEG CS    <Enter>
-X?*          <Enter>

CSEG: (1313)
0100 INITIALIZE   0103 OLD_COMM_INT 0107 COUNT       0109 STATUS
010A PORT         010B BUFPNTR      010D COMSCOPE    0190 CONTROL
020A VECTOR_INIT
```

Because the BADSCOP symbols now have meaning, a great deal of trouble can be avoided by setting a breakpoint at *CONTROL*, the entry point for Interrupt 60H, so that it will no longer be necessary to trace the *_int86* routine to find the INT 60H command. Execution will automatically stop when the Interrupt 60H handler is entered.

```
-BP CONTROL  <Enter>
-BL  <Enter>
0 e 2CB0:002E [COMMSCMD!_TEXT:_main+1E (002E)] commscmd.C:41
1 e 2CB0:0046 [COMMSCMD!_TEXT:_main+36 (0046)] commscmd.C:42
2 e 2CB0:0066 [COMMSCMD!_TEXT:_main+56 (0066)] commscmd.C:44
3 e 2CB0:0086 [COMMSCMD!_TEXT:_main+76 (0086)] commscmd.C:46
4 e 2CB0:008B [COMMSCMD!_TEXT:_main+7B (008B)] commscmd.C:49
5 e 2CB0:00AC [COMMSCMD!_TEXT:_main+9C (00AC)] commscmd.C:53
6 e 2CB0:00C7 [COMMSCMD!_TEXT:_main+B7 (00C7)] commscmd.C:58
7 e 2CB0:00F5 [COMMSCMD!_TEXT:_main+E5 (00F5)] commscmd.C:63
8 e 1313:0190 [CSEGS:CONTROL]
```

With the housekeeping tasks done, the business of debugging BADSCOP can begin. The first thing *CONTROL* does is check for a stop request. If no stop request is present, the routine jumps to the check for a start request. (The first test and jump were already complete when the trace ended above.) The test for a start request is passed. *CONTROL* places the port number in a local variable, resets the buffer pointer and the buffer count, and turns tracing status on. With all this complete, *CONTROL* returns.

```
-T 5  <Enter>
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B6   NV UP DI PL NZ NA PO NC
1313:01B6 80FC01          CMP     AH,01
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B9   NV UP DI PL ZR NA PE NC
1313:01B9 751C            JNZ     CONTROL+47 (01D7)
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01BB   NV UP DI PL ZR NA PE NC
1313:01BB 2E88160A01      MOV     CS:[PORT],DL                    CS:010A=00
```

(more)

```
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01C0   NV UP DI PL ZR NA PE NC
1313:01C0 2EC7060B010202 MOV     Word Ptr CS:[BUFPNTR],VECTOR_INIT (0209) CS:010B=0202
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01C7   NV UP DI PL ZR NA PE NC
1313:01C7 2EC70607010000 MOV     Word Ptr CS:[COUNT],0000          CS:0107=0002
-T 5  <Enter>
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01CE   NV UP DI PL ZR NA PE NC
1313:01CE 2EC606090101   MOV     Byte Ptr CS:[STATUS],01           CS:0109=01
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01D4   NV UP DI PL ZR NA PE NC
1313:01D4 EB2B           JMP     CONTROL+71 (0201)
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL ZR NA PE NC
1313:0201 CF             IRET
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F60  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6C   NV UP EI PL NZ AC PE NC
2E08:0F6C CB             RETF
AX=01BB  BX=E81E  CX=3F48  DX=0000  SP=0F64  BP=0F66  SI=1CE7  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0209   NV UP EI PL NZ AC PE NC
2CB0:0209 5D             POP     BP
```

As can be seen from the trace, *CONTROL* performed correctly, so execution of the routine can continue.

```
-G  <Enter>
Communications tracing STARTED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA6  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ NA PE NC
2CB0:00F5 C3             RET                                ;BR7
```

COMMSCMD has written the message to the user and trapped at the breakpoint set at the end of *_main*. The Examine Symbol Map command now shows that SYMDEB has automatically switched to the symbol map for COMMSCMD.

```
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
```

No problems have been encountered with the START command; now the same process of checking COMMSCMD and BADSCOP must be repeated for the STOP command. (Even if problems had been found with the START command, it would be imprudent not to test the other commands — they could have errors, too.) SYMDEB could be exited and restarted with new commands, but this would mean the loss of the painfully created set of breakpoints. Instead, a new copy of COMMSCMD is loaded without leaving SYMDEB. One problem with this, however, is that when SYMDEB loads an .EXE file, it adds the value of the initial CS register to the addresses of the segments in the symbol map whose name matches the .EXE file. This is fine the first time the program loads, but the second time, all the values are doubled and therefore incorrect. To avoid this error, the addresses must be adjusted before the load. Use the Set Symbol Value command to subtract CS from each segment name in COMMSCMD!. The Examine Symbol Map command shows the new values.

```
-Z _TEXT _TEXT-CS  <Enter>
-Z DGROUP DGROUP-CS  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [0000 _TEXT]
       0158 DGROUP
 0000 BADSCOP
       1313 CSEG
```

The Name File or Command-Tail Parameters command, N, and the Load File or Sectors command, L, can now be used to load a new copy of COMMSCMD.EXE.

```
-N COMMSCMD.EXE  <Enter>
-L  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
```

Notice that the segment values inside COMMSCMD! are the same as they were when the program was first loaded. Use the Name command again, this time to set the command tail to contain a STOP command for COM1. The breakpoint table from the first execution is still set, so the program can now be traced in the same way.

```
-N STOP 1  <Enter>
-G  <Enter>
AX=0022  BX=0F84  CX=0019  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:             if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600         MOV     AX,0036                       ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0046   NV UP EI PL ZR NA PE NC
42:                 cmd = 0;
2CB0:0046 C746FC0000     MOV     Word Ptr [BP-04],0000         ;BR1 SS:0FA2=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:         if (argc == 3)
2CB0:008B 837E0403       CMP     Word Ptr [BP+04],+03          ;BR4 SS:0FAA=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F80  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
53:                 port = port-1;
```

(more)

```
2CB0:00AC FF4EFA         DEC     Word Ptr [BP-06]            ;BR5 SS:0FA0=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F7A  BP=0FA6  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00         CALL    _int86                       ;BR6
```

COMMSCMD detected that this is a stop request for COM1 and set the arguments for *_int86* correctly. Because a breakpoint is now set at *CONTROL*, tracing until the Interrupt 60H call is found is not necessary. Simply executing the program will cause it to stop at *CONTROL*.

```
-G  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PO NC
CSEG:CONTROL:
1313:0190 80FC00         CMP     AH,00                        ;BR8
```

The registers are set correctly for a stop request on COM1 (AH = 0, DX = 0). The routine can now be traced to check for correct operation. First, however, a quick look at the symbol maps shows that SYMDEB has automatically switched to BADSCOP's symbols.

```
-X*  <Enter>
 2CB0 COMMSCMD
       2CB0 _TEXT
       2E08 DGROUP
[0000 BADSCOP]
      [1313 CSEG]
-T 5  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL ZR NA PE NC
1313:0193 7521           JNZ     CONTROL+26 (01B6)
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0195   NV UP DI PL ZR NA PE NC
1313:0195 1E             PUSH     DS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5A  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0196   NV UP DI PL ZR NA PE NC
1313:0196 53             PUSH     BX
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0197   NV UP DI PL ZR NA PE NC
1313:0197 0E             PUSH     CS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F56  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0198   NV UP DI PL ZR NA PE NC
1313:0198 1F             POP     DS
-T 5  <Enter>
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=0199   NV UP DI PL ZR NA PE NC
1313:0199 C606090100     MOV     Byte Ptr [STATUS],00           DS:0109=01
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=019E   NV UP DI PL ZR NA PE NC
1313:019E 8B1E0B01       MOV     BX,[BUFPNTR]                    DS:010B=0202
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A2   NV UP DI PL ZR NA PE NC
```

(more)

```
1313:01A2 C60780         MOV     Byte Ptr [BX],80                 DS:0202=80
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A5   NV UP DI PL ZR NA PE NC
1313:01A5 C64701FF       MOV     Byte Ptr [BX+01],FF              DS:0203=FF
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01A9   NV UP DI PL ZR NA PE NC
1313:01A9 FF060701       INC     Word Ptr [COUNT]                 DS:0107=0000
-T 5  <Enter>
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01AD   NV UP DI PL NZ NA PO NC
1313:01AD FF060701       INC     Word Ptr [COUNT]                 DS:0107=0001
AX=001E  BX=0202  CX=0000  DX=0000  SP=0F58  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01B1   NV UP DI PL NZ NA PO NC
1313:01B1 5B             POP     BX
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5A  BP=0F68  SI=7400  DI=E903
DS=1313  ES=2E08  SS=2E08  CS=1313  IP=01B2   NV UP DI PL NZ NA PO NC
1313:01B2 1F             POP     DS
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B3   NV UP DI PL NZ NA PO NC
1313:01B3 EB4C           JMP     CONTROL+71 (0201)
AX=001E  BX=3F48  CX=0000  DX=0000  SP=0F5C  BP=0F68  SI=7400  DI=E903
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL NZ NA PO NC
1313:0201 CF             IRET
```

CONTROL correctly detected that this was a stop request. It then saved the user's registers and established a DS equal to CS. (Remember that BADSCOP is a .COM file and CS = DS = SS.) Having done this, the routine moves a zero to *STATUS*, which turns the trace off. It then moves 80H FFH to the buffer to indicate the end of a trace session, increments *COUNT* to allow for the new entry, and restores the user's registers. What it does *not* do is increment the buffer pointer to allow for the stop marker. This behavior is entirely consistent with the observed phenomena: When a trace is stopped and resumed, the stop marker is missing and the count is one too high. The fix is to add

```
INC      BX           ;INCREMENT BUFFER POINTER
INC      BX           ; .
MOV      BUFPNTR,BX   ; .
```

to the *CONTROL* procedure before the registers are restored. (Insert these lines later with your favorite editor.)

Even though the bug has been found, the rest of the routine should be checked for other possible bugs.

```
-G  <Enter>
Communications tracing STOPPED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA8  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ AC PO NC
2CB0:00F5 C3             RET                                   ;BR7
```

Loading a new copy of COMMSCMD, setting the command tail to *RESUME 1*, and monitoring program execution yields the following:

```
-N COMMSCMD.EXE  <Enter>
-Z _TEXT _TEXT-CS  <Enter>
-Z DGROUP DGROUP-CS  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [0000 _TEXT]
       0158 DGROUP
 0000 BADSCOP
       1313 CSEG
-L  <Enter>
-X*  <Enter>
[2CB0 COMMSCMD]
      [2CB0 _TEXT]
       2E08 DGROUP
 0000 BADSCOP
       1313 CSEG
-N RESUME 1  <Enter>
-G  <Enter>
AX=0022  BX=0F82  CX=0019  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=002E   NV UP EI PL NZ NA PO NC
41:             if (0 == stricmp(argv[1],"STOP"))
2CB0:002E B83600         MOV     AX,0036                       ;BR0
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0086   NV UP EI PL ZR NA PE NC
46:                 cmd = 2;
2CB0:0086 C746FC0200     MOV     Word Ptr [BP-04],0002         ;BR3 SS:0FA0=0000
-G  <Enter>
AX=0000  BX=415A  CX=0000  DX=0098  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=008B   NV UP EI PL ZR NA PE NC
49:         if (argc == 3)
2CB0:008B 837E0403       CMP     Word Ptr [BP+04],+03          ;BR4 SS:0FA8=0003
-G  <Enter>
AX=0001  BX=00D0  CX=0000  DX=0000  SP=0F7E  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00AC   NV UP EI PL NZ NA PO NC
53:                 port = port-1;
2CB0:00AC FF4EFA         DEC     Word Ptr [BP-06]              ;BR5 SS:0F9E=0001
-G  <Enter>
AX=0060  BX=00D0  CX=0000  DX=0000  SP=0F78  BP=0FA4  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00C7   NV UP EI PL ZR NA PE NC
2CB0:00C7 E8EF00         CALL     _int86                        ;BR6
-G  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0190   NV UP DI PL NZ AC PE NC
CSEG:CONTROL:
1313:0190 80FC00         CMP     AH,00                         ;BR8
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0193   NV UP DI PL NZ NA PO NC
1313:0193 7521           JNZ     CONTROL+26 (01B6)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B6   NV UP DI PL NZ NA PO NC
1313:01B6 80FC01         CMP     AH,01
```

(more)

```
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01B9   NV UP DI PL NZ NA PO NC
1313:01B9 751C          JNZ     CONTROL+47 (01D7)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01D7   NV UP DI PL NZ NA PO NC
1313:01D7 80FC02        CMP     AH,02
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01DA   NV UP DI PL ZR NA PE NC
1313:01DA 7516          JNZ     CONTROL+62 (01F2)
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01DC   NV UP DI PL ZR NA PE NC
1313:01DC 2E833E0B0100  CMP     Word Ptr CS:[BUFPNTR],+00         CS:010B=0202
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E2   NV UP DI PL NZ NA PO NC
1313:01E2 741D          JZ     CONTROL+71 (0201)
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E4   NV UP DI PL NZ NA PO NC
1313:01E4 2E88160A01    MOV     CS:[PORT],DL                      CS:010A=00
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01E9   NV UP DI PL NZ NA PO NC
1313:01E9 2EC606090101  MOV     Byte Ptr CS:[STATUS],01           CS:0109=00
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=01EF   NV UP DI PL NZ NA PO NC
1313:01EF EB10          JMP     CONTROL+71 (0201)
-T 5  <Enter>
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F5A  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=1313  IP=0201   NV UP DI PL NZ NA PO NC
1313:0201 CF            IRET
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F60  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2E08  IP=0F6C   NV UP EI PL NZ AC PE NC
2E08:0F6C CB            RETF
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F64  BP=0F66  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=0209   NV UP EI PL NZ AC PE NC
2CB0:0209 5D            POP     BP
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F66  BP=0F74  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=020A   NV UP EI PL NZ AC PE NC
2CB0:020A 57            PUSH     DI
AX=0265  BX=001E  CX=3F48  DX=0000  SP=0F64  BP=0F74  SI=0000  DI=7400
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=020B   NV UP EI PL NZ AC PE NC
2CB0:020B 8B7E08        MOV     DI,[BP+08]                        SS:0F7C=0F90
-G  <Enter>
Communications tracing RESUMED for port COM1:
AX=002F  BX=0001  CX=0C13  DX=0000  SP=0FA6  BP=0000  SI=0089  DI=1065
DS=2E08  ES=2E08  SS=2E08  CS=2CB0  IP=00F5   NV UP EI PL NZ NA PE NC
2CB0:00F5 C3            RET                                 ;BR7
-Q  <Enter>
```

The processing of a resume request is correct. Thus, the problem with stop processing in BADSCOP was the only problem. The corrected BADSCOP, which is actually COMMSCOP, is shown in Figure 18-4.

CodeView

CodeView is the most sophisticated debugging monitor produced by Microsoft. It combines the philosophy and many of the commands of its predecessors, DEBUG and SYMDEB, with true source-code debugging. The availability of source lines and symbols allows CodeView to rival the convenience of program development and debugging previously available only in interpreters such as Microsoft GW-BASIC. However, this high level of interaction with the source program is also the root of its problems for advanced debugging.

In order to provide the debugger with the tools to debug at the source-line level and to interrogate program variables, CodeView is required to have a detailed knowledge of how high-order languages work and of their internal conventions. This is not a problem for languages like C, Pascal, and FORTRAN, versions of which are produced by the same company that created CodeView. The object code generated by these compilers obeys a stringent set of rules and conventions. Assembly-language programs, however, tend to follow their own rules and traditions, making them quite different from C programs, with their own separate debugging needs.

C, Pascal, and FORTRAN programmers will find CodeView a dream to use. Assembly-language programmers using versions of MASM earlier than 5.0 will find CodeView cumbersome and will have to weigh its advantages over its disadvantages. All users will, however, appreciate the good design and programming that have gone into CodeView. It is pleasing to know that someone understands the programmer's debugging needs and is trying to ease the burden.

CodeView has added several welcome functions to the debugger's repertoire, but one of these new features towers above the rest—watchpoints. The debugger can watch the values of program variables or expressions and set breakpoints on them, making it possible to stop execution if an expression evaluates to zero or if a location changes. Previous debugging monitors have been limited to tracing and breaking on instructions. This new facet of debugging changes, somewhat, the approach to resolving a bug.

In the previous discussion of debugging techniques, an orderly application of techniques from inspection and observation through instrumentation to debugging monitors was recommended. This sequence is still recommended with CodeView, but now the instrumentation features have been integrated into the debugging monitor.

A simple example

The following example shows how CodeView uses the instrumentation approach to isolate a problem and then uses the debugging monitor functions to solve it. The example is also an introduction to CodeView commands and techniques. The commands are, for the most part, similar to those used by SYMDEB. Those commands that differ greatly are indicated. This example, like all the examples and demonstrations in this article, is not intended to be a complete tutorial—CodeView commands are summarized elsewhere in this book and explained in detail in the manual accompanying the product. *See* PROGRAMMING UTILITIES: CODEVIEW. The example simply shows some of the more common CodeView commands and demonstrates debugging techniques using them.

UPPERCAS.C (Figure 18-13) is a simple program whose sole function is to convert a canned string to uppercase. When executed, the program prints a few of the characters from the string and some that aren't in the string. Inspecting the listing doesn't reveal the cause of the problem. (Some readers with experience writing C programs will see the cause of the problem, because it is quite common; pretend, for now, that the listing is of no help and enjoy the wonders of CodeView.)

```
/**************************************************************************
 *                                                                        *
 * UPPERCAS.C                                                             *
 *    This routine converts a fixed string to uppercase and prints it.    *
 *                                                                        *
 **************************************************************************/

#include <ctype.h>
#include <string.h>
#include <stdio.h>

main(argc,argv)

int argc;
char *argv[];

{
char    *cp,c;

        cp = "a string\n";

        /*  Convert *cp to uppercase and write to standard output   */

        while (*cp != '\0')
                {
                c = toupper(*cp++);
                putchar(c);
                }

}
```

Figure 18-13. An erroneous C program to convert a string to uppercase.

Like SYMDEB, CodeView requires some special preparation to produce a suitable executable file. CodeView, however, makes the job much simpler. Using the Microsoft C Compiler, compile the program with

```
C>MSC /Zi UPPERCAS;  <Enter>
```

(Remember that C is case sensitive when interpreting switches, so the /Zi switch should be entered exactly as shown.) The /Zi switch instructs the compiler to generate the symbol tables and line-number information needed by CodeView. Other options appropriate to the program can also be included, but /Zi is required.

To form an executable file, use the Microsoft Object Linker (LINK) as follows:

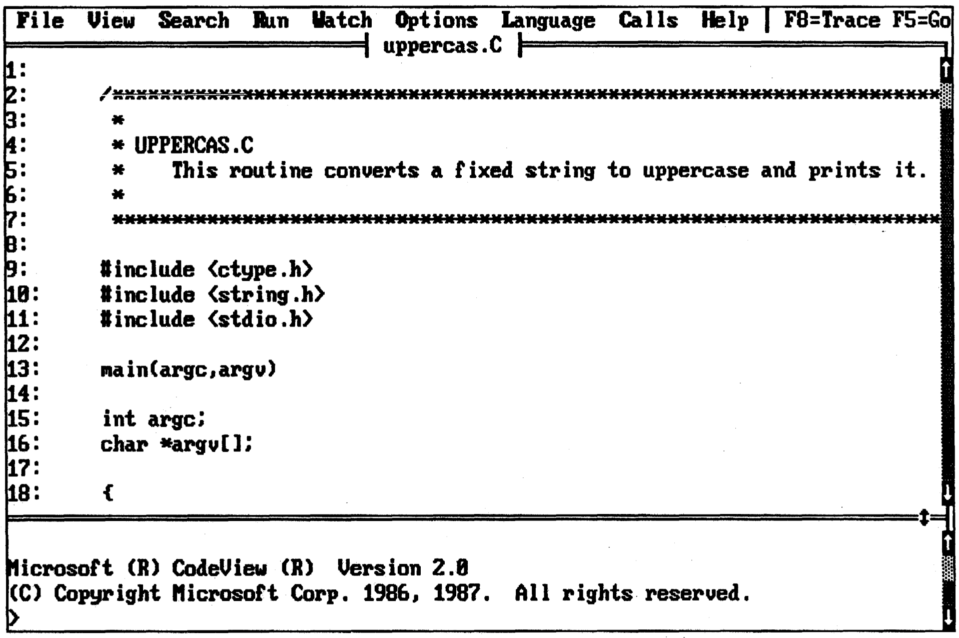
```
C>LINK /CO UPPERCAS;  <Enter>
```

This command line instructs LINK to build an executable file with the information needed for CodeView. Other options can be used as needed or desired. The output of LINK, UPPERCAS.EXE, will be larger than a .EXE file built without /CO (about 2600 bytes larger in this case), but the program will run correctly when executed without CodeView.

Starting CodeView is straightforward. Simply type

```
C>CV UPPERCAS  <Enter>
```

CodeView loads UPPERCAS.EXE. It locates UPPERCAS.C, the source file, and loads that too. It then presents a full-screen display similar to this:


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

```
       /*************************************************************************
        *
        * UPPERCAS.C
        *    This routine converts a fixed string to uppercase and prints it.
        *
        *************************************************************************

       #include <ctype.h>
       #include <string.h>
       #include <stdio.h>

       main(argc,argv)

       int argc;
       char *argv[];

       {
```

```
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

This display has two windows open: the display window, which shows the program being debugged, and the the dialog window, which currently contains only the copyright notice and a prompt (>) for input. The F6 function key moves the cursor back and forth between the two windows.

CodeView can be instructed from either window to go to a specific line (that is, to execute until a specific line is reached). If the cursor is in the display window, use the arrow keys to select a line and press the F7 key. Execution will proceed until the selected line (or the end of the program) is reached. To start execution without specifying a stop line, press F5.

The same functions can be performed from the dialog window using typed commands, which may seem more familiar. Enter the Go Execute Program command, G, optionally followed by an address. Execution will continue until the specified address is reached or until stopped by something else, such as the end of the program. In this sense, the CodeView Go command is the same as that of DEBUG and SYMDEB. Unlike those routines, however, CodeView's Go command does not allow an equals operator (=).

The address for the Go command can be specified in several ways. Because the display window is currently showing only source lines, it is appropriate to set the stop location in terms of line numbers. The syntax of a line-number specification is the same as in SYMDEB — simply enter the line number preceded by a period:

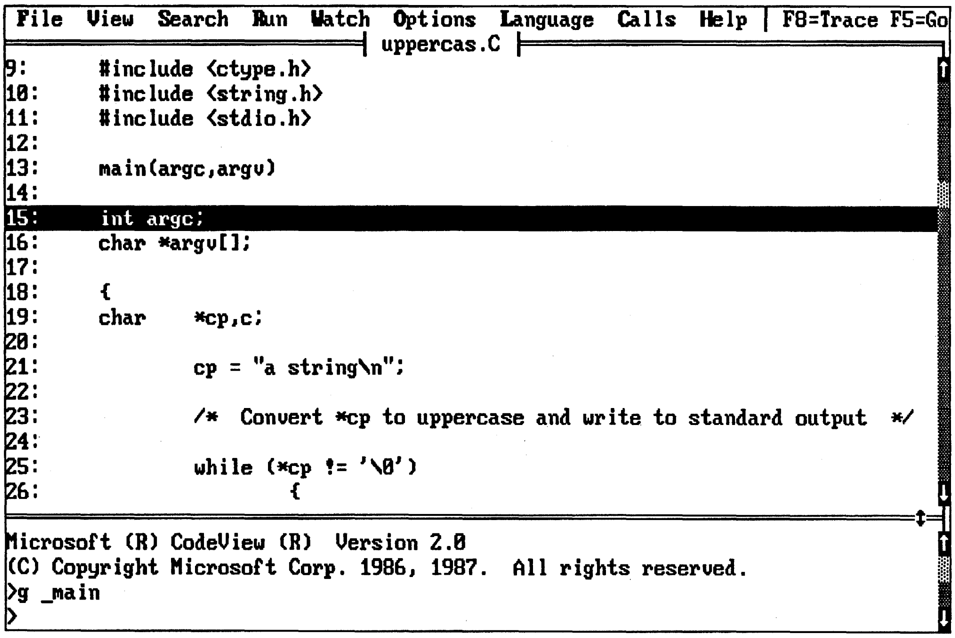
```
>G .27  <Enter>
```

Note that the line number is specified in decimal. This seemingly innocent statement uncovers one of the problem areas in CodeView, especially for assembly-language programmers. The default radix for CodeView is decimal. This convention works well for things associated with the C program, such as line numbers, but is very inconvenient for addresses and other similar items, which are usually in hexadecimal. Hexadecimal numbers must be specified using the cumbersome C notation. Thus, the number FF3EH would be entered as 0xff3e. The radix can be changed using the Change Current Radix command, N (different from the DEBUG and SYMDEB N command). (The problems associated with hexadecimal numbers in early versions of CodeView are no longer present in versions 2.0 and later.)

The radix problem can be avoided, for the moment, by using labels. Issue

```
>G _main  <Enter>
```

to cause CodeView to execute until the main routine is reached. CodeView then shows



```
9:      #include <ctype.h>
10:     #include <string.h>
11:     #include <stdio.h>
12:
13:     main(argc,argv)
14:
15:     int argc;
16:     char *argv[];
17:
18:     {
19:     char    *cp,c;
20:
21:             cp = "a string\n";
22:
23:             /*  Convert *cp to uppercase and write to standard output  */
24:
25:             while (*cp != '\0')
26:                     {
```

```
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>g _main
>
```

The display shows line 15 in reverse video, indicating that CodeView has stopped there. This is the first line of the *main()* module, but it is not executable. Press the F10 key, which has the same effect as entering the Step Through Program command, P, in the dialog window, to cause line 19 to be executed. The reverse video line is then 21, which is the next line to be executed.

To see the changes to *cp*, **cp*, and *c*, establish a watch on these three variables. To use the Watch Word command, WW, for the word *cp*, type

```
>WW cp  <Enter>
```

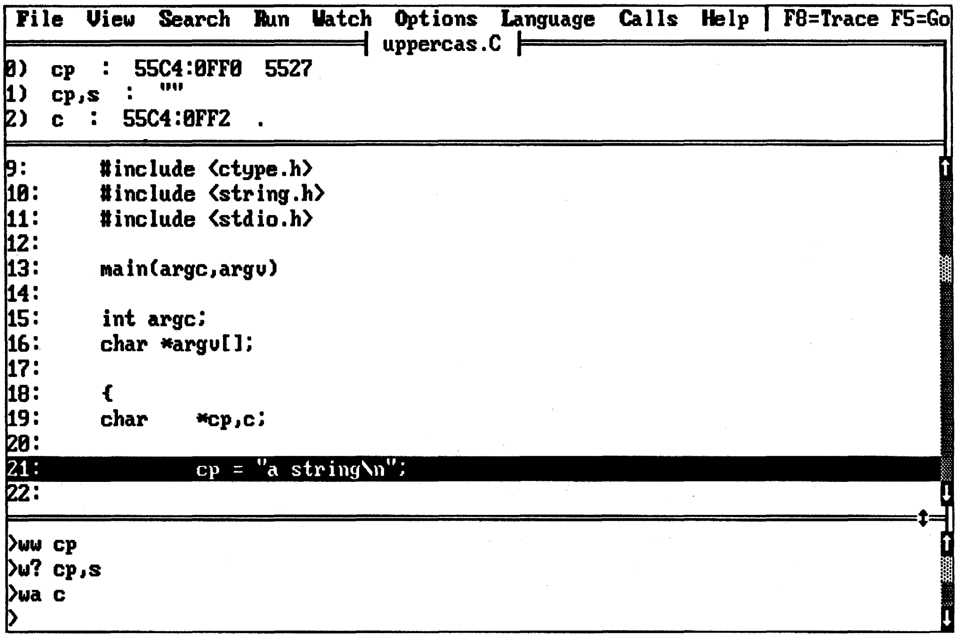
When entered from the dialog window, this command opens the watch window at the top of the screen and displays the current value of *cp*. To display the expression at **cp*, use the Watch Expression command, W?, as follows:

```
>W? cp,s  <Enter>
```

This expression will display the null-delimited string at **cp*. Finally, to see the ASCII character value of *c*, use the Watch ASCII command, WA:

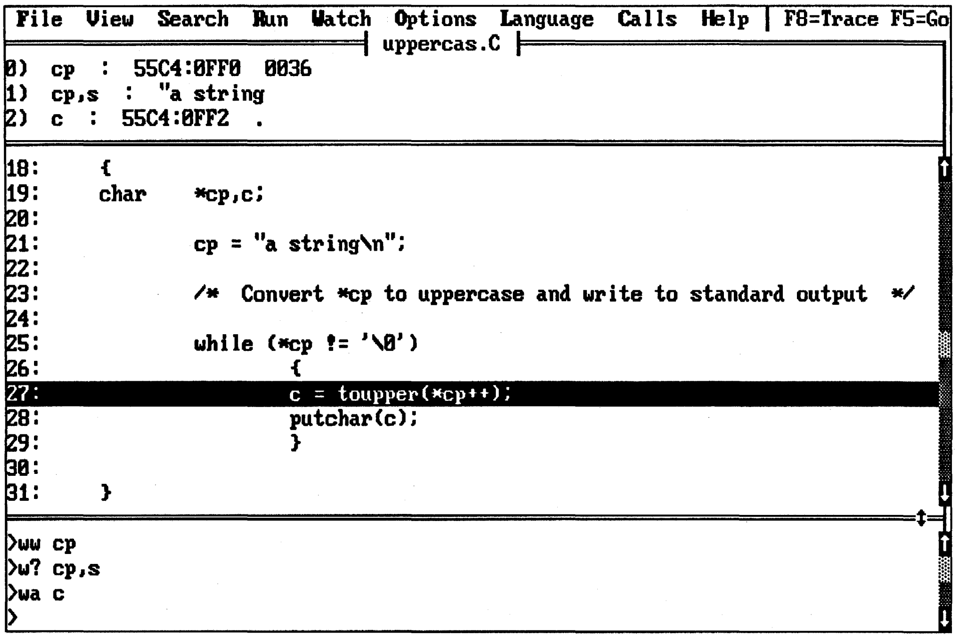
```
>WA c  <Enter>
```

The results of these watch commands are shown in the following screen:


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

```
0) cp  :  55C4:0FF0  5527
1) cp,s  :  ""
2) c  :  55C4:0FF2  .

9:      #include <ctype.h>
10:     #include <string.h>
11:     #include <stdio.h>
12:
13:     main(argc,argv)
14:
15:     int argc;
16:     char *argv[];
17:
18:     {
19:     char    *cp,c;
20:
21:             cp = "a string\n";
22:

>ww cp
>w? cp,s
>wa c
>
```

The values displayed in the watch window are not yet defined because line 21, which initialized *cp*, has not been executed. Press F8 to rectify this. Press it again to bring the execution of the program into the main loop.


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

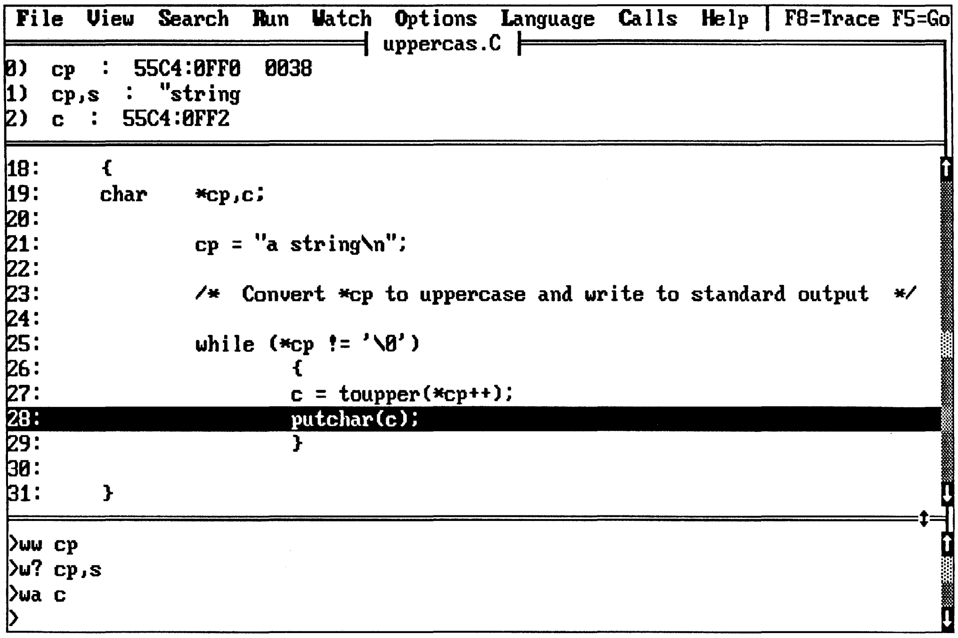
```
0) cp  :  55C4:0FF0  0036
1) cp,s  :  "a string
2) c  :  55C4:0FF2  .
```

```
        {
        char    *cp,c;

                cp = "a string\n";

                /*  Convert *cp to uppercase and write to standard output  */

                while (*cp != '\0')
                        {
                        c = toupper(*cp++);
                        putchar(c);
                        }

        }
```

```
>ww cp
>w? cp,s
>wa c
>
```

The pointer *cp* now contains the correct address. The Display Memory command, D, could be used to display the contents of DS:0036H, just as in DEBUG and SYMDEB. (This step is not necessary, however, because there is a formatted display of memory in the watch window at 1). The variable *c* has not yet been initialized.

Press the F8 key to execute line 27. A curious and unexpected thing happens, as shown in the next screen:


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

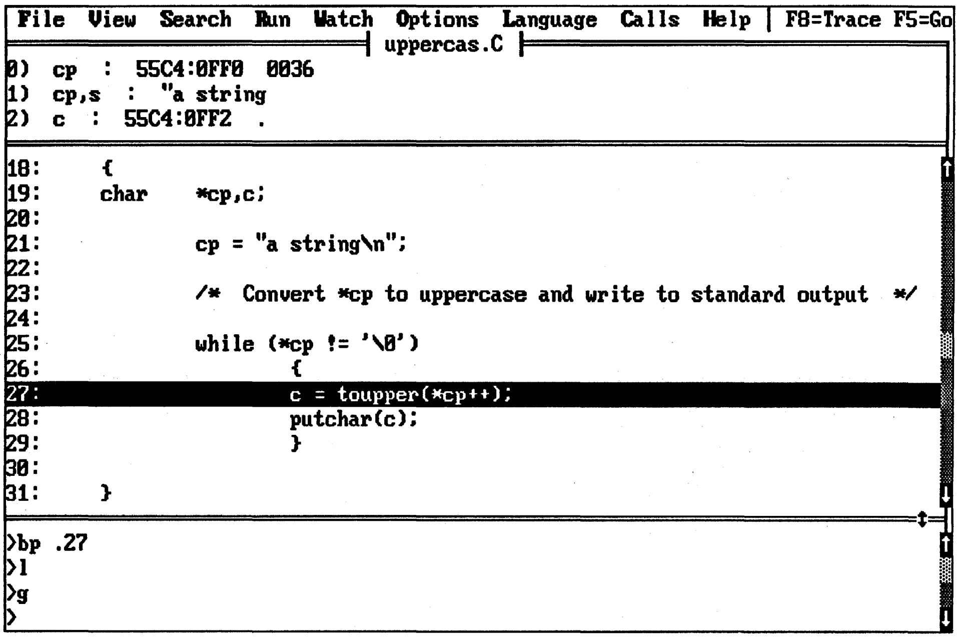
```
0) cp  :  55C4:0FF0  0038
1) cp,s  :  "string
2) c  :  55C4:0FF2

18:     {
19:     char     *cp,c;
20:
21:              cp = "a string\n";
22:
23:              /*  Convert *cp to uppercase and write to standard output  */
24:
25:              while (*cp != '\0')
26:                     {
27:                     c = toupper(*cp++);
28:                     putchar(c);
29:                     }
30:
31:     }

>ww cp
>w? cp,s
>wa c
>
```

Notice that the value of *cp* has changed from 0036H to 0038H. The line of code, however, indicates that the pointer should have been incremented by only one (**cp*++). The second character of the string, a blank, has been loaded into *c*. This could explain the apparent random selection of characters being displayed (actually every other character) and the garbage characters displayed (the zero at the end of the string might be skipped, causing the routine to continue converting until a zero is encountered somewhere in memory).

Source-line debugging does not reveal enough about what is happening in this case. To look more closely at the mechanism of the program, the program must be restarted. Before doing this, set a breakpoint at line 27:

```
>BP .27  <Enter>
```

Then restart (actually, reload) the program with the Reload Program command, L. Note that watch commands and breakpoints are preserved when a program is restarted. Executing the restarted program with G yields

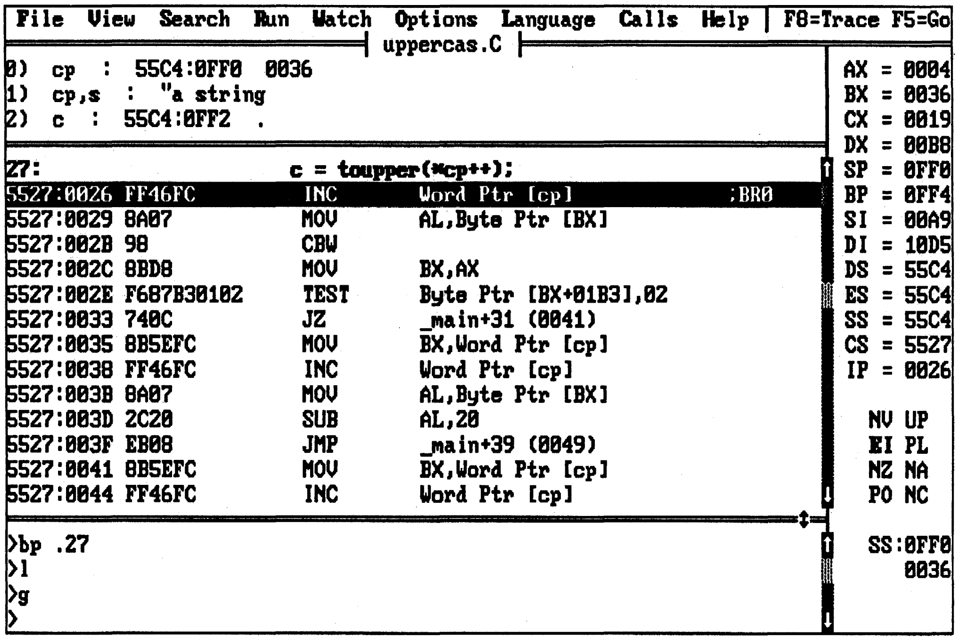

File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

```
0) cp  :  55C4:0FF0  0036
1) cp,s  :  "a string
2) c  :  55C4:0FF2  .
```

```
18:     {
19:     char     *cp,c;
20:
21:              cp = "a string\n";
22:
23:              /*  Convert *cp to uppercase and write to standard output  */
24:
25:              while (*cp != '\0')
26:                      {
27:                      c = toupper(*cp++);
28:                      putchar(c);
29:                      }
30:
31:     }
```

```
>bp .27
>l
>g
>
```

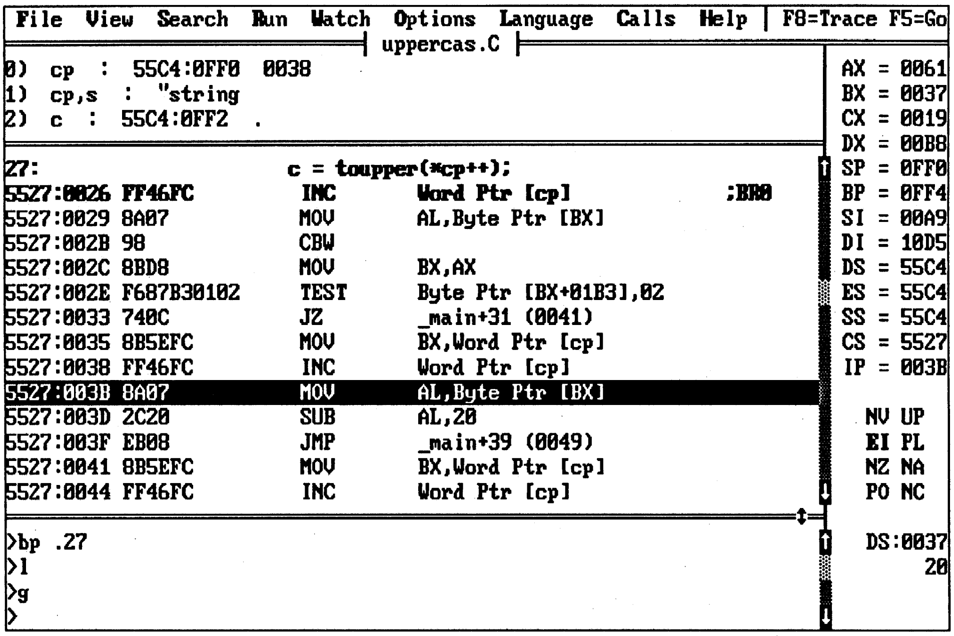
The display shows line 27 in reverse video, indicating that it is the next line to be executed. The pointer *cp* has the correct value, as shown in the watch window. Now Press the F2 key to turn on the register display and press F3 to show the assembly code.


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

```
0) cp  :  55C4:0FF0  0036                                         AX = 0004
1) cp,s  :  "a string                                             BX = 0036
2) c  :  55C4:0FF2  .                                             CX = 0019
                                                                  DX = 00B8
27:                     c = toupper(*cp++);                       SP = 0FF0
5527:0026 FF46FC        INC      Word Ptr [cp]          ;BR0      BP = 0FF4
5527:0029 8A07          MOV      AL,Byte Ptr [BX]                 SI = 00A9
5527:002B 98            CBW                                       DI = 10D5
5527:002C 8BD8          MOV      BX,AX                            DS = 55C4
5527:002E F687B30102    TEST     Byte Ptr [BX+01B3],02            ES = 55C4
5527:0033 740C          JZ       _main+31 (0041)                  SS = 55C4
5527:0035 8B5EFC        MOV      BX,Word Ptr [cp]                 CS = 5527
5527:0038 FF46FC        INC      Word Ptr [cp]                    IP = 0026
5527:003B 8A07          MOV      AL,Byte Ptr [BX]
5527:003D 2C20          SUB      AL,20                              NV UP
5527:003F EB08          JMP      _main+39 (0049)                    EI PL
5527:0041 8B5EFC        MOV      BX,Word Ptr [cp]                   NZ NA
5527:0044 FF46FC        INC      Word Ptr [cp]                      PO NC

>bp .27                                                             SS:0FF0
>l                                                                     0036
>g
>
```

The display highlights line 27, indicating that a breakpoint exists at this line. The line of code at CS:0026H is in reverse video, indicating that it is the next line to be executed.

The previous instruction has loaded BX with *[cp].* The first thing the code for line 27 does is increment the word at memory location *[cp].* The initial value of *cp* is in BX, so the **cp++* request can now be executed. Use the F8 key to single-step through the lines of code. Notice that when only source lines are on the screen, F8 steps one source line at a time, but when assembly code is shown, F8 steps one assembly line at a time. Single-stepping through the code, note how the registers and watch window change. Everything appears normal until CS:0038H is executed.



File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
uppercas.C

```
0)  cp   :  55C4:0FF0  0038
1)  cp,s  :  "string
2)  c  :  55C4:0FF2  .

27:                          c = toupper(*cp++);
5527:0026 FF46FC             INC        Word Ptr [cp]                  ;BR0
5527:0029 8A07               MOV        AL,Byte Ptr [BX]
5527:002B 98                 CBW
5527:002C 8BD8               MOV        BX,AX
5527:002E F687B30102         TEST       Byte Ptr [BX+01B3],02
5527:0033 740C               JZ         _main+31 (0041)
5527:0035 8B5EFC             MOV        BX,Word Ptr [cp]
5527:0038 FF46FC             INC        Word Ptr [cp]
5527:003B 8A07               MOV        AL,Byte Ptr [BX]
5527:003D 2C20               SUB        AL,20
5527:003F EB08               JMP        _main+39 (0049)
5527:0041 8B5EFC             MOV        BX,Word Ptr [cp]
5527:0044 FF46FC             INC        Word Ptr [cp]

>bp .27
>l
>g
>
```

```
AX = 0061
BX = 0037
CX = 0019
DX = 00B8
SP = 0FF0
BP = 0FF4
SI = 00A9
DI = 10D5
DS = 55C4
ES = 55C4
SS = 55C4
CS = 5527
IP = 003B

  NV UP
  EI PL
  NZ NA
  PO NC

  DS:0037
       20
```

Notice that the value of *cp* in the watch window has incremented again. The line of C code has two increments hidden in it, not the expected single increment. Why is this?

To find the answer, examine the *toupper()* macro. The following definition, extracted from CTYPE.H, explains what is happening:

```
#define _UPPER          0x1         /* uppercase letter */
#define _LOWER          0x2         /* lowercase letter */
#define isupper(c)       ( (_ctype+1)[c] & _UPPER )
#define islower(c)       ( (_ctype+1)[c] & _LOWER )

#define _tolower(c)      ( (c)-'A'+'a' )
#define _toupper(c)      ( (c)-'a'+'A' )

#define toupper(c)       ( (islower(c)) ? _toupper(c) : (c) )
#define tolower(c)       ( (isupper(c)) ? _tolower(c) : (c) )
```

The argument to *toupper()*, *c*, is used twice, once in the macro that checks for lowercase, *islower()*, and once in *_toupper()*. The argument is replaced in this case with **cp++*, which has the famous C unexpected side effects. Because the unary post-increment is the handiest way to perform the function desired in the program, fixing the problem by changing the code in the main loop is undesirable. Another solution to the problem is to use the function version of *toupper()*. Because *toupper()* is defined as a function in STDIO.H, simply deleting *#include <ctype.h>* would solve the problem. Unfortunately, this would also deprive the program of the other useful definitions in CTYPE.H. (Admittedly, the features are not currently used by the program, but little programs sometimes grow into mighty systems.) So to keep CTYPE.H but still remove the macro definition of *toupper()*, use the #undef command. (Because *tolower()* has the same problem, it should also be undefined.) The corrected listing is shown in Figure 18-14.

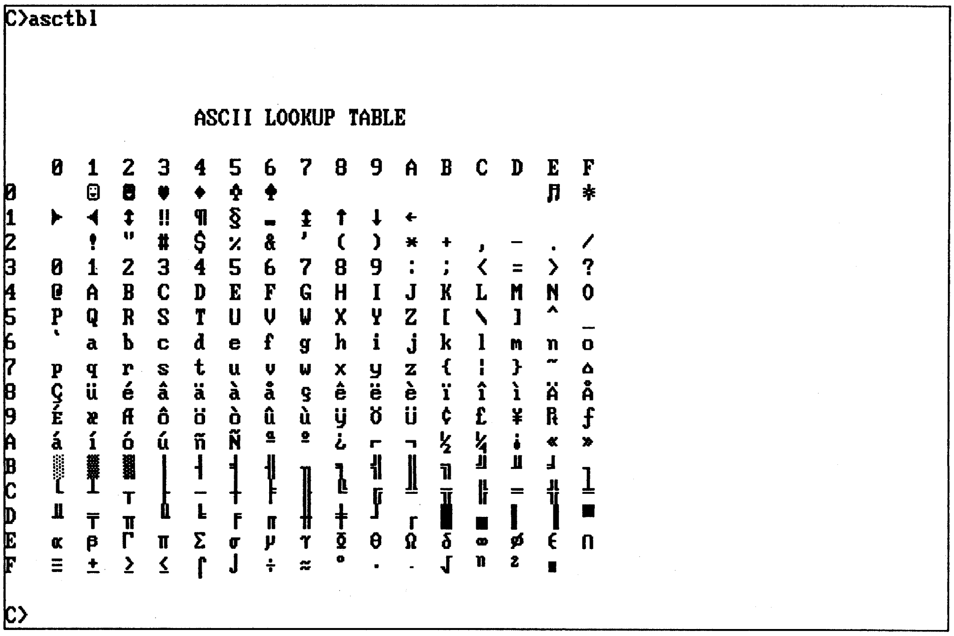
```
/**************************************************************************
 *                                                                        *
 * UPPERCAS.C                                                             *
 *    This routine converts a fixed string to uppercase and prints it.    *
 *                                                                        *
 *************************************************************************/

#include <ctype.h>
#undef toupper
#undef tolower
#include <string.h>
#include <stdio.h>

main(argc,argv)

int argc;
char *argv[];

{
char    *cp,c;

        cp = "a string\n";

        /*  Convert *cp to uppercase and write to standard output  */

        while (*cp != '\0')
                {
                c = toupper(*cp++);
                putchar(c);
                }

}
```

Figure 18-14. The corrected version of UPPERCAS.C.

An example using screen output

A problem with DEBUG is that it writes to the same screen as the program does. Both SYMDEB and CodeView, however, allow the debugger to switch back and forth between the screen containing the program's output and the screen containing the debugger's output. This feature is a special option with SYMDEB and is sometimes clumsy to use, but with CodeView, keeping a separate program output screen is automatic and switching back and forth involves simply pressing a function key (F4).

The following example program is intended to display an ASCII lookup table with all the displayable characters available on an IBM PC. The expected output is shown in Figure 18-15.



```
C>asctbl



                  ASCII LOOKUP TABLE

     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0       ☺  ☻  ♥  ♦  ♣  ♠                       ♫  ☼
1    ►  ◄  ↕  ‼  ¶  §  ▬  ↨  ↑  ↓  ←
2       !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
3    0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
4    @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
5    P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
6    `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
7    p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  ⌂
8    Ç  ü  é  â  ä  à  å  ç  ê  ë  è  ï  î  ì  Ä  Å
9    É  æ  Æ  ô  ö  ò  û  ù  ÿ  Ö  Ü  ¢  £  ¥  ₧  ƒ
A    á  í  ó  ú  ñ  Ñ  ª  º  ¿  ⌐  ¬  ½  ¼  ¡  «  »
B    ░  ▒  ▓  │  ┤  ╡  ╢  ╖  ╕  ╣  ║  ╗  ╝  ╜  ╛  ┐
C    └  ┴  ┬  ├  ─  ┼  ╞  ╟  ╚  ╔  ╩  ╦  ╠  ═  ╬  ╧
D    ╨  ╤  ╥  ╙  ╘  ╒  ╓  ╫  ╪  ┘  ┌  █  ▄  ▌  ▐  ▀
E    α  ß  Γ  π  Σ  σ  µ  τ  Φ  Θ  Ω  δ  ∞  φ  ε  ∩
F    ≡  ±  ≥  ≤  ⌠  ⌡  ÷  ≈  °  ∙  ·  √  ⁿ  ²  ■

C>
```

Figure 18-15. The output expected from ASCTBL.C.

The program that should produce this display, ASCTBL.C, is shown in Figure 18-16.

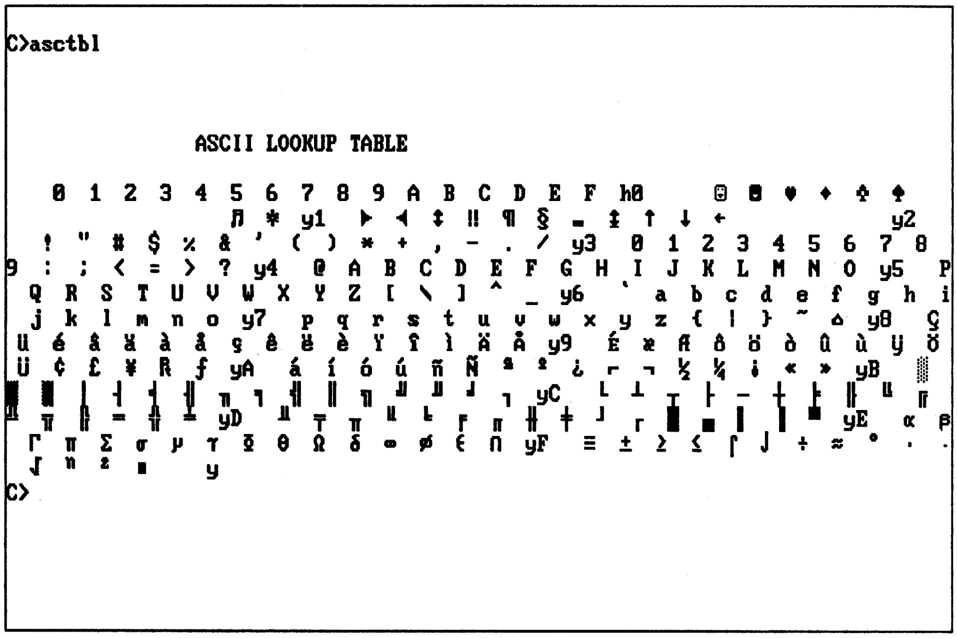
```
/**********************************************************************
 *                                                                    *
 *  ASCTBL.C                                                          *
 *   This program generates an ASCII lookup table for all displayable  *
 *   ASCII and extended IBM PC codes, leaving blanks for nondisplayable *
 *   codes.                                                           *
 *                                                                    *
 **********************************************************************/

#include <ctype.h>
#include <stdio.h>

main()
{
int i, j, k;
        /* Print table title. */
        printf("\n\n\n                ASCII LOOKUP TABLE\n\n");

        /* Print column headers. */
        printf("    ");
        for (i = 0; i < 16; i++)
                printf("%X  ", i);
        fputchar("\n");
```

Figure 18-16. An erroneous program to display ASCII characters. *(more)*

```
        /* Print each line of the table. */
        for (i = 0, k = 0; i < 16; i++)
                {
                /* Print first hex digit of symbols on this line. */
                printf("%X   ", i);
                /* Print each of the 16 symbols for this line. */
                for (j = 0; j < 16; j++)
                        {
                        /* Filter nonprintable characters. */
                        if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31))
                                printf("   ");
                        else
                                printf("%c  ", k);
                        k++;
                        }
                fputchar("\n");
                }
}
```

Figure 18-16. Continued.

The problem to be debugged in this example is evident when the program in Figure 18-16 is compiled, linked, and executed. Here is the resulting display:

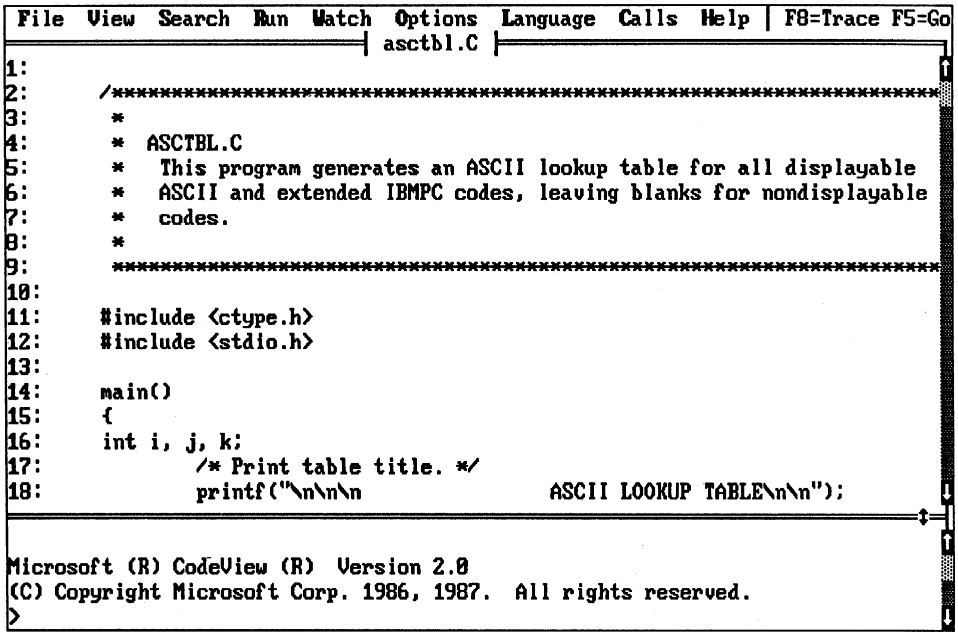


```
C>asctbl


                ASCII LOOKUP TABLE

   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  h0     ☺  ☻  ♥  ♦  ♣  ♠
                  ♫  ☼  y1  ►  ◄  ↕  ‼  ¶  §  ▬  ↨  ↑  ↓  ←              y2
  !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  y3  0  1  2  3  4  5  6  7  8
9  :  ;  <  =  >  ?  y4  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  y5  P
 Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  y6  `  a  b  c  d  e  f  g  h  i
 j  k  l  m  n  o  y7  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~  ⌂  y8  Ç
ü  é  â  ä  à  å  ç  ê  ë  è  ï  î  ì  Ä  Å  y9  É  æ  Æ  ô  ö  ò  û  ù  ÿ  Ö
Ü  ¢  £  ¥  ₧  ƒ  yA  á  í  ó  ú  ñ  Ñ  ª  º  ¿  ⌐  ¬  ½  ¼  ¡  «  »  yB  ░
▒  ▓  │  ┤  ╡  ╢  ╖  ╕  ╣  ║  ╗  ╝  ╜  ╛  ┐  yC  └  ┴  ┬  ├  ─  ┼  ╞  ╟  ╚  ╔
╩  ╦  ╠  ═  ╬  ╧  yD  ╨  ╤  ╥  ╙  ╘  ╒  ╓  ╫  ╪  ┘  ┌  █  ▄  ▌  ▐  ▀  yE  α  ß
 Γ  π  Σ  σ  µ  τ  Φ  Θ  Ω  δ  ∞  φ  ε  ∩  yF  ≡  ±  ≥  ≤  ⌠  ⌡  ÷  ≈  °  ∙  ·
 √  ⁿ  ²  ■     y
C>
```

Something is clearly wrong. The output is jumbled and no pattern is immediately obvious. To locate the problem, first prepare a .EXE file and start CodeView as follows:

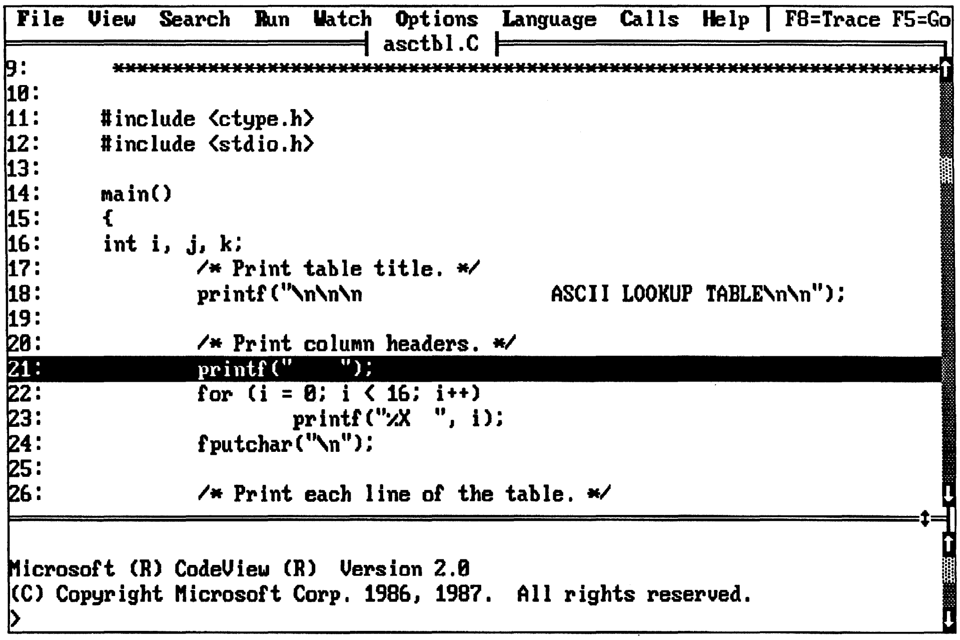
```
C>MSC /Zi ASCTBL;  <Enter>
C>LINK /CO ASCTBL;  <Enter>
C>CV ASCTBL  <Enter>
```

CodeView starts and displays the following screen:



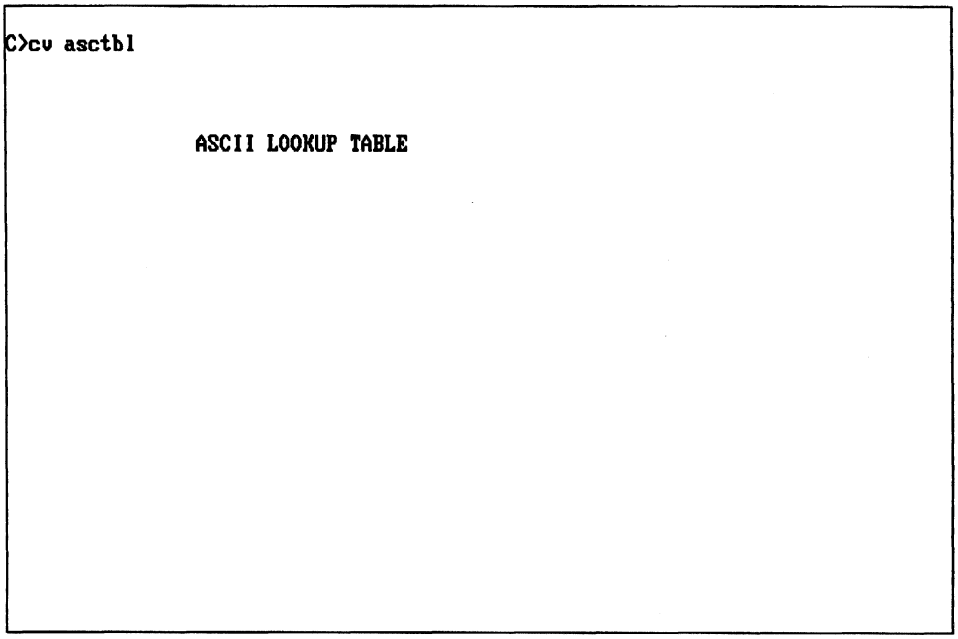
```
 File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                                  asctbl.C
1:
2:      /*********************************************************************
3:       *
4:       *  ASCTBL.C
5:       *   This program generates an ASCII lookup table for all displayable
6:       *   ASCII and extended IBMPC codes, leaving blanks for nondisplayable
7:       *   codes.
8:       *
9:       *********************************************************************
10:
11:     #include <ctype.h>
12:     #include <stdio.h>
13:
14:     main()
15:     {
16:     int i, j, k;
17:             /* Print table title. */
18:             printf("\n\n\n                ASCII LOOKUP TABLE\n\n");

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

The start of the source program is shown in the display window and the dialog window contains an input prompt. Press the F10 key three times to bring execution to line 21. (Remember that the line indicated in reverse video has not yet been executed.)


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
asctbl.C

```
9:        **********************************************************************
10:
11:     #include <ctype.h>
12:     #include <stdio.h>
13:
14:     main()
15:     {
16:     int i, j, k;
17:             /* Print table title. */
18:             printf("\n\n\n                ASCII LOOKUP TABLE\n\n");
19:
20:             /* Print column headers. */
21:             printf("    ");
22:             for (i = 0; i < 16; i++)
23:                     printf("%X  ", i);
24:             fputchar("\n");
25:
26:             /* Print each line of the table. */

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

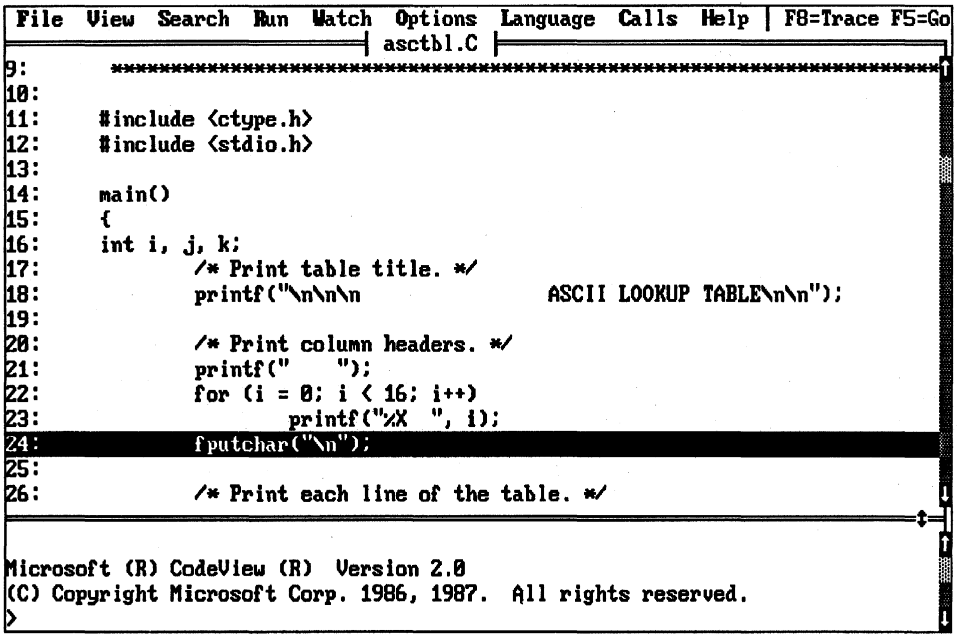
The display heading has been printed at line 18. Press the F4 key to display what the program has written on the screen.



```
C>cv asctbl



                ASCII LOOKUP TABLE
```

Note: Any information on the screen when you started CodeView will remain on the virtual output screen until program execution clears it or forces it to scroll off.

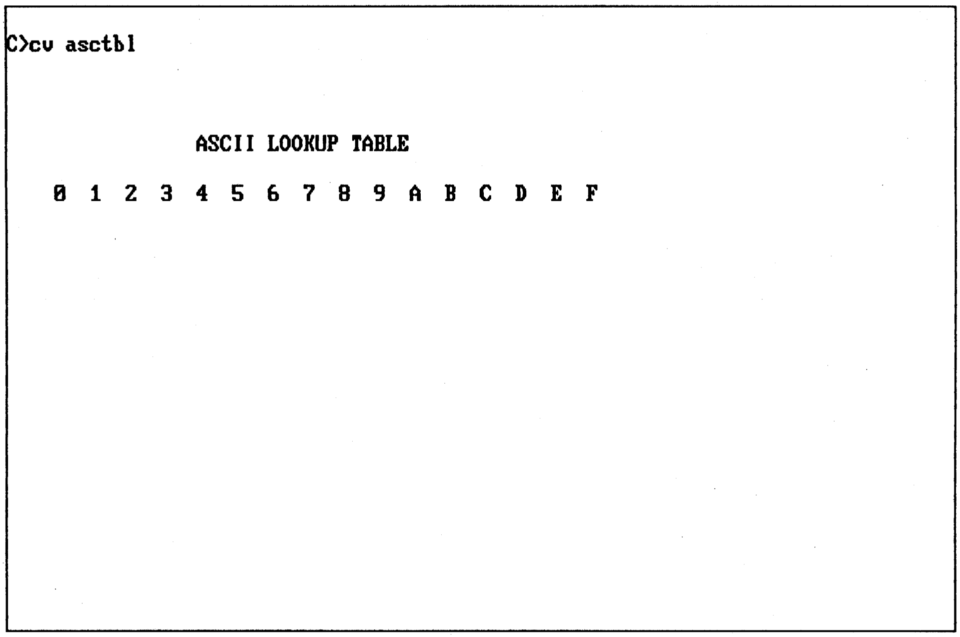
The table heading has been properly written to the screen. Press the F4 key again to return to the CodeView display. Continue executing the program with the F10 key to bring the program to line 24.


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
asctbl.C

```
9:        ***********************************************************************
10:
11:     #include <ctype.h>
12:     #include <stdio.h>
13:
14:     main()
15:     {
16:     int i, j, k;
17:             /* Print table title. */
18:             printf("\n\n\n                ASCII LOOKUP TABLE\n\n");
19:
20:             /* Print column headers. */
21:             printf("    ");
22:             for (i = 0; i < 16; i++)
23:                     printf("%X  ", i);
24:             fputchar("\n");
25:
26:             /* Print each line of the table. */
```

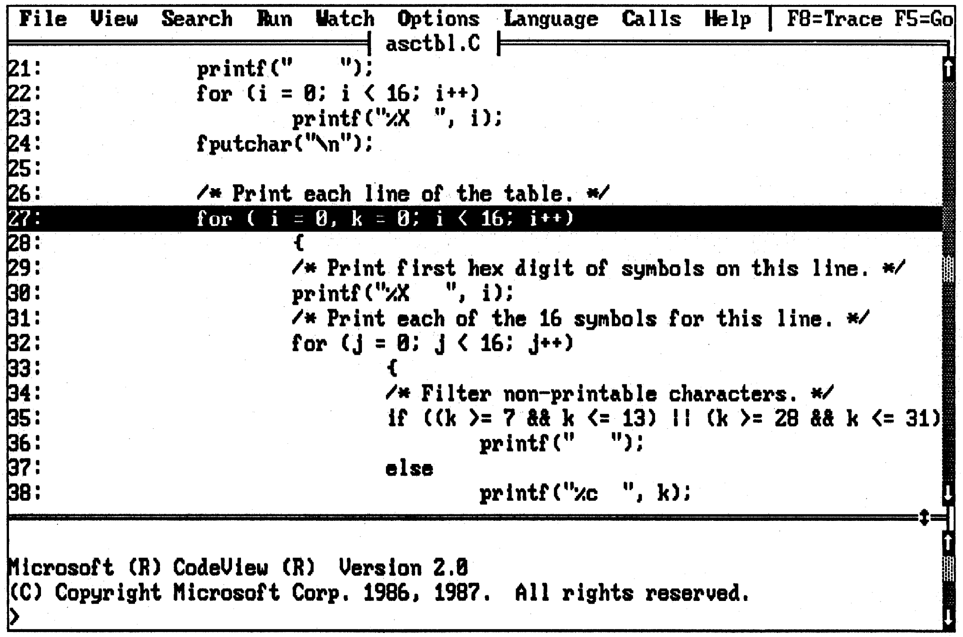
```
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

At this point in program execution, the column headings have been written on the screen. Press the F4 key again to see the results.



```
C>cv asctbl


               ASCII LOOKUP TABLE

    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

The output of the program is still correct, so allow execution to continue by pressing F4 to return to the CodeView screen and then pressing the F10 key. This will execute the call to the *fputchar()* function to write a newline character.

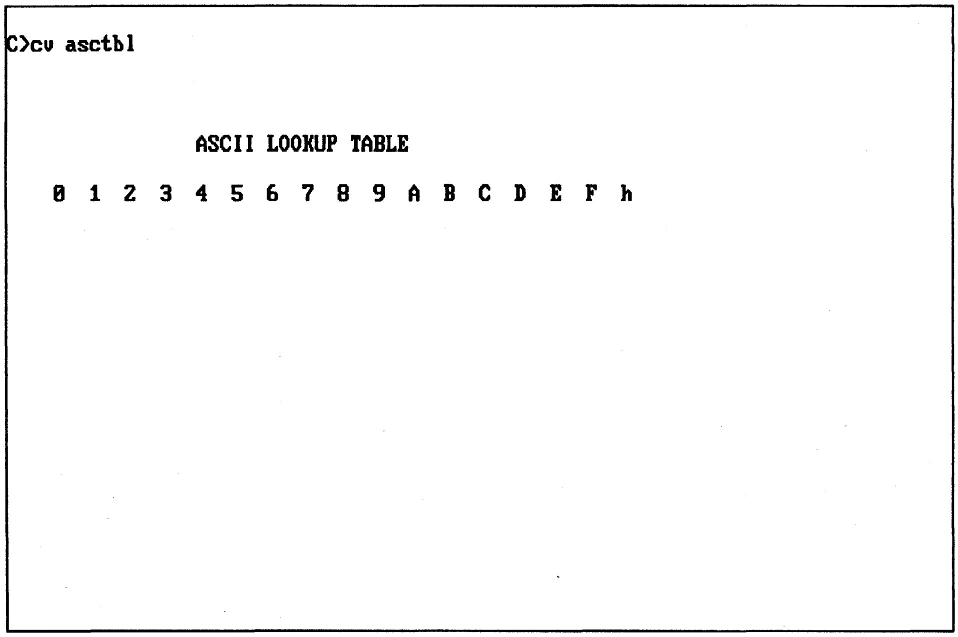


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
asctbl.C

```
        printf("    ");
        for (i = 0; i < 16; i++)
                printf("%X  ", i);
        fputchar("\n");

        /* Print each line of the table. */
        for ( i = 0, k = 0; i < 16; i++)
                {
                /* Print first hex digit of symbols on this line. */
                printf("%X   ", i);
                /* Print each of the 16 symbols for this line. */
                for (j = 0; j < 16; j++)
                        {
                        /* Filter non-printable characters. */
                        if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31)
                                printf("   ");
                        else
                                printf("%c  ", k);
```

```
Microsoft (R) CodeView (R) Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987. All rights reserved.
>
```

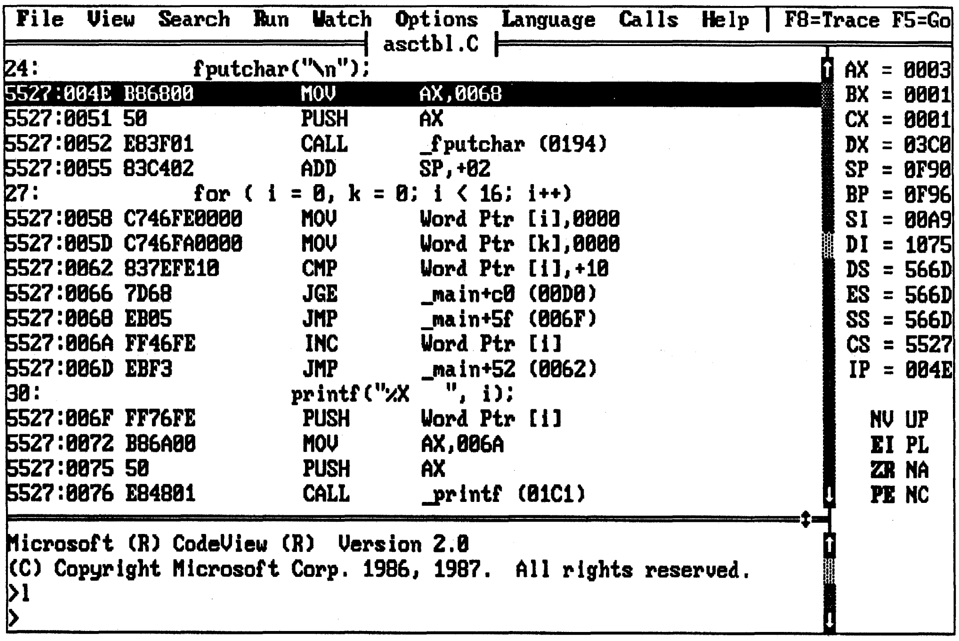
Examination of the output screen shows that the display is now incorrect.



```
C>cv asctbl


                ASCII LOOKUP TABLE

   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  h
```

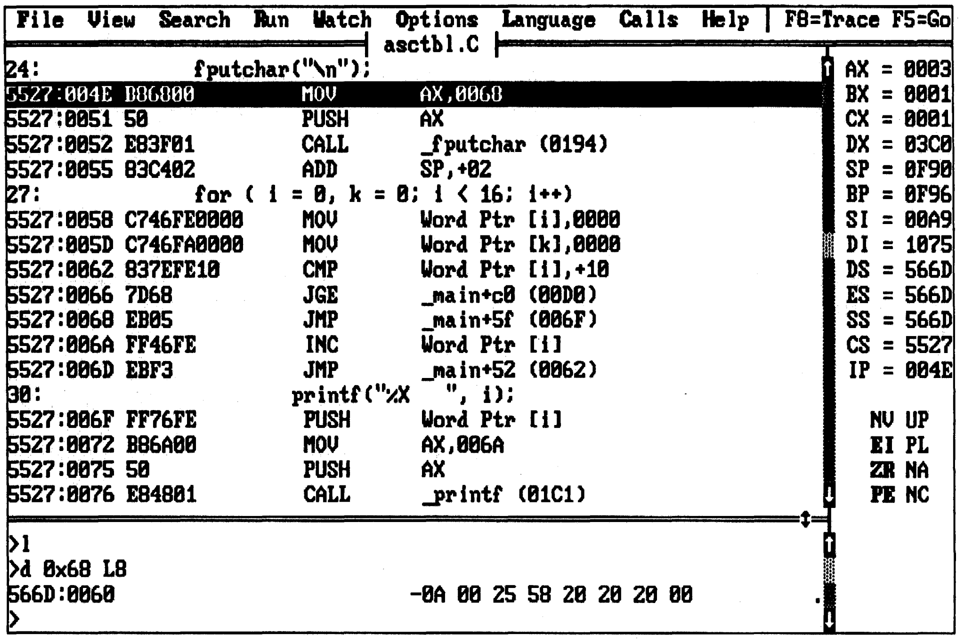
A lowercase *h* has been written to the screen instead of a newline character. Further execution demonstrates that newline characters written with *fputchar()* are not working. A closer inspection of the *fputchar()* function is needed.

To see what is happening, use the Reload Program command to restart execution at the top of the program. Change the cursor window with the F6 key, use the arrow keys to place the cursor on line 24, and press F7. This brings execution back to line 24, where *fputchar()* is called. Press the F3 key to place the display in assembly mode and the F2 key to show the CPU registers and flags. The first assembly instruction of the *fputchar()* function call is about to be executed.


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
asctbl.C

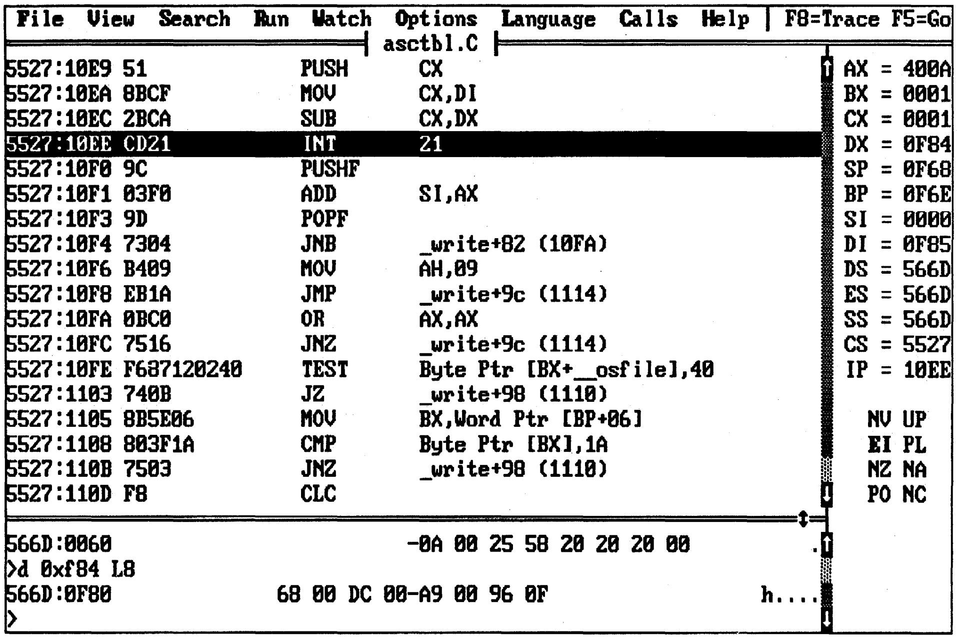
```
24:              fputchar("\n");                               AX = 0003
5527:004E B86800        MOV     AX,0068                        BX = 0001
5527:0051 50            PUSH    AX                             CX = 0001
5527:0052 E83F01        CALL    _fputchar (0194)               DX = 03C0
5527:0055 83C402        ADD     SP,+02                         SP = 0F90
27:              for ( i = 0, k = 0; i < 16; i++)              BP = 0F96
5527:0058 C746FE0000    MOV     Word Ptr [i],0000              SI = 00A9
5527:005D C746FA0000    MOV     Word Ptr [k],0000              DI = 1075
5527:0062 837EFE10      CMP     Word Ptr [i],+10               DS = 566D
5527:0066 7D68          JGE     _main+c0 (00D0)                ES = 566D
5527:0068 EB05          JMP     _main+5f (006F)                SS = 566D
5527:006A FF46FE        INC     Word Ptr [i]                   CS = 5527
5527:006D EBF3          JMP     _main+52 (0062)                IP = 004E
30:                     printf("%X   ", i);
5527:006F FF76FE        PUSH    Word Ptr [i]                     NV UP
5527:0072 B86A00        MOV     AX,006A                          EI PL
5527:0075 50            PUSH    AX                               ZR NA
5527:0076 E84801        CALL    _printf (01C1)                   PE NC

Microsoft (R) CodeView (R) Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>l
>
```

Notice that the parameter being passed to the function by means of the stack is 0068H. Use the Display Memory command to display DS:0068H. (Note the hexadecimal notation.)


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
asctbl.C

```
24:              fputchar("\n");                               AX = 0003
5527:004E B86800        MOV     AX,0068                        BX = 0001
5527:0051 50            PUSH    AX                             CX = 0001
5527:0052 E83F01        CALL    _fputchar (0194)               DX = 03C0
5527:0055 83C402        ADD     SP,+02                         SP = 0F90
27:              for ( i = 0, k = 0; i < 16; i++)              BP = 0F96
5527:0058 C746FE0000    MOV     Word Ptr [i],0000              SI = 00A9
5527:005D C746FA0000    MOV     Word Ptr [k],0000              DI = 1075
5527:0062 837EFE10      CMP     Word Ptr [i],+10               DS = 566D
5527:0066 7D68          JGE     _main+c0 (00D0)                ES = 566D
5527:0068 EB05          JMP     _main+5f (006F)                SS = 566D
5527:006A FF46FE        INC     Word Ptr [i]                   CS = 5527
5527:006D EBF3          JMP     _main+52 (0062)                IP = 004E
30:                     printf("%X   ", i);
5527:006F FF76FE        PUSH    Word Ptr [i]                     NV UP
5527:0072 B86A00        MOV     AX,006A                          EI PL
5527:0075 50            PUSH    AX                               ZR NA
5527:0076 E84801        CALL    _printf (01C1)                   PE NC

>l
>d 0x68 L8
566D:0060                    -0A 00 25 58 20 20 20 00         .
>
```

The contents of memory at this address consist of a null-delimited string containing a newline character. The representation of \n is correct. To see how the string is handled, use the trace key, F8, to single-step through *fputchar()* and subordinate functions. These functions are complicated; nearly 100 steps are required to reach the MS-DOS Interrupt 21H call that actually writes the screen.



```
                                  asctbl.C
5527:10E9 51              PUSH    CX                                AX = 400A
5527:10EA 8BCF            MOV     CX,DI                             BX = 0001
5527:10EC 2BCA            SUB     CX,DX                             CX = 0001
5527:10EE CD21            INT     21                                DX = 0F84
5527:10F0 9C              PUSHF                                     SP = 0F68
5527:10F1 03F0            ADD     SI,AX                             BP = 0F6E
5527:10F3 9D              POPF                                      SI = 0000
5527:10F4 7304            JNB     _write+82 (10FA)                  DI = 0F85
5527:10F6 B409            MOV     AH,09                             DS = 566D
5527:10F8 EB1A            JMP     _write+9c (1114)                  ES = 566D
5527:10FA 0BC0            OR      AX,AX                             SS = 566D
5527:10FC 7516            JNZ     _write+9c (1114)                  CS = 5527
5527:10FE F687120240      TEST    Byte Ptr [BX+__osfile],40         IP = 10EE
5527:1103 740B            JZ      _write+98 (1110)
5527:1105 8B5E06          MOV     BX,Word Ptr [BP+06]                 NV UP
5527:1108 803F1A          CMP     Byte Ptr [BX],1A                    EI PL
5527:110B 7503            JNZ     _write+98 (1110)                    NZ NA
5527:110D F8              CLC                                         PO NC

566D:0060                      -0A 00 25 58 20 20 20 00          .
>d 0xf84 L8
566D:0F80             68 00 DC 00-A9 00 96 0F                h....
>
```

The AH register's contents, 40H, indicate that the Interrupt 21H call is a request for a write to a device. The BX register has the handle of the device, 1, which is the special file handle for standard output (*stdout*). For this program as it was invoked, standard output is the screen. The CX register indicates that 1 byte is to be written; DS:DX points to the data to be written. The contents of memory at DS:0F84H finally reveal the cause of the problem: This memory location contains the *address* of the data to be written, not the data. The *fputchar()* function was called with the wrong level of indirection.

Examination of the listing shows that all the newline requests were made with

```
fputchar("\n");
```

Strings specified with double quotes are replaced in C functions with the address of the string, but the function expected the actual character and not its address. The problem can be corrected by replacing the *fputchar()* calls with

```
fputchar('\n');
```

The newline character will now be passed directly to the function.

This kind of problem can be avoided. C provides the ability to check the type of each parameter passed to a function against the expected type. If the following definition is included at the top of the C program, incorrect types will generate error messages:

```
#define LINT_ARGS
```

The corrected listing is shown in Figure 18-17. This new program produces the correct output.

```
/*************************************************************************
 *                                                                       *
 *  ASCTBL.C                                                             *
 *   This program generates an ASCII lookup table for all displayable    *
 *   ASCII and extended IBM PC codes, leaving blanks for nondisplayable  *
 *   codes.                                                              *
 *                                                                       *
 *************************************************************************/

#define LINT_ARGS
#include <ctype.h>
#include <stdio.h>

main()
{
int i, j, k;
        /* Print table title. */
        printf("\n\n\n                ASCII LOOKUP TABLE\n\n");

        /* Print column headers. */
        printf("    ");
        for (i = 0; i < 16; i++)
                printf("%X  ", i);
        fputchar('\n');

        /* Print each line of the table. */
        for (i = 0, k = 0; i < 16; i++)
               {
               /* Print first hex digit of symbols on this line. */
               printf("%X   ", i);
               /* Print each of the 16 symbols for this line. */
               for (j = 0; j < 16; j++)
                       {
                       /* Filter nonprintable characters. */
                       if ((k >= 7 && k <= 13) || (k >= 28 && k <= 31))
                               printf("   ");
                       else
                               printf("%c  ", k);
                       k++;
                       }
               fputchar('\n');
               }
}
```

Figure 18-17. The correct ASCII table generation program.

CodeView is a good choice for debugging C, Pascal, BASIC, and FORTRAN programs. The fact that versions of MASM earlier than 5.0 do not generate data for CodeView makes CodeView a poorer choice for these assembly-language programs. These disadvantages must be weighed against the ability to set watchpoints and to trap nonmaskable interrupts (NMIs). CodeView is also not as well suited as SYMDEB for debugging programs that interact with TSRs and device drivers, because CodeView does not provide any mechanism for including symbol tables for routines not linked together.

Hardware debugging aids

Hardware debuggers are a combination of hardware and software designed to be installed in a PC system. The software provides features much like those available with SYMDEB and CodeView. The advantages of hardware debuggers over purely software debuggers can be summarized in three points:

- Crash protection
- Manual execution break
- Hardware breakpoints

A hardware debugger can provide program crash protection because of its independence from the PC software. If the program being debugged goes wild and destroys the operating system of the PC, the hardware debugger is protected by virtue of being a separate hardware system and is capable of recovering enough control to allow the user to find out what happened.

All hardware debuggers offer a means of breaking into the program under test from some external source — usually a push button in the hands of the programmer. The mechanism used to get the attention of the PC's CPU is the nonmaskable interrupt (NMI). This interrupt provides a more reliable means of interrupting program execution than the Break key because its operation is independent of the state of interrupts and other conditions.

Hardware debuggers usually have access to the address and data lines on the PC bus, allowing them to set *hardware* breakpoints. Thus, these debuggers can be set to break when specific addresses are referenced. They execute the breakpoint code from a debugging monitor, which generally runs from their own memory. This memory is usually protected from the regular operating system and the application program.

Although hardware debuggers can be used to instrument a program, they should not be confused with the external hardware instrumentation discussed earlier in this article. The logic analyzers and in-circuit emulators mentioned there are general-purpose test instruments; the hardware debuggers are highly specific devices intended to do only one thing on one type of hardware — provide debugging monitor functions at a hardware level to IBM PC-type machines. It is this specialization that makes hardware debuggers so much easier to use for programmers trying to get a piece of code running.

Because this volume deals only with MS-DOS and associated Microsoft software, a detailed discussion of hardware debuggers and debugging would not be appropriate. Instead, a few popular hardware products that work with MS-DOS utilities are mentioned and a general discussion of debugging with hardware is presented.

Several manufacturers make hardware products that can be used for debugging. These products vary in the features offered and in their suitability for various kinds of debugging. Three of these products that can be used with SYMDEB are

- IBM Professional Debug Utility
- PC Probe and AT Probe from Atron Corporation
- Periscope from The Periscope Company, Inc.

These boards can be used with SYMDEB by specifying the /N switch when the program is started. When used in this way, however, the hardware provides little more than a source of NMIs to interrupt program execution; otherwise, SYMDEB runs as usual. This restriction may not be acceptable to a programmer who wants to use the sophisticated debugging software that accompanies these products and makes use of their hardware features. For this reason, these boards are rarely used with SYMDEB.

The general techniques of debugging with hardware aids will already be familiar to the reader — they are the same techniques discussed at length earlier in this article. The techniques of inspection and observation should still be applied; instrumentation is facilitated by hardware; a debugging monitor accompanies all hardware debuggers and the same techniques discussed for DEBUG, SYMDEB, and CodeView apply. No new techniques are needed to use these devices. The changes in the details of the techniques come with the added features available with the hardware debuggers. (Remember that all these features are not universally available on all hardware debuggers.)

The manual interrupt feature of hardware debuggers is useful in a system crash. Every programmer, especially assembly-language programmers, has had the situation where the program runs wild, destroys the operating system, and locks up the system. The techniques described in previous sections of this article show that about the only way to solve these problems without hardware help is to set breakpoints at strategic locations in the program and see how many are passed before the system locks up. The breakpoints are placed at finer and finer increments until the instruction causing the crash is located.

This long and ugly procedure can sometimes be shortened with a hardware debugger. When the system crashes, the programmer can push the manual interrupt button, suspend program execution, and give control to the debugger card. At this point, the programmer can use the debugging monitor software supplied with the card to sniff around memory looking for something suspicious. Clues can sometimes be found by examining the program's stack and data areas — provided, of course, that they are still in memory and haven't been destroyed, along with the operating system, by the rampaging program. This approach is not always an immediate solution to the problem, however; often, the start-and-set-breakpoints process has to be repeated even with a hardware debugger. The hardware will, however, possibly shed some light on the causes of the problem and shorten the procedure.

Another feature offered by many of the debugging boards is the ability to set breakpoints on events other than the execution of a line of code. Often, these boards will allow the programmer to break on a reference to a specific memory location, to a range of memory locations, or to an I/O port. This feature allows a watch to be set on data, analogous to the watchpoint feature of CodeView. This technique is almost always useful, as it is with CodeView, but there is one class of problems where it is *essential* to reaching a solution.

Consider the case of a program that seems to be running well. Every so often, however, an ampersand appears in the middle of a payroll amount, or occasionally the program makes an erroneous branch and executes the wrong path. Suppose that, after painstaking investigation, the programmer discovers that these problems are being caused by a change in a specific location in memory sometime during the execution of the program. In debugging, the discovery of the cause of a problem usually leads almost instantly to a fix. Not so in this case. That byte of memory could be changed by an error in the program, by a glitch in the operating system or in a device driver, or by cosmic rays from outer space. Discovering the culprit in a case like this is almost impossible without the help of hardware breakpoints. Setting a breakpoint on the affected memory location and running the program will solve the problem. As soon as the memory location is changed, the breakpoint will be executed and the state of the system registers will point a clear finger at the instruction that caused the problem.

Hardware debuggers can provide significant aid to the serious programmer. They are especially helpful in debugging operating systems and operating-system services such as device drivers. They are also helpful in complicated situations where many programs may be running at the same time. The consensus among programmers who have hardware debuggers is that they are well worth the money.

Summary

Although Microsoft and others have provided an impressive array of technology to aid in program debugging, the most important tool a programmer has is his or her native wit and talent. As the examples in this article have illustrated, the technology makes the task easier, but never easy. In all cases, however, it is the programmer who debugs the program and solves the problems.

Technology will never be able to replace the person for solving the problem of a bug-ridden program. (This is an area where artificial intelligence will undoubtedly fail.) Therefore, it is the skills discussed in the first part of this article — debugging by inspection and observation — that deserve the greatest attention and practice. All the other techniques and technologies, with their ever-increasing sophistication, are only extensions of these basic techniques. A programmer who can debug effectively at the lowest level of technology will always be ready to use whatever advanced technology is available.

Therefore, as a final word, remember the rule that opened this article:

Gather enough information and the solution will be obvious.

All the rest of this article was merely a discussion of ways to gather the information.

Steve Bostwick

Article 19
Object Modules

Object modules are used primarily by programmers. The end user of an MS-DOS application need never be concerned with object code, object modules, and object libraries because application programs are almost always distributed as .EXE or .COM files that can be executed with a simple startup command.

An application programmer writing in a high-level language can use object modules and object libraries without knowing either the format of object code or the details of what the utilities that process object modules, such as the Microsoft Library Manager (LIB) and the Microsoft Object Linker (LINK), are actually doing. Most application programmers simply regard the contents of an object module as a "black box" and trust their compilers and object module utility programs to do the right thing.

A programmer using assembly language or an assembly-language debugger such as DEBUG or SYMDEB, however, might want to know more about the content and function of object modules. The use of assembly language gives the programmer more control over the actual contents of object modules, so knowing how the modules are constructed and examining their contents can sometimes help with program debugging.

Finally, a programmer writing a compiler, an assembler, or a language translator must know the details of object module format and processing. To take advantage of LIB and LINK, a language translator must construct object modules that conform to the format and usage conventions specified by Microsoft.

Note: This article assumes some background knowledge of the process by which source code is converted into an executable file in the MS-DOS environment. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program; PROGRAMMING TOOLS: The Microsoft Object Linker; PROGRAMMING UTILITIES.

The Use of Object Modules

Although some MS-DOS language translators generate executable 8086-family machine code directly from source code, most produce object code instead. Typically, a translator processes each file of source code individually and leaves the resulting object module in a separate file bearing a .OBJ extension. The source-code files themselves remain unchanged. After all of a program's source-code modules have been translated, the resulting object modules can be linked into a single executable program. Because object modules frequently represent only a portion of a complete program, each source-code module usually contains instructions that indicate how its corresponding object code is to be combined with the object code in other object modules when they are linked.

The object code contained in each object module consists of a binary image of the program plus program structure information. This object code is not directly executable. The binary image corresponds to the executable code that will ultimately be loaded into memory for execution; it contains both machine code and program data. The program structure information includes descriptions of logical groupings defined in the source code (such as named subroutines or segments) and symbolic references to addresses in other object modules.

The program structure information is used by a linkage editor, or linker, such as Microsoft LINK to edit the binary image of the program contained in the object module. The linker combines the binary images from one or more object modules into a complete executable program.

The linker's output is a .EXE file — a file containing executable machine code that can be loaded into RAM and executed (Figure 19-1). The linker leaves intact all of the object modules it processes.

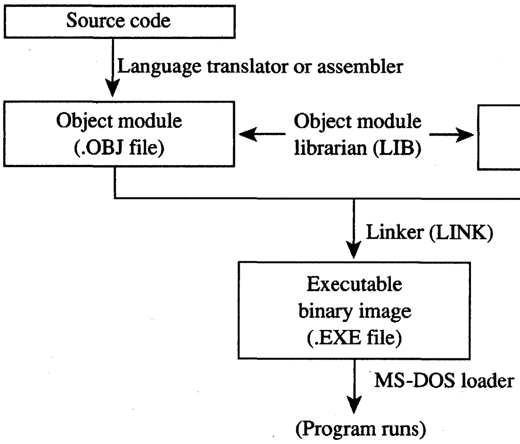

Source code
Language translator or assembler
Object module (.OBJ file)
Object module librarian (LIB)
Object library (.LIB file)
Linker (LINK)
Executable binary image (.EXE file)
MS-DOS loader
(Program runs)

Figure 19-1. Generation of an executable (.EXE) file.

Object code thus serves as an intermediate form for compiled programs. This form offers two major advantages:

- Modular intermediate code. The use of object modules eliminates the overhead of repeated compilation of an entire program whenever changes are made to parts of its source code. Instead, only those object modules affected by source-code revisions need be recompiled.
- Shareable format. Object module format is well defined, so object modules can be linked even if they were produced by different translators. Many high-level-language compilers take advantage of this commonality of object-code format to support "interlanguage" linkage.

Contents of an object module

Object modules contain five basic types of information. Some of this information exists explicitly in the source code (and is subsequently passed on to the object module), but much is inferred by the program translator from the structure of the source code and the way memory is accessed by the 8086.

Binary Image. As described earlier, the binary image comprises executable code (such as opcodes and addresses) and program data. When object modules are linked, the linker builds an executable program from the binary image in each object module it processes. The binary image in each object module is always associated with program structure information that tells the linker how to combine it with related binary images in other object modules.

External References. Because an object module generally represents only a small portion of a larger program that will be constructed from several object modules, it usually contains symbols that allow it to be linked to the other modules. Such references to corresponding symbols in other object modules are resolved when the modules are linked.

For example, consider the following short C program:

```
main()
{
        puts("Hello, world\n");
}
```

This program calls the C function *puts()* to display a character string, but *puts()* is not defined in the source code. Rather, the name *puts* is a reference to a function that is external to the program's *main()* routine. When the C compiler generates an object module for this program, it will identify *puts* as an external reference. Later, the linker will resolve the external reference by linking the object module containing the *puts()* routine with the module containing the *main()* routine.

Address References. When a program is built from a group of object modules, the actual values of many addresses cannot be computed until the linker combines the binary image of executable code and the program data from each of the program's constituent object modules. Object modules contain information that tells the linker how to resolve the values of such addresses, either symbolically (as in the case of external references) or relatively, in terms of some other address (such as the beginning of a block of executable code or program data).

Debugging Information. An object module can also contain information that relates addresses in the executable program to the corresponding source code. After the linker performs its address fixups, it can use the object module's debugging information to relate a line of source code in a program module to the executable code that corresponds to it.

Miscellaneous Information. Finally, an object module can contain comments, lists of symbols defined in or referenced by the module, module identification information, and information for use by an object library manager or a linker (for example, the names of object libraries to be searched by default).

Object module terminology

When the linker generates an executable program, it organizes the structural components of the program according to the information contained in the object modules. The layout of the executable program can be conceptually described as a run-time memory map after it has been loaded into memory.

The basic structure of every executable program for the 8086 family of microprocessors must conform to the segmented architecture of the microprocessor. Thus, the run-time memory map of an executable program is partitioned into segments, each of which can be addressed by using one of the microprocessor's segment registers. This segmented structure of 8086-based programs is the basis for most of the following terminology.

Frames. The memory address space of the 8086 is conceptually divided into a sequence of paragraph-aligned, overlapping 64 KB regions called frames. Frame 0 in the 8086's address space is the 64 KB of memory starting at physical address 00000H (0000:0000 in segment:offset notation), frame 1 is the 64 KB of memory starting at 00010H (0001:0000), and so on. A frame number thus denotes the beginning of any paragraph-aligned 64 KB of memory. For example, the location of a 64 KB buffer that starts at address B800:0000 can be specified as frame 0B800H.

Logical Segments. The run-time memory map for every 8086 program is partitioned into one or more logical segments, which are groupings of logically related portions of the program. Typically, an MS-DOS program includes at least one code segment (that contains all of the program's executable code), one or more data segments (that contain program data), and one stack segment.

When a program is loaded into RAM to be executed, each logical segment in the program can be addressed with a frame number — that is, a physical 8086 segment address. Before the MS-DOS loader transfers control to a program in memory, it initializes the CS and SS registers with the segment addresses of the program's executable code and stack segments. If an MS-DOS program has a separate logical segment for program data, the program itself usually stores this segment's address in the DS register.

Relocatable Segments. In MS-DOS programs, most logical segments are relocatable. The loader determines the physical addresses of a program's relocatable segments when it places the program into memory to be executed. However, this address determination poses a problem for the MS-DOS loader, because a program may contain references to the address of a relocatable segment even though the address value is not determined until the program is loaded. The problem is solved by indicating where such references occur within the program's object modules. The linker then extracts this information from the object modules and uses it to build a list of such address references into a segment relocation table in the header of executable files. After the loader copies a program into memory for execution, it uses the segment relocation table to update, or fix up, the segment address references within the program.

Consider the following example, in which a program loads the starting addresses of two data segments into the DS and ES segment registers:

```
mov     ax,seg _DATA
mov     ds,ax               ; make _DATA segment addressable through DS
mov     ax,seg FAR_DATA
mov     es,ax               ; make FAR_DATA segment addressable through ES
```

The actual addresses of the _DATA and FAR_DATA segments are unknown when the source code is assembled and the corresponding object module is constructed. The assembler indicates this by including segment fixup information, instead of actual segment addresses, in the program's object module. When the object module is linked, the linker builds this segment fixup information into the segment relocation table in the header of the program's .EXE file. Then, when the .EXE file is loaded, the MS-DOS loader uses the information in the .EXE file's header to patch the actual address values into the program.

Absolute Segments. Sometimes a program needs to address a predetermined segment of memory. In this case, the program's source code must declare an absolute segment so that a reference to the corresponding frame number can be built into the program's object module.

For example, a program might need to address a video display buffer located at a specific physical address. The following assembler directive declares the name of the segment and its frame number:

```
VideoBufferSeg    SEGMENT at 0B800h
```

Segment Alignment. When a program is loaded, the physical address of each logical segment is constrained by the segment's alignment. A segment can be page aligned (aligned on a 256-byte boundary), paragraph aligned (aligned on a 16-byte paragraph boundary), word aligned (aligned on an even-byte boundary), or byte aligned (not aligned on any particular boundary). A specification of each segment's alignment is part of every object module's program structure information.

High-level-language translators generally align segments according to the type of data they contain. For example, executable code segments are usually byte aligned; program data segments are usually word aligned. With an assembler, segment alignment can be specified with the SEGMENT directive and the assembler will build this information into the program's object module.

Concatenated Segments. The linker can concatenate logical segments from different object modules when it builds the executable program. For example, several object modules may each contain part of a program's executable code. When the linker processes these object modules, it can concatenate the executable code from the different object modules into one range of contiguous addresses.

The order in which the linker concatenates logical segments in the executable program is determined by the order in which the linker processes its input files and by the program structure information in the object modules. With a high-level-language translator, the translator infers which segments can be concatenated from the structure of the source code and builds appropriate segment concatenation information into the object modules it generates. With an assembler, the segment class type can be used to indicate which segments can be concatenated.

Groups of Segments. Segments with different names may also be grouped together by the linker so that they can all be addressed within the same 64 KB frame, even though they are not concatenated. For example, it might be desirable to group program data segments and a stack segment within the same 64 KB frame so that program data items and data on the stack can be addressed with the same 8086 segment register.

In high-level languages, it is up to the translator to incorporate appropriate segment grouping information into the object modules it generates. With an assembler, groups of segments can be declared with the GROUP directive.

Fixups. Sometimes a compiler or an assembler encounters addresses whose values cannot be determined from the source code. The addresses of external symbols are an obvious example. The addresses of relocatable segments and of labels within those segments are another example.

A fixup is a language translator's way of passing the buck about such addresses to the linker. Typically, a translator builds a zero value in the binary image at locations where it cannot store an actual address. Accompanying each such location is fixup information, which allows the linker to determine the correct address. The linker then completes the fixup by calculating the correct address value and adding it to the value in the corresponding location in the binary image. The only fixups the linker cannot fully resolve are those that refer to the segment address of a relocatable segment. Such addresses are not known until the program is actually loaded, so the linker, in turn, passes the responsibility to the MS-DOS loader by creating a segment relocation table in the header of the executable file.

To process fixups properly, the linker needs three pieces of information: the LOCATION of the value in the object module, the nature of the TARGET (the address whose value is not yet known), and the FRAME in which the address calculations are to take place. Object modules contain the LOCATION, TARGET, and FRAME information the linker uses to calculate the appropriate address for any given fixup.

Consider the "program" in Figure 19-2. The statement:

```
start:  call    far ptr FarProc
```

contains a reference to an address in the logical segment *FarSeg2*. Because the assembler does not know the address of *FarSeg2*, it places fixup information about the address into the object module. The LOCATION to be fixed up is 1 byte past the label *start* (the 4-byte pointer following the *call* opcode 9AH). The TARGET is the address referenced in the *call* instruction — that is, the label *FarProc* in the segment *FarSeg2*. The FRAME to which the fixup relates is designated by the group *FarGroup* and is inferred from the statement

```
ASSUME  cs:FarGroup
```

in the *FarSeg2* segment.

```
                                       title   fixups


                               FarGroup GROUP  FarSeg1,FarSeg2

0000                           CodeSeg SEGMENT byte public 'CODE'
                                       ASSUME  cs:CodeSeg

0000  9A 0000 ---- R           start:  call    far ptr FarProc

0005                           CodeSeg ENDS


0000                           FarSeg1 SEGMENT byte public     ;part of FarGroup

0000                           FarSeg1 ENDS


0000                           FarSeg2 SEGMENT byte public
                                       ASSUME  cs:FarGroup

0000                           FarProc PROC    far
0000  CB                               ret                     ;a FAR return
                               FarProc ENDP

0001                           FarSeg2 ENDS

                                       END
```

Figure 19-2. A sample "program" containing statements from which the assembler derives fixup information.

There are several different ways for a language translator to identify a fixup. For example, the LOCATION might be a single byte, a 16-bit offset, or a 32-bit pointer, as in Figure 19-2. The TARGET might be a label whose offset is relative either to the base (beginning) of a particular segment or to the LOCATION itself. The FRAME might be a relocatable segment, an absolute segment, or a group of segments.

Taken together, all the information in an object module that concerns the alignment and grouping of segments can be regarded as a specification of a program's run-time memory map. In effect, the object module specifies what goes where in memory when a program is loaded. The linker can then take the program structure information in the object modules and generate a file containing an executable program with the corresponding structure.

The Structure of an Object Module

Although object modules contain the information that ultimately determines the structure of an executable program, they bear little structural resemblance to the resulting executable program. Each object module is made up of a sequence of variable-length object records. Different types of object records contain different types of program information.

Each object record begins with a 1-byte field that identifies its type. This is followed by a 2-byte field containing the length (in bytes) of the remainder of the record. Next comes the actual structural or program information, represented in one or more fields of varied lengths. Finally, each record ends with a 1-byte checksum.

The sequence in which object records appear in an object module is important. Because the records vary in length, each object module must be constructed linearly, from start to end. More important, however, is the fact that some types of object records contain references to preceding object records. Because the linker processes object records sequentially, the position of each object record within an object module depends primarily on the type of information each record contains.

Types of object records

Microsoft LINK currently recognizes 14 types of object records, each of which carries a specific type of information within the object module. Each type of object record is assigned an identifying six-letter abbreviation, but these abbreviations are used only in documentation, not within an object module itself. As already mentioned, the first byte of each object record contains a value that indicates its type. In a hexadecimal dump of the contents of an object module, these identifying bytes identify the start of each object record.

Table 19-1 lists the types of object records supported by LINK. The value of each record's identifying byte (in hexadecimal) is included, along with the six-letter abbreviation and a brief functional description. The functions of the 14 types of object records fall into six general categories:

- Binary data (executable code and program data) is contained in the LEDATA and LIDATA records.
- Address binding and relocation information is contained in FIXUPP records.
- The structure of the run-time memory map is indicated by SEGDEF, GRPDEF, COMDEF, and TYPDEF records.
- Symbol names are declared in LNAMES, EXTDEF, and PUBDEF records.
- Debugging information is in the LINNUM record.
- Finally, the structure of the object module itself is determined by the THEADR, COMENT, and MODEND records.

Table 19-1. Types of 8086 Object Records Supported by Microsoft LINK.

ID byte	Abbreviation	Description
80H	THEADR	Translator Header Record
88H	COMENT	Comment Record
8AH	MODEND	Module End Record
8CH	EXTDEF	External Names Definition Record
8EH	TYPDEF	Type Definition Record
90H	PUBDEF	Public Names Definition Record
94H	LINNUM	Line Number Record
96H	LNAMES	List of Names Record
98H	SEGDEF	Segment Definition Record
9AH	GRPDEF	Group Definition Record
9CH	FIXUPP	Fixup Record
0A0H	LEDATA	Logical Enumerated Data Record
0A2H	LIDATA	Logical Iterated Data Record
0B0H	COMDEF	Communal Names Definition Record

Object record order

The sequence in which the types of object records appear in an object module is fairly flexible in some respects. Several record types are optional, and if the type of information they carry is unnecessary, they are omitted from an object module. In addition, most object record types can occur more than once in the same object module. And, because object records are variable in length, it is often possible to choose, as a matter of convenience, between combining information into one large record or breaking it down into several smaller records of the same type.

As stated previously, an important constraint on the order in which object records appear is the need for some types of object records to refer to information contained in other records. Because the linker processes the records sequentially, object records containing such information must precede the records that refer to it. For example, two types of object records, SEGDEF and GRPDEF, refer to the names contained in an LNAMES record. Thus, an LNAMES record must appear before any SEGDEF or GRPDEF records that refer to it so that the names in the LNAMES record are known to the linker by the time it processes the SEGDEF or GRPDEF records.

A typical object module

Figure 19-3 contains the source code for HELLO.ASM, an assembly-language program that displays a short message. Figure 19-4 is a hexadecimal dump of HELLO.OBJ, the object module generated by assembling HELLO.ASM with the Microsoft Macro Assembler. Figure 19-5 isolates the object records within the object module.

```
        NAME    HELLO

_TEXT   SEGMENT byte public 'CODE'

        ASSUME  cs:_TEXT,ds:_DATA

start:                                  ;program entry point
        mov     ax,seg msg
        mov     ds,ax
        mov     dx,offset msg           ;DS:DX -> msg
        mov     ah,09h
        int     21h                     ;perform int 21H function 09H
                                        ;(Output character string)
        mov     ax,4C00h
        int     21h                     ;perform int 21H function 4CH
                                        ;(Terminate with return code)
_TEXT   ENDS


_DATA   SEGMENT word public 'DATA'

msg     DB      'Hello, world',0Dh,0Ah,'$'

_DATA   ENDS


_STACK  SEGMENT stack 'STACK'

        DW      80h dup(?)              ;stack depth = 128 words

_STACK  ENDS

        END     start
```

Figure 19-3. The source code for HELLO.ASM.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000 80 07 00 05 48 45 4C 4C 4F 00 96 25 00 00 04 43  ....HELLO..%...C
0010 4F 44 45 04 44 41 54 41 05 53 54 41 43 4B 05 5F  ODE.DATA.STACK._
0020 44 41 54 41 06 5F 53 54 41 43 4B 05 5F 54 45 58  DATA._STACK._TEX
0030 54 8B 98 07 00 28 11 00 07 02 01 1E 98 07 00 48  T....(.........H
0040 0F 00 05 03 01 01 98 07 00 74 00 01 06 04 01 E1  .........t......
0050 A0 15 00 01 00 00 B8 00 00 8E D8 BA 00 00 B4 09  ................
0060 CD 21 B8 00 4C CD 21 D5 9C 0B 00 C8 01 04 02 02  .!..L.!.........
0070 C4 06 04 02 02 B6 A0 13 00 02 00 00 48 65 6C 6C  ............Hell
0080 6F 2C 20 77 6F 72 6C 64 0D 0A 24 A8 8A 07 00 C1  o, world..$.....
0090 00 01 01 00 00 AC                                ......
```

Figure 19-4. A hexadecimal dump of HELLO.OBJ.

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

THEADR
0000  80 07 00 05 48 45 4C 4C 4F 00                     ....HELLO.

LNAMES
0000                                96 25 00 00 04 43             .%...C
0010  4F 44 45 04 44 41 54 41 05 53 54 41 43 4B 05 5F   ODE.DATA.STACK._
0020  44 41 54 41 06 5F 53 54 41 43 4B 05 5F 54 45 58   DATA._STACK._TEX
0030  54 8B                                             T.

SEGDEF
0030        98 07 00 28 11 00 07 02 01 1E                 ...(......

SEGDEF
0030                                      98 07 00 48               ...H
0040  0F 00 05 03 01 01                                 ......

SEGDEF
0040                    98 07 00 74 00 01 06 04 01 E1         ...t......

LEDATA
0050  A0 15 00 01 00 00 B8 00 00 8E D8 BA 00 00 B4 09   ................
0060  CD 21 B8 00 4C CD 21 D5                           .!..L.!.

FIXUPP
0060                          9C 0B 00 C8 01 04 02 02           ........
0070  C4 06 04 02 02 B6                                 ......

LEDATA
0070                    A0 13 00 02 00 00 48 65 6C 6C         ......Hell
0080  6F 2C 20 77 6F 72 6C 64 0D 0A 24 A8               o, world..$.

MODEND
0080                                      8A 07 00 C1               ....
0090  00 01 01 00 00 AC                                 ......
```

Figure 19-5. The object records in HELLO.OBJ.

As shown most clearly in Figure 19-5, each of the object records begins with the single byte value identifying the record's type. The second and third bytes of each record contain a single 16-bit value, stored with its low-order byte first, that represents the length (in bytes) of the remainder of the object record.

The first record, THEADR, identifies the object module and the last record, MODEND, terminates the object module. The second record, LNAMES, contains a list of segment names and segment class names that LINK will use to lay out the run-time memory map. The three succeeding SEGDEF records describe the three corresponding segments defined in the source code.

The order in which the object records appear reflects both the structure of the source code and the record order constraints already mentioned. The LNAMES record appears before the three SEGDEF records because each SEGDEF record contains a reference to a name in the LNAMES record.

The binary data representing each of the two segments in the source code is contained in the two LEDATA records. The first LEDATA record represents the *_TEXT* segment; the second specifies the data in the *_DATA* segment. The FIXUPP record following the first LEDATA record contains information about the address references in the *_TEXT* segment. Again, the order in which the records appear is important: the FIXUPP record refers to the LEDATA record preceding it.

References between object records

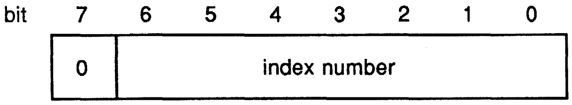
Object records can refer to information in other records either indirectly, by means of implicit references, or directly, by means of indexed references to names or other records.

Implicit References. Some types of object records implicitly reference another record in the same object module. The most important example of such implicit referencing is in the FIXUPP record, which always contains fixup information for the preceding LEDATA or LIDATA record in the object module. Whenever an LEDATA or LIDATA record contains a value that needs to be fixed up, the next record in the object module is always a FIXUPP record containing the actual fixup information.

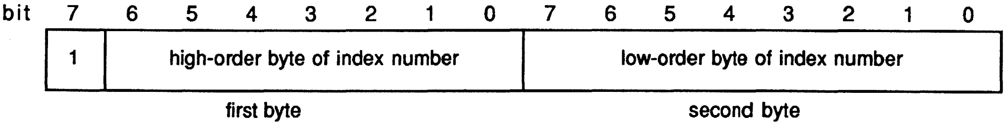
Indexed References to Names. An object record that refers to a symbolic name, such as the name of a segment or an external routine, uses an index into a list of names contained in a previous object record. (The LNAMES record in Figure 19-5 is an example.) The first name in such a list has the index number 1, the second name has index number 2, the third has index number 3, and so on. Altogether, a list of as many as 32,767 (7FFFH) names can be incorporated into an object module—generally adequate for even the most verbose programmer. (LINK does, however, impose its own version-specific limits.)

Indexed References to Object Records. An object record can also refer to a previous object record by using the same type of index. In this case, the index number refers to one of a list of object records of a particular type. For example, a FIXUPP record might refer to a segment by referencing one of several preceding SEGDEF records in the object module. In that case, a value of 1 would indicate the first SEGDEF record in the object module, a value of 2 would indicate the second, and so on.

The *index-number* field in an object record can be either 1 or 2 bytes long. If the number is in the range 0–7FH, the high-order bit (bit 7) is 0 and the low-order 7 bits contain the index number, so the field is only 1 byte long:



If the index number is in the range 80–7FFFH, the field is 2 bytes long. The high-order bit of the first byte in the field is set to 1, and the high-order byte of the index number (which must be in the range 0–7FH) fits in the remaining 7 bits. The low-order byte of the index number is specified in the second byte of the field:


bit
7
6
5
4
3
2
1
0
7
6
5
4
3
2
1
0
1
high-order byte of index number
low-order byte of index number
first byte
second byte

The same format is used whether an index refers to a list of names or to a previous object record.

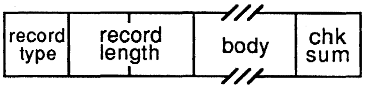
Microsoft 8086 Object Record Formats

Just as the design of the Intel 8086 microprocessor reflects the design of its 8-bit predecessors, 8086 object record formats are reminiscent of the 8-bit software tradition. In 8-bit systems, disk space and RAM were often at a premium. To minimize the space consumed by object records, information is packed into bit fields within bytes and variable-length fields are frequently used.

Microsoft LINK recognizes a major subset of Intel's original 8086 object module specification (Intel Technical Specification 121748-001). Intel also proposed a six-letter name for each type of object record and symbolic names for fields. These names are documented in the following descriptions, which appear in the order shown earlier in Table 19-1.

The Intel record types that are not recognized by LINK provide information about an executable program that MS-DOS obtains in other ways. (For example, information about run-time overlays is supplied in LINK's command line rather than being encoded in object records.) Because they are ignored by LINK, they are not included here.

All 8086 object records conform to the following format:



The *record type* field is a 1-byte field containing the hexadecimal number that identifies the type of object record (*see* Table 19-1).

The *record length* is a 2-byte field that gives the length of the remainder of the object record in bytes (excluding the bytes in the *record type* and *record length* fields). The record length is stored with the low-order byte first.

The *body* field of the record varies in size and content, depending on the record type.

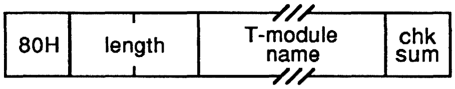
The *checksum* is a 1-byte field that contains the negative sum (modulo 256) of all other bytes in the record. In other words, the checksum byte is calculated so that the low-order byte of the sum of all the bytes in the record, including the checksum byte, equals zero.

Note: As shown in the preceding example, the boxes used to depict the fields vary in size. The square boxes used for *record type* and *chksum* indicate a single byte, the rectangular box used for *record length* indicates 2 bytes, and the diagonal lines used for *body* indicate a variable-length field.

80H THEADR Translator Header Record

The THEADR record contains the name of the object module. This name identifies an object module within an object library or in messages produced by the linker.

Record format



T-module name

The *T-module name* field is a variable-length field that contains the name of the object module. The first byte of the field contains the number of subsequent bytes that contain the name itself. The name can be uppercase or lowercase and can be any string of characters.

The *T-module name* is used by LIB and LINK within error messages. Language translators frequently derive the *T-module name* from the name of the file that contains a program's source code. Assembly-language programmers can specify the *T-module name* explicitly with the assembler NAME directive.

Location in object module

As its name implies, the THEADR record must be the first record in every object module generated by a language translator.

Example

The following THEADR record was generated by the Microsoft C Compiler:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  80 09 00 07 68 65 6C 6C 6F 2E 63 CB              ....hello.c.
```

- Byte 00H contains 80H, indicating a THEADR record.
- Bytes 01–02H contain 0009H, the length of the remainder of the record.
- Bytes 03–0AH contain the *T-module name.* Byte 03H contains 07H, the length of the name, and bytes 04H through 0AH contain the name itself (*hello.c*). (In object modules generated by the Microsoft C Compiler, the THEADR record indicates the filename that contained the C source code for the module.)
- Byte 0BH contains the checksum, 0CBH.

88H COMENT Comment Record

The COMENT record contains a character string that may represent a plain text comment, a symbol meaningful to a program such as LIB or LINK, or even binary-encoded identification data. An object module can contain any number of COMENT records.

Record format

88H	length	attrib	comment class	comment	chk sum

Attrib

Attrib is a 1-byte field in which only the first 2 bits are meaningful:

bit	7	6	5	4	3	2	1	0
	no purge	no list	0	0	0	0	0	0

- If bit 7 (*no purge*) is set to 1, utility programs that manipulate object modules should not delete the comment record from the object module. Bit 7 can thus protect an important comment, such as a copyright message, from deletion.
- If bit 6 (*no list*) is set to 1, utility programs that can list the contents of object modules are directed not to list the comment. Bit 6 can thus hide a comment.
- Bits 5 through 0 are unused and should be set to 0.

Microsoft LIB ignores the *attrib* field.

Comment class

Comment class is a 1-byte field whose value provides information about the type of comment. The original Intel specification provided for the following possible *comment class* values:

Value	Use
00H	Language-translator comment (the name of the translator that generated the object module).
01H	Copyright comment.
02–9BH	Reserved for Intel proprietary software.

Microsoft language translators can generate several other classes of COMENT record that communicate specific information about the object module to LINK:

Value	Use
81H	Obsolete; replaced by *comment class* 9FH.
9CH	MS-DOS version number. Some language translators create a COMENT record with a 2-byte binary value in the *comment* field indicating the MS-DOS version under which the module was created. This record is ignored by LINK.
9DH	Memory model. The *comment* field contains a string that indicates the memory model used by the language translator. The string contains one of the lowercase letters s, c, m, l, and h to designate small, compact, medium, large, and huge memory models. Microsoft language translators generate COMENT records with this *comment class* only for compatibility with the XENIX version of LINK. The MS-DOS version of LINK ignores these COMENT records.
9EH	Sets Microsoft LINK's DOSSEG switch.
9FH	Default library search name. LINK interprets the contents of the *comment* field as the name of a library to be searched in order to resolve external references within the object module. The default library search can be overridden with LINK's NODEFAULTLIBRARYSEARCH switch.
0A1H	Indicates that Microsoft extensions to the Intel object record specification are used in the object module. For example, when COMDEF records are used within an object module, a COMENT record with *comment class* 0A1H must appear in the object module at some point before the first COMDEF record. LINK ignores the *comment* string in COMENT records with this *comment class.*
0C0H–0FFH	Reserved for user-defined comment classes.

Comment

The *comment* field is a variable-length string of bytes that represent the comment. The length of the string is inferred from the length of the object record.

Location in object module

A COMENT record can appear almost anywhere in an object module. Only two restrictions apply:

- A COMENT record cannot be placed between a FIXUPP record and the LEDATA or LIDATA record to which it refers.
- A COMENT record cannot be the first or last record in an object module. (The first record must always be a THEADR record and the last must always be MODEND.)

Examples

The following three examples are typical COMENT records taken from an object module generated by the Microsoft C Compiler.

This first example is a language-translator comment:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 07 00 00 00 4D 53 20 43 6E                     .....MS Cn
```

- Byte 00H contains 88H, indicating that this is a COMENT record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.
- Byte 03H (the *attrib* field) contains 00H. Bit 7 (*no purge*) is set to 0, indicating that this COMENT record may be purged from the object module by a utility program that manipulates object modules. Bit 6 (*no list*) is set to 0, indicating that this comment need not be excluded from any listing of the module's contents. The remaining bits are all 0.
- Byte 04H (the *comment class* field) contains 00H, indicating that this COMENT record contains the name of the language translator that generated the object module.
- Bytes 05H through 08H contain the name of the language translator, MS C.
- Byte 09H contains the checksum, 6EH.

The second example contains the name of an object library to be searched by default when LINK processes the object module containing this COMENT record:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 09 00 00 9F 53 4C 49 42 46 50 10               .....SLIBFP.
```

- Byte 04H (the *comment class* field) contains 9FH, indicating that this record contains the name of a library for LINK to use to resolve external references.
- Bytes 05–0AH contain the library name, SLIBFP. In this example, the name refers to the Microsoft C Compiler's floating-point function library, SLIBFP.LIB.

The last example indicates that the object module contains Microsoft-defined extensions to the Intel object module specification:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  88 06 00 00 A1 01 43 56 37                        .....CV7
```

- Byte 04H indicates the *comment class*, 0A1H.
- Bytes 05–07H, which contain the *comment* string, are ignored by LINK.

8AH MODEND Module End Record

The MODEND record denotes the end of an object module. It also indicates whether the object module contains the main routine in a program, and it can, optionally, contain a reference to a program's entry point.

Record format

8AH	length	module type	start address	chk sum

Module type

The *module type* field is an 8-bit (1-byte) field:

bit	7	6	5	4	3	2	1	0
	main	start	0	0	0	0	0	1

- Bit 7 (*main*) is set to 1 if the module is a main program module.
- Bit 6 (*start*) is set to 1 if the MODEND record contains an entry point (*start address*).
- Bit 0 is set to 1 if the *start address* field contains a relocatable address reference that LINK must fix up. If bit 6 is set to 1, bit 0 must also be set to 1. (The Intel specification allows bit 0 to be set to 0, to indicate that *start address* is an absolute physical address, but this capability is not supported by LINK.)

Start address

The *start address* field appears in the MODEND record only when bit 6 is set to 1:

end dat	frame datum	target datum	target displacement

The format and interpretation of the *start address* field corresponds to the *fixup* field of the FIXUPP record. The *end dat* field corresponds to the *fix dat* field in the FIXUPP record. Bit 2 of the *end dat* field, which corresponds to the *P* bit in a *fix dat* field, must be zero.

Location in object module

A MODEND record can appear only as the last record in an object module.

Example

Consider the *MODEND* record of the HELLO.ASM example:

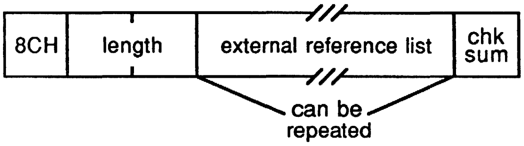
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8A 07 00 C1 00 01 01 00 00 AC                    ..........
```

- Byte 00H contains 8AH, indicating a MODEND record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.
- Byte 03H contains 0C1H (11000001B). Bit 7 is set to 1, indicating that this module is the main module of the program. Bit 6 is set to 1, indicating that a *start address* field is present. Bit 0 is set to 1, indicating that the address referenced in the *start address* field must be fixed up by LINK.
- Byte 04H (*end dat* in the *start address* field) contains 00H. As in a FIXUPP record, bit 7 indicates that the frame for this fixup is specified explicitly, and bits 6 through 4 indicate that a SEGDEF index specifies the frame. Bit 3 indicates that the target reference is also specified explicitly, and bits 2 through 0 indicate that a SEGDEF index also specifies the target. *See also* FIXUPP 9CH Fixup Record below.
- Byte 05H (*frame datum* in the *start address* field) contains 01H. This is a reference to the first SEGDEF record in the module, which in this example corresponds to the *_TEXT* segment. This reference tells LINK that the start address lies in the *_TEXT* segment of the module.
- Byte 06H (*target datum* in the *start address* field) contains 01H. This too is a reference to the first SEGDEF record in the object module, which corresponds to the *_TEXT* segment. LINK uses the following *target displacement* field to determine where in the *_TEXT* segment the address lies.
- Bytes 07–08H (*target displacement* in the *start address* field) contain 0000H. This is the offset (in bytes) of the *start address*.
- Byte 09H contains the checksum, 0ACH.

8CH EXTDEF External Names Definition Record

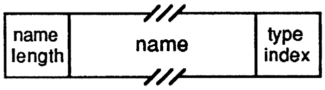
The EXTDEF record contains a list of symbolic external references — that is, references to symbols defined in other object modules. The linker resolves external references by matching the symbols declared in EXTDEF records with symbols declared in PUBDEF records.

Record format



External reference list

The *external reference list* is a variable-length field containing a list of names and name types, each formatted as follows:



- The *name length* is a 1-byte field containing the length of the *name* field that follows it. (LINK restricts *name length* to a value between 01H and 7FH.)
- The *type index* is a 1-byte reference to the TYPDEF record in the object module that describes the type of symbol the name represents. A *type index* value of zero indicates that no TYPDEF record is associated with the symbol. A nonzero value indicates which TYPDEF record is associated with the external name. Microsoft LINK recognizes TYPDEF records only for the purpose of declaring communal variables. *See* 8EH TYPDEF Type Definition Record below.

LINK imposes a limit of 1023 external names.

Location in object module

Any EXTDEF records in an object module must appear before the FIXUPP records that reference them. Also, if an EXTDEF record contains a nonzero *type index*, the indexed TYPDEF record must precede the EXTDEF record.

Example

Consider this EXTDEF record generated by the Microsoft C Compiler:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8C 25 00 0A 5F 5F 61 63 72 74 75 73 65 64 00 05   .%..__acrtused..
0010  5F 6D 61 69 6E 00 05 5F 70 75 74 73 00 08 5F 5F   _main.._puts..__
0020  63 68 6B 73 74 6B 00 A5                           chkstk..
```

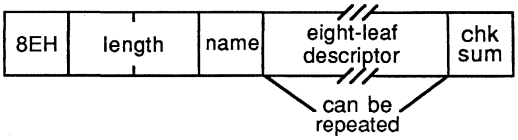
- Byte 00H contains 8CH, indicating that this is an EXTDEF record.
- Bytes 01–02H contain 0025H, the length of the remainder of the record.
- Bytes 03–26H contain a list of external references. The first reference starts in byte 03H, which contains 0AH, the length of the name __*acrtused.* The name itself follows in bytes 04–0DH. Byte 0EH contains 00H, which indicates that the symbol's type is not defined by any TYPDEF record in this object module. Bytes 0F–26H contain similar references to the external symbols _*main*, _*puts*, and __*chkstk.*
- Byte 27H contains the checksum, 0A5H.

8EH TYPDEF Type Definition Record

The TYPDEF record contains details about the type of data represented by a name declared in a PUBDEF or an EXTDEF record. This information may be used by the linker to validate references to names, or it may be used by a debugger to display data according to type.

Starting with Microsoft LINK version 3.50, the COMDEF record should be used for declaration of communal variables. For compatibility, however, later versions of LINK recognize TYPDEF records as well as COMDEF records.

Record format



Although the original Intel specification allowed for many different type specifications, such as scalar, pointer, and mixed data structure, LINK uses TYPDEF records to declare only communal variables. Communal variables represent globally shared memory areas — for example, FORTRAN common blocks or uninitialized public variables in C.

The size of a communal variable is declared explicitly in the TYPDEF record. If a communal variable has different sizes in different object modules, LINK uses the largest declared size when it generates an executable module.

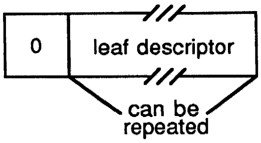
Name

The *name* field of a TYPDEF record is a 1-byte field that is always null; that is, it contains a single zero byte.

Eight-leaf descriptor

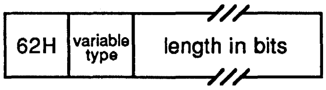
The *eight-leaf descriptor* field, in the original Intel specification, was a variable-length field that contained as many as eight "leaves" that could be used to describe mixed data structures.

Microsoft uses a stripped-down version of the *eight-leaf descriptor*, because the field's only function is to describe communal variables:



- The first field in the *eight-leaf descriptor* is a 1-byte field that contains a zero byte.
- The *leaf descriptor* field is a variable-length field that is itself divided into four fields ("leaves") that describe the size and type of a variable. The two possible variable types are NEAR and FAR.

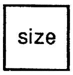
  If the field describes a NEAR variable (one that can be referenced as an offset within a default data segment), the format is



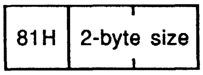
  - The 1-byte field containing 62H signifies a NEAR variable.
  - The *variable type* field is a 1-byte field that specifies the variable type:

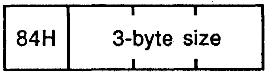
77H	Array
79H	Structure
7BH	Scalar

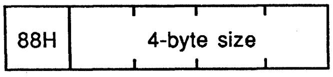
    This field is ignored by LINK.
  - The *length in bits* field is a variable-length field that indicates the size of the communal variable. Its format depends on the size it represents. If the size is less than 128 (80H) bits, *length in bits* is a 1-byte field containing the actual size of the field:



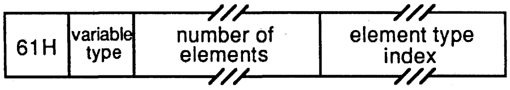
    If the size is 128 bits or greater, it cannot be represented in a single byte value, so the *length in bits* field is formatted with an extra initial byte that indicates whether the size is represented as a 2-, 3-, or 4-byte value:







If the *leaf descriptor* field describes a FAR variable (one that must be referenced with an explicit segment and offset), the format is



- The 1-byte field containing 61H signifies a FAR variable.
- The 1-byte *variable type* for a FAR communal variable is restricted to 77H (array). (As with the NEAR *variable type* field, LINK ignores this field.)
- The *number of elements* is a variable-length field that contains the number of elements in the array. It has the same format as the *length in bits* field in the *leaf descriptor* for a NEAR variable.
- The *element type index* is an index field that references a previous TYPDEF record. A value of 1 indicates the first TYPDEF record in the object module, a value of 2 indicates the second TYPDEF record, and so on. The TYPDEF record referenced must describe a NEAR variable. This way, the data type and size of the elements in the array can be determined.

Location in object module

Any TYPDEF records in an object module must precede the EXTDEF or PUBDEF records that reference them.

Examples

The following three examples of TYPDEF records were generated by the Microsoft C Compiler version 3.0. (Later versions use COMDEF records.)

The first sample TYPDEF record corresponds to the public declaration

```
int    foo;            /* 16-bit integer */
```

The TYPEDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 06 00 00 00 62 7B 10 7F                       .....b{..
```

- Byte 00H contains 8EH, indicating that this is a TYPDEF record.
- Bytes 01–02H contain 0006H, the length of the remainder of the record.
- Byte 03H (the *name* field) contains 00H, a null name.
- Bytes 04–07H represent the *eight-leaf descriptor* field. The first byte of this field (byte 04H) contains 00H. The remaining bytes (bytes 05–07H) represent the *leaf descriptor* field:
  - Byte 05H contains 62H, indicating this TYPDEF record describes a NEAR variable.
  - Byte 06H (the *variable type* field) contains 7BH, which describes this variable as a scalar.
  - Byte 07H (the *length in bits* field) contains 10H, the size of the variable in bits.

- Byte 08H contains the checksum, 7FH.

The next example demonstrates how the variable size contained in the *length in bits* field of the *leaf descriptor* is formatted:

```
char    foo2[32768];            /* 32 KB array */

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 09 00 00 00 62 7B 84 00 00 04 04              .....b{.....
```

- The *length in bits* field (bytes 07–0AH) starts with a byte containing 84H, which indicates that the actual size of the variable is represented as a 3-byte value (the following 3 bytes). Bytes 08–0AH contain the value 040000H, the size of the 32 KB array in bits.

This third C statement, because it declares a FAR variable, causes two TYPDEF records to be generated:

```
char    far     foo3[10][2][20];        /* 400-element FAR array */
```

The two TYPDEF records are

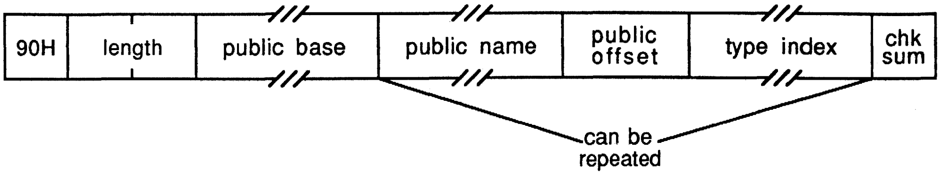
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  8E 06 00 00 00 62 7B 08 87 8E 09 00 00 00 61 77  .....b{.......aw
0010  81 90 01 01 7E                                   ....¦
```

- Bytes 00–08H contain the first TYPDEF record, which defines the data type of the elements of the array (NEAR, scalar, 8 bits in size).
- Bytes 09–14H contain the second TYPDEF record. The *leaf descriptor* field of this record declares that the variable is FAR (byte 0EH contains 61H) and an array (byte 0FH, the *variable type*, contains 77H).
  - Because this TYPDEF record describes a FAR variable, bytes 10–12H represent a *number of elements* field. The first byte of the field is 81H, indicating a 2-byte value, so the next 2 bytes (bytes 11–12H) contain the number of elements in the array, 0190H (400D).
- Byte 13H (the *element type index*) contains 01H, which is a reference to the first TYPDEF record in the object module — in this example, the one in bytes 00–08H.

90H PUBDEF Public Names Definition Record

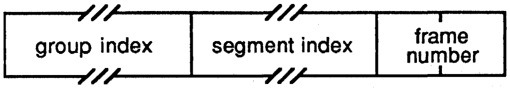
The PUBDEF record contains a list of public names. When object modules are linked, the linker uses these names to resolve external references in other object modules.

Record format



Public base

Each name in the PUBDEF record refers to a location (a 16-bit offset) in a particular segment or group. The *public base*, a variable-length field that specifies the segment or group, is formatted as follows:



- *Group index* is an index field that references a previous GRPDEF record in the object module. If the *group index* value is 0, no group is associated with this PUBDEF record.
- *Segment index* is also an index field. It associates a particular segment with this PUBDEF record by referencing a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on. If the *segment index* value is 0, the *group index* must also be 0—in this case, the *frame number* appears in the *public base* field.
- The 2-byte *frame number* appears in the *public base* field only when the *group index* and *segment index* are both 0. In other words, the *frame number* specifies the start of an absolute segment. If present, the value in the *frame number* field indicates the number of the frame containing the public name.

Public name

Public name is a variable-length field containing a public name. The first byte specifies the length of the name; the remainder is the name itself. (The Intel specification allows names of 1 to 255 bytes. Microsoft LINK restricts the maximum length of a public name to 127 bytes.)

Public offset

Public offset is a 2-byte field containing the offset of the location referred to by the *public name.* This offset is assumed to lie within the segment, group, or frame specified in the *public base* field.

Type index

Type index is an index field that references a previous TYPDEF record in the object module. A value of 1 indicates the first TYPDEF record in the module, a value of 2 indicates the second, and so on. The *type index* value can be 0 if no data type is associated with the public name.

The *public name, public offset,* and *type index* fields can be repeated within a single PUBDEF record. Thus, one PUBDEF record can declare a list of public names.

Location in object module

Any PUBDEF records in an object module must appear after the GRPDEF and SEGDEF records to which they refer. Because PUBDEF records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

Examples

The following two examples show PUBDEF records created by the Microsoft Macro Assembler.

The first example is the record for the statement

```
        PUBLIC  GAMMA
```

The PUBDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  90 0C 00 00 01 05 47 41 4D 4D 41 02 00 00 F9      ......GAMMA....
```

- Byte 00H contains 90H, indicating a PUBDEF record.
- Bytes 01–02H contain 000CH, the length of the remainder of the record.
- Bytes 03–04H represent the *public base* field. Byte 03H (the *group index*) contains 0, indicating that no group is associated with the name in this PUBDEF record. Byte 04H (the *segment index*) contains 1, a reference to the first SEGDEF record in the object module. This is the segment to which the name in this PUBDEF record refers.
- Bytes 05–0AH represent the *public name* field. Byte 05H contains 05H (the length of the name), and bytes 06–0AH contain the name itself, *GAMMA*.
- Bytes 0B–0CH contain 0002H, the *public offset.* The name *GAMMA* thus refers to the location that is offset 2 bytes from the beginning of the segment referenced by the *public base.*
- Byte 0DH is the *type index.* The value of the *type index* is 0, indicating that no data type is associated with the name *GAMMA.*
- Byte 0EH contains the checksum, 0F9H.

The next example is the PUBDEF record for the following absolute symbol declaration:

```
        PUBLIC    ALPHA
ALPHA   EQU       1234h
```

The PUBDEF record is

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  90 0E 00 00 00 00 00 05 41 4C 50 48 41 34 12 00  ......ALPHA4....
0010  B1
```

- Bytes 03–06H (the *public base* field) contain a *group index* of 0 (byte 03H) and a *segment index* of 0 (byte 04H). Since both the *group index* and *segment index* are 0, a *frame number* also appears in the *public base* field. In this instance, the *frame number* (bytes 05–06H) also happens to be 0.
- Bytes 07–0CH (the *public name* field) contain the name *ALPHA*, preceded by its length.
- Bytes 0D–0EH (the *public offset* field) contain 1234H. This is the value associated with the symbol *ALPHA* in the assembler EQU directive. If *ALPHA* is declared in another object module with the declaration

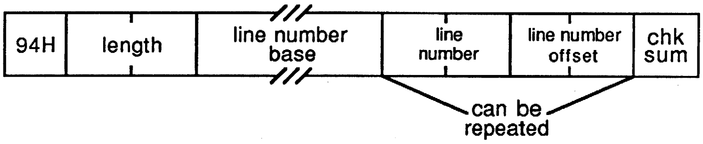
  ```
  EXTRN     ALPHA:ABS
  ```

  any references to *ALPHA* in that object module are fixed up as absolute references to offset 1234H in frame 0. In other words, *ALPHA* would have the value 1234H.
- Byte 0FH (the *type index*) contains 0.

94H LINNUM Line Number Record

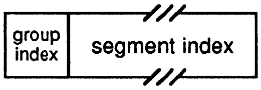
The LINNUM record relates line numbers in source code to addresses in object code.

Record format



Line number base

The *line number base* describes the segment to which the line number refers. Although the complete Intel specification allows the line number base to refer to a group or to an absolute segment as well as to a relocatable segment, Microsoft restricts references in this field to relocatable segments. The format of the *line number base* field is



- The *group index* field always contains a single zero byte.
- The *segment index* is an index field that references a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on.

Line number

Line number is a 2-byte field containing a line number between 0 and 32,767 (0–7FFFH).

Line number offset

The *line number offset* is a 2-byte field that specifies the offset of the executable code (in the segment specified in the *line number base* field) to which the line number in the *line number* field refers.

The *line number* and *line number offset* fields can be repeated, so a single LINNUM record can specify multiple line numbers in the same segment.

Location in object module

Any LINNUM records in an object module must appear after the SEGDEF records to which they refer. Because LINNUM records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

Example

The following LINNUM record was generated by the Microsoft C Compiler:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  94 0F 00 00 01 02 00 00 00 03 00 08 00 04 00 0F  ................
0010  00 3C                                            ..
```

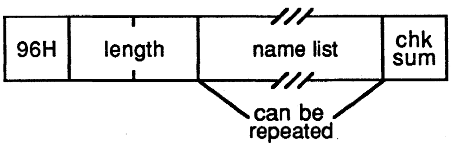
- Byte 00H contains 94H, indicating that this is a LINNUM record.
- Bytes 01–02H contain 000FH, the length of the remainder of the record.
- Bytes 03–04H represent the *line number base* field. Byte 03H (the *group index* field) contains 00H, as it must. Byte 04H (the *segment index* field) contains 01H, indicating that the line numbers in this LINNUM record refer to code in the segment defined in the first SEGDEF record in this object module.
- Bytes 05–06H (a *line number* field) contain 0002H, and bytes 07–08H (a *line number offset* field) contain 0000H. Together, they indicate that source-code line number 0002 corresponds to offset 0000H in the segment indicated in the *line number base* field.

  Similarly, the two pairs of *line number* and *line number offset* fields in bytes 09–10H specify that line number 0003 corresponds to offset 0008H and that line number 0004 corresponds to offset 000FH.
- Byte 11H contains the checksum, 3CH.

96H LNAMES List of Names Record

The LNAMES record is a list of names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

Record format



Name list

Name list is a variable-length field that contains the list of names. Each name is preceded by 1 byte that defines its length, which can be a value between 0 and 255 (0–0FFH).

The names in the list are indexed implicitly in the order they appear: The first name in the list has an index of 1, the second name has an index of 2, and so forth. References to the names contained in *name list* by subsequent object records, such as SEGDEF, are accomplished by using this index number. LINK imposes a limit of 255 logical names per object module.

Location in object module

Any LNAMES records in an object module must appear before the GRPDEF or SEGDEF records that refer to them. Because it does not refer to any other type of object records, an LNAMES record usually appears near the start of an object module.

Example

The following LNAMES record contains the segment and class names specified in all three of the assembler statements:

```
_TEXT    SEGMENT byte public 'CODE'
_DATA    SEGMENT word public 'DATA'
_STACK   SEGMENT para public 'STACK'
```

The LNAMES record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  96 25 00 00 04 43 4F 44 45 04 44 41 54 41 05 53  .%...CODE.DATA.S
0010  54 41 43 4B 05 5F 44 41 54 41 06 5F 53 54 41 43  TACK._DATA._STAC
0020  4B 05 5F 54 45 58 54 8B                          K._TEXT.
```

- Byte 00H contains 96H, indicating that this is an LNAMES record.
- Bytes 01–02H contain 0025H, the length of the remainder of the record.

- Byte 03H contains 00H, a zero-length name.
- Byte 04H contains 04H, the length of the class name CODE, which is found in bytes 05–08H. Bytes 09–26H contain the class names DATA and STACK and the segment names *_DATA, _STACK,* and *_TEXT,* each preceded by 1 byte giving its length.
- Byte 27H contains the checksum, 8BH.

98H SEGDEF Segment Definition Record

The SEGDEF record describes a logical segment in an object module. It defines the segment's name, length, and alignment, and the way the segment can be combined with other logical segments. LINK imposes a limit of 255 SEGDEF records per object module.

Object records that follow a SEGDEF record can refer to it to identify a particular segment.

Record format

98H	length	segment attributes	segment length	segment name index	class name index	overlay name index	chk sum

Segment attributes

Segment attributes is a variable-length field:

ACBP byte	frame number	offset

The ACBP byte

The contents and size of the *segment attributes* field depend on the first byte of the field, the ACBP byte:

bit	7	6	5	4	3	2	1	0
	A			C			B	P

The bit fields in the ACBP byte describe the following characteristics of the segment:

A	Alignment in the run-time memory map
C	Combination with other segments
B	Big (a segment of exactly 64 KB)
P	Page-resident (not used in MS-DOS)

The A field. Bits 7–5 of the ACBP byte, the *A* field, describe the logical segment's alignment:

A = 0 (000B)	Absolute (located at a specified frame address)
A = 1 (001B)	Relocatable, byte aligned
A = 2 (010B)	Relocatable, word aligned
A = 3 (011B)	Relocatable, paragraph aligned
A = 4 (100B)	Relocatable, page aligned

The original Intel specification includes two additional segment-alignment values not supported in MS-DOS.

The following examples of Microsoft assembler SEGMENT directives show the resulting values for the *A* field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400h               ; A = 0
bseg    SEGMENT byte public 'CODE'    ; A = 1
cseg    SEGMENT para stack 'STACK'    ; A = 3
```

The C field. Bits 4–2 of the ACBP byte, the *C* field, describe how the linker can combine the segment with other segments. Under MS-DOS, segments with the same name and class can be combined in two ways. They can be concatenated to form one logical segment, or they can be overlapped. In the latter case, they have either the same starting address or the same end address and they describe a common area of memory.

The value in the *C* field corresponds to one of these two methods of combining segments. Meaningful values, however, also depend on whether the segment is absolute (A = 0) or relocatable (A = 1, 2, 3, or 4). If A = 0, then *C* must also be 0, because absolute segments cannot be combined. Values for the *C* field are

C = 0 (000B) Cannot be combined; used for segments whose combine type is not explicitly specified (private segments).

C = 1 (001B) Not used by Microsoft.

C = 2 (010B) Can be concatenated with another segment of the same name; used for segments with the *public* combine type.

C = 3 (011B) Undefined.

C = 4 (100B) As defined by Microsoft, same as C = 2.

C = 5 (101B) Can be concatenated with another segment with the same name; used for segments with the *stack* combine type.

C = 6 (110B) Can be overlapped with another segment with the same name; used for segments with the *common* combine type.

C = 7 (111B) As defined by Microsoft, same as C = 2.

The following examples of assembler SEGMENT directives show the resulting values for the *C* field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400H              ; C = 0
bseg    SEGMENT public 'DATA'        ; C = 2
cseg    SEGMENT stack 'STACK'        ; C = 5
dseg    SEGMENT common 'COMMON'      ; C = 6
```

See PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: The Microsoft Object Linker.

The B and P fields. Bit 1 of the ACBP byte, the *B* field, is set to 1 (and the *segment length* field is set to 0) only if the segment is exactly 64 KB long.

Bit 0 of the ACBP byte, the *P* field, is unused in MS-DOS. Its value should always be 0.

Frame number and offset

The *frame number* and *offset* fields of the *segment attributes* field are present only if the segment is an absolute segment (A = 0 in the ACBP byte). Taken together, the *frame number* and *offset* indicate the starting address of the segment.

- *Frame number* is a 2-byte field that contains the frame number of the start of the segment.
- *Offset* is a 1-byte field that contains an offset between 00H and 0FH within the specified frame. LINK ignores the *offset* field.

Segment length

Segment length is a 2-byte field that specifies the length of the segment in bytes. The length can be from 00H to FFFFH. If a segment is exactly 64 KB (10000H) in size, *segment length* should be 0 and the *B* field in the ACBP byte should be 1.

Segment name index, class name index, and overlay name index

Each of the *segment name index*, *class name index*, and *overlay name index* fields contains an index into the list of names defined in previous LNAMES records in the object module. An index value of 1 indicates the first name in the LNAMES record, a value of 2 the second, and so on.

- The *segment name index* identifies the segment with a unique name. The name may have been assigned by the programmer, or it may have been generated by a compiler.
- The *class name index* identifies the segment with a class name (such as CODE, FAR_DATA, and STACK). The linker places segments with the same class name into a contiguous area of memory in the run-time memory map.
- The *overlay name index* identifies the segment with a run-time overlay. Starting with version 2.40, however, LINK ignores the *overlay name index*. In versions 2.40 and later, command-line parameters to LINK, rather than information contained in object modules, determine the creation of run-time overlays.

Location in object module

SEGDEF records must follow the LNAMES record to which they refer. In addition, SEGDEF records must precede any PUBDEF, LINNUM, GRPDEF, FIXUPP, LEDATA, or LIDATA records that refer to them.

Examples

In this first example, the segment is byte aligned:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  98 07 00 28 11 00 07 02 01 1E                  ...(......
```

- Byte 00H contains 98H, indicating that this is a SEGDEF record.
- Bytes 01–02H contain 0007H, the length of the remainder of the record.

- Byte 03H contains 28H (00101000B), the ACBP byte. Bits 7–5 (the *A* field) contain 1 (001B), indicating that this segment is relocatable and byte aligned. Bits 4–2 (the *C* field) contain 2 (010B), which represents a *public* combine type. (When this object module is linked, this segment will be concatenated with all other segments with the same name.) Bit 1 (the *B* field) is 0, indicating that this segment is smaller than 64 KB. Bit 0 (the *P* field) is ignored and should be zero, as it is here.
- Bytes 04–05H contain 0011H, the size of the segment in bytes.
- Bytes 06–08H index the list of names defined in the module's LNAMES record. Byte 06H (*the segment name index*) contains 07H, so the name of this segment is the seventh name in the LNAMES record. Byte 07H (the *class name index*) contains 02H, so the segment's class name is the second name in the LNAMES record. Byte 08H (the *overlay name index*) contains 1, a reference to the first name in the LNAMES record. (This name is usually null, as MS-DOS ignores it anyway.)
- Byte 09H contains the checksum, 1EH.

The second SEGDEF record declares a word-aligned segment. It differs only slightly from the first.

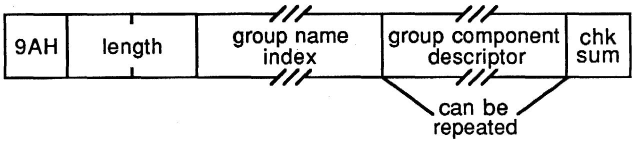
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  98 07 00 48 0F 00 05 03 01 01                    ...H......
```

- Bits 7–5 (the *A* field) of byte 03H (the ACBP byte) contain 2 (010B), indicating that this segment is relocatable and word aligned.
- Bytes 04–05H contain the size of the segment, 000FH.
- Byte 06H (the *segment name index*) contains 05H, which refers to the fifth name in the previous LNAMES record.
- Byte 07H (the *class name index*) contains 03H, a reference to the third name in the LNAMES record.

9AH GRPDEF Group Definition Record

The GRPDEF record defines a group of segments, all of which lie within the same 64 KB frame in the run-time memory map. LINK imposes a limit of 21 GRPDEF records per object module.

Record format

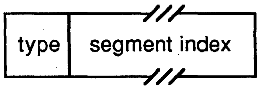


Group name index

Group name index is an index field whose value refers to a name in the *name list* field of a previous LNAMES record.

Group component descriptor

The *group component descriptor* consists of two fields:



- *Type* is a 1-byte field whose value is always 0FFH, indicating that the following field contains a *segment index* value. The original Intel specification defines four other types of *group component descriptor* with the values 0FEH, 0FDH, 0FBH, and 0FAH. LINK ignores these other *type* values, however, and assumes that the *group component descriptor* contains a *segment index* value.
- The *segment index* field contains an index number that refers to a previous SEGDEF record. A value of 1 indicates the first SEGDEF record in the object module, a value of 2 indicates the second, and so on.

The *group component descriptor* field is usually repeated within the GRPDEF record, so all segments constituting the group can be included in one GRPDEF record.

Location in object module

GRPDEF records must follow the LNAMES and SEGDEF records to which they refer. They must also precede any PUBDEF, LINNUM, FIXUPP, LEDATA, or LIDATA records that refer to them.

Example

The following example of a GRPDEF record corresponds to the assembler directive:

```
tgroup  GROUP seg1,seg2,seg3
```

The GRPDEF record is

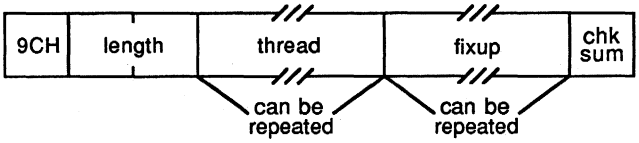
```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9A 08 00 06 FF 01 FF 02 FF 03 55                   ..........U
```

- Byte 00H contains 9AH, indicating that this is a GRPDEF record.
- Bytes 01–02H contain 0008H, the length of the remainder of the record.
- Byte 03H contains 06H, the *group name index.* In this instance, the index number refers to the sixth name in the previous LNAMES record in the object module. That name is the name of the group of segments defined in the remainder of the record.
- Bytes 04–05H contain the first of three *group component descriptor* fields. Byte 04H contains the required 0FFH, indicating that the subsequent field is a *segment index.* Byte 05H contains 01H, a *segment index* that refers to the first SEGDEF record in the object module. This SEGDEF record declared the first of three segments in the group.
- Bytes 06–07H represent the second *group component descriptor,* this one referring to the second SEGDEF record in the object module.
- Similarly, bytes 08–09H are a *group component descriptor* field that references the third SEGDEF record.
- Byte 0AH contains the checksum, 55H.

9CH FIXUPP Fixup Record

The FIXUPP record contains information that allows the linker to resolve (fix up) addresses whose values cannot be determined by the language translator. FIXUPP records describe the LOCATION of each address value to be fixed up, the TARGET address to which the fixup refers, and the FRAME relative to which the address computation is performed.

Record format



Thread and fixup fields

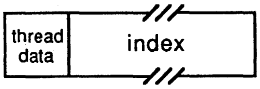
A FIXUPP record can contain zero or more *thread* fields and zero or more *fixup* fields. Each *fixup* field describes the method to be used by the linker to compute the TARGET address to be placed at a particular location in the executable image, relative to a particular FRAME. The information that determines the LOCATION, TARGET, and FRAME can be specified explicitly in the *fixup* field. It can also be specified within the *fixup* field by a reference to a previous *thread* field.

A *thread* field describes only the method to be used by the linker to refer to a particular TARGET or FRAME. Because the same *thread* field can be referenced in several subsequent *fixup* fields, a FIXUPP record that uses *thread* fields may be smaller than one in which *thread* fields are not used.

Thread and *fixup* fields are distinguished from one another by the high-order bit of the first byte in the field. If the high-order bit is 0, the field is a *thread* field. If the high-order bit is 1, the field is a *fixup* field.

The thread field

A *thread* field contains information that can be referenced in subsequent *thread* or *fixup* fields in the same or subsequent FIXUPP records. It has the following format:



The *thread data* field is a single byte comprising five subfields:

bit	7	6	5	4	3	2	1	0
	0	D	0	method			thread number	

- Bit 7 of the *thread data* byte is 0, indicating the start of a *thread* field.
- The *D* field (bit 6) indicates whether the *thread* field specifies a FRAME or a TARGET. The *D* bit is set to 1 to indicate a FRAME or to 0 to indicate a TARGET.
- Bit 5 of the *thread data* byte is not used. It should always be set to 0.
- Bits 4 through 2 represent the *method* field. If *D* = 1, the *method* field contains 0, 1, 2, 4, or 5. Each of these numbers corresponds to one method of specifying a FRAME (*see* Table 19-2). If *D* = 0, the *method* field contains 0, 1, 2, 4, 5, or 6, each of which corresponds to one of the methods of specifying a TARGET (*see* Table 19-3).

  In the case of a TARGET address, only bits 3 and 2 of the *method* field are used. When *D* = 0, the high-order bit of the value in the *method* field is derived from the *P* bit in the *fix dat* field of any subsequent *fixup* field that refers to this *thread* field. Thus, if *D* = 0, bit 4 of the *method* field is also 0, and the only meaningful values for the *method* field are 0, 1, and 2.
- The *thread number* field (bits 1 and 0) contains a number between 0 and 3. This number is used in subsequent *fixup* or *thread* fields to refer to this particular *thread* field.

  The *thread number* is implicitly associated with the *D* field by the linker, so as many as eight different *thread* fields (four FRAMEs and four TARGETs) can be referenced at any time. A *thread number* can be reused in an object module and, if it is, always refers to the *thread* field in which it last appeared.

Table 19-2. FRAME Fixup Methods.

Method	Description
0	The FRAME is specified by a segment index.
1	The FRAME is specified by a group index.
2	The FRAME is indicated by an external index. LINK determines the FRAME from the external name's corresponding PUBDEF record in another object module, which specifies either a logical segment or a group.
3	The FRAME is identified by an explicit frame number. (Not supported by LINK.)
4	The FRAME is determined by the segment in which the LOCATION is defined. In this case, the largest possible frame number is used.
5	The FRAME is determined by the TARGET's segment, group, or external index.

Table 19-3. TARGET Fixup Methods.

Method	Description
0	The TARGET is specified by a segment index and a displacement. The displacement is given in the *target displacement* field of the FIXUPP record.
1	The TARGET is specified by a group index and a *target displacement.*
2	The TARGET is specified by an external index and a *target displacement.* LINK adds the displacement to the address it determines from the external name's corresponding PUBDEF record in another object module.
3	The TARGET is identified by an explicit frame number. (Not supported by LINK.)
4*	The TARGET is specified by a segment index only.
5*	The TARGET is specified by a group index only.
6*	The TARGET is specified by an external index. The TARGET is the address associated with the external name.
7*	The TARGET is identified by an explicit frame number. (Not supported by LINK.)

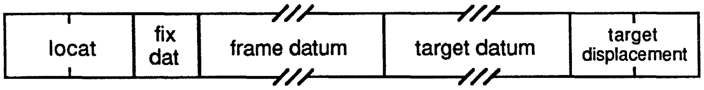
*TARGET methods 4–7 are analogous to the preceding four, except that methods 4–7 do not use an explicit displacement to identify the TARGET. Instead, a displacement of 0 is assumed.

The *index* field either contains an index value that refers to a previous SEGDEF, GRPDEF, or EXTDEF record, or it contains an explicit frame number. The interpretation of the index value depends on the value of the *method* field of the *thread data* field:

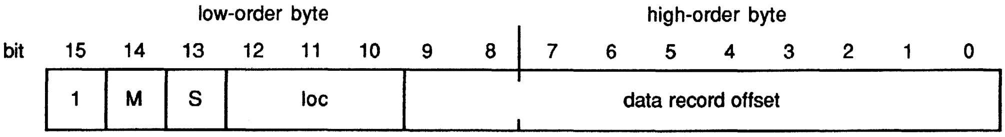
method = 0	Segment index (reference to a previous SEGDEF record)
method = 1	Group index (reference to a previous GRPDEF record)
method = 2	External index (reference to a previous EXTDEF record)
method = 3	Frame number (not supported by LINK; ignored)

The fixup field

The *fixup* field provides the information needed by the linker to resolve a reference to a relocatable or external address. The *fixup* field has the following format:



The 2-byte *locat* field has an unusual format. Contrary to the usual byte order in Intel data structures, the most significant bits of the *locat* field are found in the low-order, rather than the high-order, byte:



- Bit 15 (the high-order bit of the *locat* field) contains 1, indicating that this is a *fixup* field.
- Bit 14 (the *M* bit) is 1 if the fixup is segment relative and 0 if the fixup is self-relative.
- Bit 13 (the *S* bit) is currently unused and should always be set to 0.
- Bits 12 through 10 represent the *loc* field. This field contains a number between 0 and 5 that indicates the type of LOCATION to be fixed up:

loc = 0	Low-order byte
loc = 1	Offset
loc = 2	Segment
loc = 3	Pointer (segment:offset)
loc = 4	High-order byte (not recognized by LINK)
loc = 5	Loader-resolved offset (treated as *loc* = 1 by the linker)

- Bits 9 through 0 (the *data record offset*) indicate the position of the LOCATION to be fixed up in the LEDATA or LIDATA record immediately preceding the FIXUPP record. This offset indicates either a byte in the *data* field of an LEDATA record or a data byte in the *content* field of an *iterated data block* in an LIDATA record.

The *fix dat* field is a single byte comprising five fields:

bit	7	6	5	4	3	2	1	0
	F	frame			T	P	targt	

- Bit 7 (the *F* bit) is set to 1 if the FRAME for this fixup is specified by a reference to a previous *thread* field. The *F* bit is 0 if the FRAME method is explicitly defined in this *fixup* field.
- The interpretation of the *frame* field in bits 6 through 4 depends on the value of the *F* bit. If *F* = 1, the *frame* field contains a number between 0 and 3 that indicates the *thread* field containing the FRAME method. If *F* = 0, the *frame* field contains 0, 1, 2, 4, or 5, corresponding to one of the methods of specifying a FRAME listed in Table 19-2.
- Bit 3 (the *T* bit) is set to 1 if the TARGET for the fixup is specified by a reference to a previous *thread* field. If the *T* bit is 0, the TARGET is explicitly defined in this *fixup* field.
- Bit 2 (the *P* bit) and bits 1 and 0 (the *targt* field) can be considered a 3-bit field analogous to the *frame* field.
- If the *T* bit indicates that the TARGET is specified by a previous *thread* reference (*T* = 1), the *targt* field contains a number between 0 and 3 that refers to a previous *thread* field containing the TARGET method. In this case, the *P* bit, combined with the 2 low-order bits of the *method* field in the *thread* field, determines the TARGET method.

If the *T* bit is 0, indicating that the target is explicitly defined, the *P* and *targt* fields together contain 0, 1, 2, 4, 5, or 6. This number corresponds to one of the TARGET fixup methods listed in Table 19-3. (In this case, the *P* bit can be regarded as the high-order bit of the method number.)

Frame datum is an index field that refers to a previous SEGDEF, GRPDEF, or EXTDEF record, depending on the FRAME method.

Similarly, the *target datum* field contains a segment index, a group index, or an external index, depending on the TARGET method.

The *target displacement* field, a 2-byte field, is present only if the *P* bit in the *fixdat* field is set to 0, in which case the *target displacement* field contains the 16-bit offset used in methods 0, 1, and 2 of specifying a TARGET.

Location in object module

FIXUPP records must appear after the SEGDEF, GRPDEF, or EXTDEF records to which they refer. In addition, if a FIXUPP record contains any *fixup* fields, it must immediately follow the LEDATA or LIDATA record to which the fixups refer.

Examples

Although crucial to the proper linking of object modules, FIXUPP records are terse: Almost every bit is meaningful. For these reasons, the following three examples of FIXUPP records are particularly detailed.

A good way to understand how a FIXUPP record is put together is to compare it to the corresponding source code. The Microsoft Macro Assembler is helpful in this regard, because it marks in its source listing address references it cannot resolve. The "program" in Figure 19-6 is designed to show how some of the most frequently encountered fixups are encoded in FIXUPP records.

```
                                TITLE    fixupps
                         _TEXT  SEGMENT  byte public 'CODE'
                                ASSUME   cs:_TEXT
                                EXTRN    NearLabel:near
                                EXTRN    FarLabel:far

0000                     NearProc        PROC    near

0000  E9 0000 E                 jmp      NearLabel       ;relocatable word offset
0003  EB 00 E                   jmp      short NearLabel ;relocatable byte offset
0005  EA 0000 ---- R            jmp      far ptr FarProc ;far jump to a known seg
000A  EA 0000 ---- E            jmp      FarLabel     ;far jump to an unknown seg

000F  BB 0015 R                 mov      bx,offset LocalLabel ;relocatable offset
0012  B8 ---- R                 mov      ax,seg LocalLabel    ;relocatable seg
```

Figure 19-6. A sample "program" showing how some common fixups are encoded in FIXUPP records. (more)

```
0015  C3                      LocalLabel:     ret

                              NearProc        ENDP

0016                          _TEXT   ENDS


0000                          FAR_TEXT        SEGMENT byte public 'FAR_CODE'
                                      ASSUME  cs:FAR_TEXT

0000                          FarProc PROC    far

0000  CB                              ret

                              FarProc ENDP

0001                          FAR_TEXT        ENDS

                                      END
```

Figure 19-6. Continued.

The assembler generates one LEDATA record for this program:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0010  A0 1A 00 01 00 00 E9 00 00 EB 00 EA 00 00 00 00  ................
0020  EA 00 00 00 00 BB 00 00 B8 00 00 C3 67           ............g
```

Bytes 06–2BH (the *data* field) of this LEDATA record contain 8086 opcodes for each of the instruction mnemonics in the source code. The gaps (zero values) in the *data* field correspond to address values that the assembler cannot resolve. The linker will fix up the address values in the gaps by computing the correct values and adding them to the zero values in the gaps. The FIXUPP record that tells the linker how to do this immediately follows the LEDATA record in the object module:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

- Byte 00H contains 9CH, indicating this is a FIXUPP record.
- Bytes 01–02H contain 0021H, the length of the remainder of the record.
- Bytes 03–07H represent the first of the six *fixup* fields in this record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

The information in this *fixup* field will allow the linker to resolve the address reference in the statement

```
jmp    NearLabel
```

  - Bytes 03–04H (the *locat* field) contain 8401H (1000010000000001B). (Recall that this field does not conform to the usual Intel byte order.) Bit 15 is 1, signifying that this is a *fixup* field, not a *thread* field. Bit 14 (the *M* bit) is 0, so this fixup is self-relative. Bit 13 is unused and should be set to 0, as it is here. Bits 12–10 (the *loc* field) contain 1 (001B), so the LOCATION to be fixed up is a 16-bit offset. Bits 9–0 (the *data record offset*) contain 1 (0000000001B), which informs the linker that the LOCATION to be fixed up is at offset 1 in the *data* field of the LEDATA record immediately preceding this FIXUPP record—in other words, the 2 bytes immediately following the first opcode 0E9H.
  - Byte 05H (the *fix dat* field) contains 06H (00000110B). Bit 7 (the *F* bit) is 0, meaning the FRAME for this fixup is explicitly specified in this *fixup* field. Bits 6–4 (the *frame* field) contain 0 (000B), indicating that FRAME method 0 specifies the FRAME. Bit 3 (the *T* bit) is 0, so the TARGET for this fixup is also explicitly specified. Bits 2–0 (the *P* bit) and the *targt* field contain 6 (110B), so TARGET method 6 specifies the TARGET.
  - Byte 06H is a *frame datum* field, because the FRAME is explicitly specified (the *F* bit of the *fix dat* field = 0). And, because method 0 is specified, the *frame datum* is an index field that refers to a previous SEGDEF record. In this example, the *frame datum* field contains 1, which indicates the first SEGDEF record in the object module: the _TEXT segment.
  - Similarly, byte 07H is a *target datum*, because the TARGET is also explicitly specified (the *T* bit of the *fix dat* field = 0). The *fix dat* field also indicates that TARGET method 6 is used, so the *target datum* is an index field that refers to the *external reference list* in a previous EXTDEF record. The value of this index is 2, so the TARGET is the second external reference declared in the EXTDEF record: *NearLabel* in this object module.

- Bytes 08–0CH represent the second *fixup* field:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

  This *fixup* field corresponds to the statement

```
jmp     short NearLabel
```

  The only difference between this statement and the first is that the jump uses an 8-bit, rather than a 16-bit, offset. Thus, the *loc* field (bits 12–10 of byte 08H) contains 0 (000B) to indicate that the LOCATION to be fixed up is a low-order byte.

- Bytes 0D–11H represent the third *fixup* field in this FIXUPP record:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                      ....
```

  This *fixup* field corresponds to the statement

```
jmp     far ptr FarProc
```

In this case, both the TARGET's frame (the segment *FAR_TEXT*) and offset (the label *FarProc*) are known to the assembler. Both the segment address and the label offset are relocatable, however, so in the FIXUPP record the assembler passes the responsibility for resolving the addresses to the linker.

- Bytes 0D–0EH (the *locat* field) indicate that the field is a *fixup* field (bit 15 = 1) and that the fixup is segment relative (bit 14—the *M* bit = 1). The *loc* field (bits 12–10) contains 3 (011B), so the LOCATION being fixed up is a 32-bit (FAR) pointer (segment and offset). The *data record offset* (bits 9–0) is 6 (0000000110B); the LOCATION is the 4 bytes following the first far jump opcode (EAH) in the preceding LEDATA record.
- In byte 0FH (the *fix dat* field), the *F* bit and the *frame* field are 0, indicating that method 0 (a segment index) is used to specify the FRAME. The *T* bit is 0 (meaning the *target* is explicitly defined in the *fixup* field); therefore, the *P* bit and *targt* fields together indicate method 4 (a segment index) to specify the TARGET.
- Because the FRAME is specified with a segment index, byte 10H (the *frame datum* field) is a reference to the second SEGDEF record in the object module, which in this example declared the *FAR_TEXT* segment. Similarly, byte 11H (the *target datum* field) references the *FAR_TEXT* segment. In this case, the FRAME is the same as the TARGET segment; had *FAR_TEXT* been one of a group of segments, the FRAME could have referred to the group instead.

- The fourth assembler statement is different from the third because it references a segment not known to the assembler:

```
jmp     FarLabel
```

Bytes 12–16H contain the corresponding *fixup* field:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                       ....
```

The significant difference between this and the preceding *fixup* field is that the *P* bit and *targt* field of the *fix dat* byte (byte 14H) specify TARGET method 6. In this *fixup* field, the *target datum* (byte 16H) refers to the first EXTDEF record in the object module, which declares *FarLabel* as an external reference.

- The fifth *fixup* field (bytes 17–1DH) is

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
0020  04 01 01 A3                                       ....
```

This *fixup* field contains information that enables the linker to calculate the value of the relocatable offset *LocalLabel*:

```
mov     bx,offset LocalLabel
```

- Bytes 17–18H (the *locat* field) contain C410H (1100010000010000B). Bit 15 is 1, denoting a *fixup* field. The *M* bit (bit 14) is 1, indicating that this fixup is segment relative. The *loc* field (bits 12–10) contains 1 (001B), so the LOCATION is a 16-bit offset. The *data record offset* (bits 9–0) is 10H (0000010000B), a reference to the 2 bytes in the LEDATA record following the opcode 0BBH.
- Byte 19H (the *fix dat* byte) contains 00H. The *F* bit, *frame* field, *T* bit, *P* bit, and *targt* field are all 0, so FRAME method 0 and TARGET method 0 are explicitly specified in this *fixup* field.
- Because FRAME method 0 is used, byte 1AH (the *frame datum* field) is an index field. It contains 01H, a reference to the first SEGDEF record in the object module, which declares the segment *_TEXT*.

  Similarly, byte 1BH (the *target datum* field) references the *_TEXT* segment.
- Because TARGET method 0 is specified, an offset, in addition to a segment, is required to define the TARGET. This offset appears in the *target displacement* field in bytes 1C–1DH. The value of this offset is 0015H, corresponding to the offset of the TARGET (*LocalLabel*) in its segment (*_TEXT*).

● The sixth and final *fixup* field in this FIXUPP record (bytes 1E–22H) is

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
  0000  9C 21 00 84 01 06 01 02 80 04 06 01 02 CC 06 04  .!..............
  0010  02 02 CC 0B 06 01 01 C4 10 00 01 01 15 00 C8 13  ................
  0020  04 01 01 A3                                      ....
```

This corresponds to the segment of the relocatable address *LocalLabel*:

```
mov     ax,seg LocalLabel
```

- Bytes 1E–1FH (the *locat* field) contain C813H (1100100000010011B). Bit 15 is 1, so this is a *fixup* field. The *M* bit (bit 14) is 1, so the fixup is segment relative. The *loc* field (bits 12–10) contains 2 (010B), so the LOCATION is a 16-bit segment value. The *data record offset* (bits 9–0) indicates the 2 bytes in the LEDATA record following the opcode 0B8H.
- Byte 20H (the *fix dat* byte) contains 04H, so FRAME method 0 and TARGET method 4 are explicitly specified in this *fixup* field.
- Byte 21H (the *frame datum* field) contains 01H. Because FRAME method 0 is specified, the *frame datum* is an index value that refers to the first SEGDEF record in the object module (corresponding to the *_TEXT* segment).
- Byte 22H (the *target datum* field) contains 01H. Because TARGET method 4 is specified, the *target datum* also references the *_TEXT* segment.

● Finally, byte 23H contains this FIXUPP record's checksum, 0A3H.

The next two FIXUPP records show how *thread* fields are used. The first of the two contains six *thread* fields that can be referenced by both *thread* and *fixup* fields in subsequent FIXUPP records in the same object module:

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   9C 0D 00 00 03 01 02 02 01 03 04 40 01 45 01 C0  ..........@....
```

Bytes 03–04H, 05–06H, 07–08H, 09–0AH, 0B–0CH, and 0D–0EH represent the six *thread* fields in this FIXUPP record. The high-order bit of the first byte of each of these fields is 0, indicating that they are, indeed, *thread* fields and not *fixup* fields.

- Byte 03H, which contains 00H, is the *thread data* byte of the first *thread* field. Bit 7 of this byte is 0, indicating this is a *thread* field. Bit 6 (the *D* bit) is 0, so this field specifies a TARGET. Bit 5 is 0, as it must always be. Bits 4 through 2 (the *method* field) contain 0 (000B), which specifies TARGET method 0. Finally, bits 1 and 0 contain 0 (00B), the *thread number* that identifies this *thread* field.

  Byte 04H represents a segment *index* field, because method 0 of specifying a TARGET references a segment. The value of the index, 3, is a reference to the third SEGDEF record defined in the object module.
- Bytes 05–06H, 07–08H, and 09–0AH contain similar *thread* fields. In each, the *method* field specifies TARGET method 0. The three *thread* fields also have *thread numbers* of 1, 2, and 3. Because TARGET method 0 is specified for each *thread* field, bytes 06H, 08H, and 0AH represent segment *index* fields, which reference the second, first, and fourth SEGDEF records, respectively.
- Byte 0BH (the *thread data* byte of the fifth *thread* field in this FIXUPP record) contains 40H (01000000B). The *D* bit (bit 6) is 1, so this *thread* field specifies a FRAME. The *method* field (bits 4 through 2) contains 0 (000B), which specifies FRAME method 0. Byte 0CH (which contains 01H) is therefore interpreted as a segment *index* reference to the first SEGDEF record in the object module.
- Byte 0DH is the *thread data* byte of the sixth *thread* field. It contains 45H (01000101B). Bit 6 is 1, which indicates that this *thread* specifies a FRAME. The *method* field (bits 4 through 2) contains 1 (001B), which specifies FRAME method 1. Byte 0EH (which contains 01H) is therefore interpreted as a group *index* to the first preceding GRPDEF record.

  The *thread number* fields of the fifth and sixth *thread* fields contain 0 and 1, respectively, but these *thread numbers* do not conflict with the ones used in the first and second *thread* fields, because the latter represent TARGET references, not FRAME references.

The next FIXUPP example appears after the preceding record, in the same object module. This FIXUPP record contains a *fixup* field in bytes 03–05H that refers to a *thread* in the previous FIXUPP record:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  9C 04 00 C4 09 9D F6                              .......
```

- Bytes 03–04H represent the 16-bit *locat* field, which contains C409H (1100010000001001B). Bit 15 of the *locat* field is 1, indicating a *fixup* field. The *M* bit (bit 14) is 1, so this fixup is relative to a particular segment, which is specified later in the *fixup* field. Bit 13 is 0, as it should be. Bits 12–10 (the *loc* field) contain 1 (001B), so the LOCATION to be fixed up is a 16-bit offset. Bits 9–0 (the *data record offset* field) contain 9 (0000001001B), so the LOCATION to be fixed up is represented at an offset of 9 bytes into the data field of the preceding LEDATA or LIDATA record.

- Byte 05H (the *fix dat* byte) contains 9DH (10011101B). The *F* bit (bit 7) is 1, so this *fixup* field references a *thread* field that, in turn, defines the method of specifying the FRAME for the fixup. Bits 6–4 (the *frame* field) contain 1 (001B), the number of the *thread* that contains the FRAME method. This *thread* contains a *method* number of 1, which references the first GRPDEF record in the object module, thus specifying the FRAME.

  The *T* bit (bit 3 in the *fix dat* byte) is 1, so the TARGET method is also defined in a preceding *thread* field. The *targt* field (bits 1 and 0 in the *fix dat* byte) contains 1 (01B), so the TARGET *thread* field whose *thread number* is 1 specifies the TARGET. The *P* bit (bit 3 in the *fix dat* byte) contains 1, which is combined with the low-order bits of the *method* field in the *thread* field that describes the target to obtain TARGET method number 4 (100B). The TARGET *thread* references the second SEGDEF record to specify the TARGET.

The last FIXUPP example illustrates that the linker performs a fixup by adding the calculated address value to the value in the LOCATION being fixed up. This function of the linker can be exploited to use fixups to modify opcodes or program data, as well as to resolve address references.

Consider how the following assembler instruction might be fixed up:

```
lea  bx,alpha+10h    ; alpha is an external symbol
```

Typically, this instruction is translated into an LEDATA record with zero in the LOCATION (bytes 08–09H) to be fixed up:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   A0 08 00 01 00 00 8D 1E 00 00 AC                     ...........
```

The corresponding FIXUPP record contains a *target displacement* of 10H bytes (bytes 08–09H):

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   9C 08 00 C4 02 02 01 01 10 00 82                     ...........
```

This FIXUPP record specifies TARGET method 2, which is indicated by the *targt* field (bits 2–0) of the *fixdat* field (byte 05H). In this case, the linker adds the *target displacement* to the address it has determined for the TARGET (*alpha*) and then completes the fixup by adding this calculated address value to the zero value in the LOCATION.

The same result can be achieved by storing the displacement (10H) directly in the LOCATION in the LEDATA record:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   A0 08 00 01 00 00 8D 1E 10 00 9C                     ...........
```

Then, the *target displacement* can be omitted from the FIXUPP record:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   9C 06 00 C4 02 06 01 01 90                           .........
```

This FIXUPP record specifies TARGET method 6, which does not use a *target displacement.* The linker performs this fixup by adding the address of *alpha* to the value in the LOCATION, so the result is identical to the preceding one.

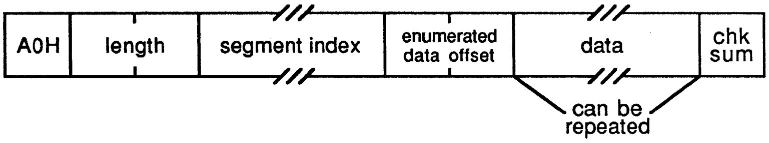
The difference between the two techniques is that in the latter the linker does not perform error checking when it adds the calculated fixup value to the value in the LOCATION. If this second technique is used, the linker will not flag arithmetic overflow or underflow errors when it adds the displacement to the TARGET address. The first technique, then, traps all errors; the second can be used when overflow or underflow is irrelevant and an error message would be undesirable.

0A0H LEDATA Logical Enumerated Data Record

The LEDATA record contains contiguous binary data — executable code or program data — that is eventually copied into the program's executable binary image.

The binary data in an LEDATA record can be modified by the linker if the record is followed by a FIXUPP record.

Record format



Segment index

The *segment index* is a variable-length index field. The index number in this field refers to a previous SEGDEF record in the object module. A value of 1 indicates the first SEGDEF record, a value of 2 the second, and so on. That SEGDEF record, in turn, indicates the segment into which the data in this LEDATA record is to be placed.

Enumerated data offset

The *enumerated data offset* is a 2-byte offset into the segment referenced by the *segment index*, relative to the base of the segment. Taken together, the *segment index* and the *enumerated data offset* fields indicate the location where the enumerated data will be placed in the run-time memory map.

Data

The *data* field contains the actual data, which can be either executable 8086 instructions or program data. The maximum size of the *data* field is 1024 bytes.

Location in object module

Any LEDATA records in an object module must be preceded by the SEGDEF records to which they refer. Also, if an LEDATA record requires a fixup, a FIXUPP record must immediately follow the LEDATA record.

Example

The following LEDATA record contains a simple text string:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  A0 13 00 02 00 00 48 65 6C 6C 6F 2C 20 77 6F 72   ......Hello, wor
0010  6C 64 0D 0A 24 A8                                  ld..$.
```

- Byte 00H contains 0A0H, which identifies this as an LEDATA record.
- Bytes 01–02H contain 0013H, the length of the remainder of the record.

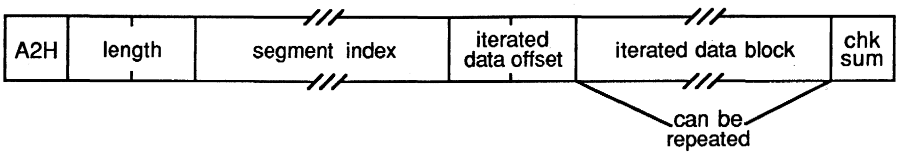
- Byte 03H (the *segment index* field) contains 02H, a reference to the second SEGDEF record in the object module.
- Bytes 04–05H (the *enumerated data offset* field) contain 0000H. This is the offset, from the base of the segment indicated by the *segment index* field, at which the data in the *data* field will be placed when the program is linked. Of course, this offset is subject to relocation by the linker because the segment declared in the specified SEGDEF record may be relocatable and may be combined with other segments declared in other object modules.
- Bytes 06–14H (the *data* field) contain the actual data.
- Byte 15H contains the checksum, 0A8H.

0A2H LIDATA Logical Iterated Data Record

Like the LEDATA record, the LIDATA record contains binary data — executable code or program data. The data in an LIDATA record, however, is specified as a repeating pattern (iterated), rather than by explicit enumeration.

The data in an LIDATA record may be modified by the linker if the LIDATA record is followed by a FIXUPP record.

Record format



Segment index

The *segment index* is a variable-length index field. The index number in this field refers to a previous SEGDEF record in the object module. A value of 1 indicates the first SEGDEF record, 2 indicates the second, and so on. That SEGDEF record, in turn, indicates the segment into which the data in this LIDATA record is to be placed when the program is executed.

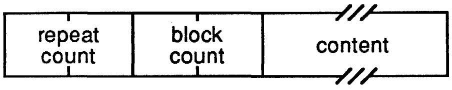
Iterated data offset

The *iterated data offset* is a 2-byte offset into the segment referenced by the *segment index*, relative to the base of the segment. Taken together, the *segment index* and the *iterated data offset* fields indicate the location where the iterated data will be placed in the run-time memory map.

Iterated data block

The *iterated data block* is a variable-length field containing the actual data — executable code and program data. *Iterated data blocks* can be nested, so one *iterated data block* can contain one or more other *iterated data blocks*. Microsoft LINK restricts the maximum size of an *iterated data block* to 512 bytes.

The format of the *iterated data block* is



- *Repeat count* is a 2-byte field indicating the number of times the *content* field is to be repeated.
- *Block count* is a 2-byte field indicating the number of *iterated data blocks* in the *content* field. If the *block count* is 0, the *content* field contains data only.

- *Content* is a variable-length field that can contain either nested *iterated data blocks* (if the *block count* is nonzero) or data (if the *block count* is 0). If the *content* field contains data, the field contains a 1-byte count of the number of data bytes in the field, followed by the actual data.

Location in object module

Any LIDATA records in an object module must be preceded by the SEGDEF records to which they refer. Also, if an LIDATA record requires a fixup, a FIXUPP record must immediately follow the LIDATA record.

Example

This sample LIDATA record corresponds to the following assembler statement, which declares a 10-element array containing the strings *ALPHA* and *BETA*:

```
db      10 dup('ALPHA','BETA')
```

The LIDATA record is

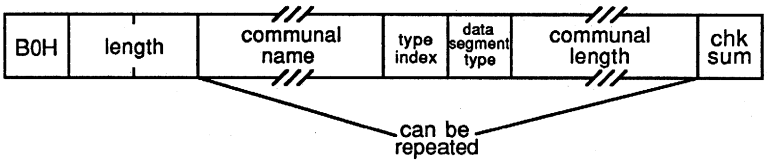
```
         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000    A2 1B 00 01 00 00 0A 00 02 00 01 00 00 00 05 41   ...............A
0010    4C 50 48 41 01 00 00 00 04 42 45 54 41 A9          LPHA.....BETA.
```

- Byte 00H contains 0A2H, identifying this as an LIDATA record.
- Bytes 01–02H contain 1BH, the length of the remainder of the record.
- Byte 03H (the *segment index*) contains 01H, a reference to the first SEGDEF record in this object module, indicating that the data declared in this LIDATA record is to be placed into the segment described by the first SEGDEF record.
- Bytes 04–05H (the *iterated data offset*) contain 0000H, so the data in this LIDATA record is to be located at offset 0000H in the segment designated by the *segment index.*
- Bytes 06–1CH represent an *iterated data block*:
  - Bytes 06–07H contain the *repeat count*, 000AH, which indicates that the *content* field of this *iterated data block* is to be repeated 10 times.
  - Bytes 08–09H (the *block count* for this *iterated data block*) contain 0002H, which indicates that the *content* field of this *iterated data block* (bytes 0A–1CH) contains two nested *iterated data block* fields (bytes 0A–13H and bytes 14–1CH).
  - Bytes 0A–0BH contain 0001H, the *repeat count* for the first nested *iterated data block.* Bytes 0C–0DH contain 0000H, indicating that the *content* field of this nested *iterated data block* contains data, rather than more nested *iterated data blocks.* The *content* field (bytes 0E–13H) contains the data: Byte 0EH contains 05H, the number of subsequent data bytes, and bytes 0F–13H contain the actual data (the string *ALPHA*).
  - Bytes 14–1CH represent the second nested *iterated data block*, which has a format similar to that of the block in bytes 0A–13H. This second nested *iterated data block* represents the 4-byte string *BETA*.
- Byte 1DH is the checksum, 0A9H.

0B0H COMDEF Communal Names Definition Record

The COMDEF record is a Microsoft extension to the basic set of 8086 object record types defined by Intel that declares a list of one or more communal variables. The COMDEF record is recognized by versions 3.50 and later of LINK. Microsoft encourages the use of the COMDEF record for declaration of communal variables.

Record format



Communal name

The *communal name* field is a variable-length field that contains the name of a communal variable. The first byte of this field indicates the length of the name contained in the remainder of the field.

Type index

The *type index* field is an index field that references a previous TYPDEF record in the object module. A value of 1 indicates the first TYPDEF record in the module, a value of 2 indicates the second, and so on. The *type index* value can be 0 if no data type is associated with the public name.

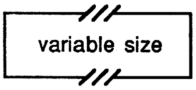
Data segment type

The *data segment type* field is a single byte that indicates whether the communal variable is FAR or NEAR. There are only two possible values for *data segment type*:

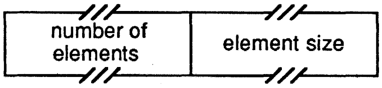
61H	FAR variable
62H	NEAR variable

Communal length

The *communal length* is a variable-length field that indicates the amount of memory to be allocated for the communal variable. The contents of this field depend on the value in the *data segment type* field. If the *data segment type* is NEAR (62H), the *communal length* field contains the size (in bytes) of the communal variable:



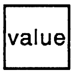
If the *data segment type* is FAR (61H), the *communal length* field is formatted as follows:



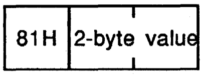
A FAR communal variable is viewed as an array of elements of a specified size. Thus, the *number of elements* field is a variable-length field representing the number of elements in the array, and the *element size* field is a variable-length field that indicates the size (in bytes) of each element. The amount of memory required for a FAR communal variable is thus the product of the *number of elements* and the *element size*.

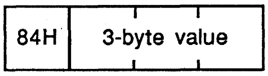
The format of the *variable size*, *number of elements*, and *element size* fields depends upon the magnitude of the values they contain:

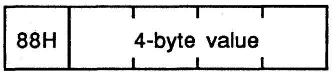
- If the value is less than 128 (80H), the field is formatted as a 1-byte field containing the actual value:



- If the value is 128 (80H) or greater, the field is formatted with an extra initial byte that indicates whether the value is represented in the subsequent 2, 3, or 4 bytes:







Groups of *communal name*, *type index*, *data segment type*, and *communal length* fields can be repeated so that more than one communal variable can be declared in the same COMDEF record.

Location in object module

Any object module that contains COMDEF records must also contain one COMENT record with the *comment class* 0A1H, indicating that Microsoft extensions to the Intel object record specification are included in the object module. This COMENT record must appear before any COMDEF records in the object module.

Example

The following COMDEF record was generated by the Microsoft C Compiler version 4.0 for these public variable declarations:

```
int     foo;                    /* 2-byte integer */
char    foo2[32768];            /* 32768-byte array */
char    far foo3[10][2][20];    /* 400-byte array */
```

The COMDEF record is

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  B0 20 00 04 5F 66 6F 6F 00 62 02 05 5F 66 6F 6F   . .._foo.b.._foo
0010  32 00 62 81 00 80 05 5F 66 6F 6F 33 00 61 81 90   2.b...._foo3.a..
0020  01 01 99                                          ...
```

- Byte 00H contains 0B0H, indicating that this is a COMDEF record.
- Bytes 01–02H contain 0020H, the length of the remainder of the record.
- Bytes 03–0AH, 0B–15H, and 16–21H represent three declarations for the communal variables *foo*, *foo2*, and *foo3*. The C compiler prepends an underscore to each of the names declared in the source code, so the symbols represented in this COMDEF record are *_foo*, *_foo2*, and *_foo3*.
  - Byte 03H contains 04H, the length of the first *communal name* in this record. Bytes 04–07H contain the name itself (*_foo*). Byte 08H (the *type index* field) contains 00H, as required. Byte 09H (the *data segment type* field) contains 62H, indicating this is a NEAR variable. Byte 0AH (the *communal length* field) contains 02H, the size of the variable in bytes.
  - Byte 0BH contains 05H, the length of the second *communal name*. Bytes 0C–10H contain the name, *_foo2*. Byte 11H is the *type index* field, which again contains 00H as required. Byte 12H (the *data segment type* field) contains 62H, indicating that *_foo2* is a NEAR variable.

    Bytes 13–15H (the *communal length* field) contain the size in bytes of the variable. The first byte of the *communal length* field (byte 13H) is 81H, indicating that the size is represented in the subsequent 2 bytes of data — bytes 14–15H, which contain the value 8000H.
  - Bytes 16–1BH represent the *communal name* field for *_foo3*, the third communal variable declared in this record. Byte 1CH (the *type index* field) again contains 00H as required. Byte 1DH (the *data segment type* field) contains 61H, indicating this is a FAR variable. This means the *communal length* field is formatted as a *number of elements* field (bytes 1E–20H, which contain the value 0190H) and an *element size* field (byte 21H, which contains 01H). The total size of this communal variable is thus 190H times 1, or 400 bytes.
- Byte 22H contains the checksum, 99H.

Richard Wilton

Article 20
The Microsoft Object Linker

MS-DOS object modules can be processed in two ways: They can be grouped together in object libraries, or they can be linked into executable files. All Microsoft language translators are distributed with two utility programs that process object modules: The Microsoft Library Manager (LIB) creates and modifies object libraries; the Microsoft Object Linker (LINK) processes the individual object records within object modules to create executable files.

The following discussion focuses on LINK because of its crucial role in creating an executable file. Before delving into the complexities of LINK, however, it is worthwhile reviewing how object modules are managed.

Object Files, Object Libraries, and LIB

Compilers and assemblers translate source-code modules into object modules (Figure 20-1). *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules. An object module consists of a sequence of object records that describe the form and content of part of an executable program. An MS-DOS object module always starts with a THEADR record; subsequent object records in the module follow the sequence discussed in the Object Modules article.

Object modules can be stored in either of two types of MS-DOS files: object files and object libraries. By convention, object files have the filename extension .OBJ and object libraries have the extension .LIB. Although both object files and object libraries contain one or

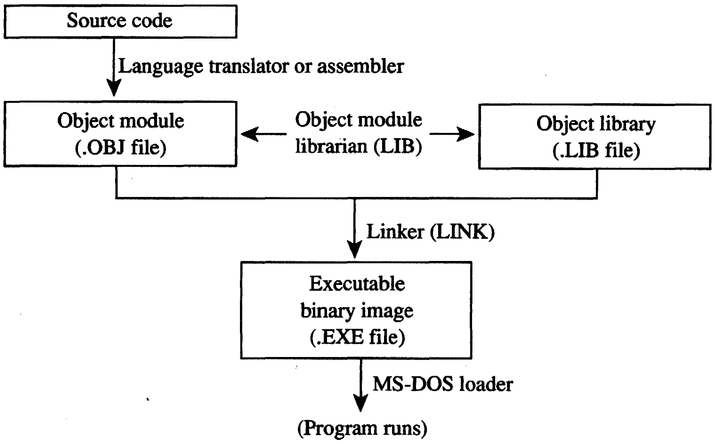


Figure 20-1. Object modules, object libraries, LIB, and LINK.

more object modules, the files and the libraries have different internal organization. Furthermore, LINK processes object files and libraries differently.

The structures of object files and libraries are compared in Figure 20-2. An object file is a simple concatenation of object modules in any arbitrary order. (Microsoft discourages the use of object files that contain more than one object module; Microsoft language translators never generate more than one object module in an object file.) In contrast, a library contains a hashed dictionary of all the public symbols declared in each of the object modules, in addition to the object modules themselves. Each symbol in the dictionary is associated with a reference to the object module in which the symbol was declared.

LINK processes object files differently than it does libraries. When LINK builds an executable file, it incorporates all the object modules in all the object files it processes. In contrast, when LINK processes libraries, it uses the hashed symbol dictionary in each library to extract object modules selectively—it uses an object module from a library only when the object module contains a symbol that is referenced within some other object module. This distinction between object files and libraries is important in understanding what LINK does.

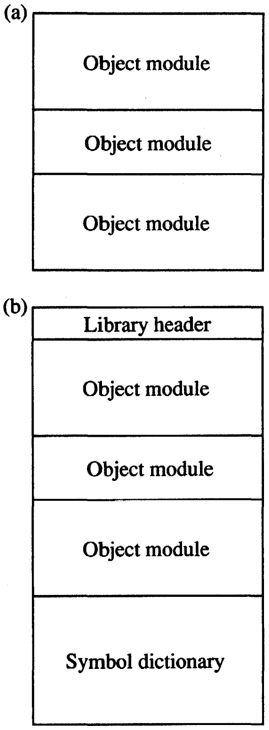


Figure 20-2. Structures of an object file and an object library. (a) An object file contains one or more object modules. (Microsoft discourages using more than one object module per object file.) (b) An object library contains one or more object modules plus a hashed symbol dictionary indicating the object modules in which each public symbol is defined.

What LINK Does

The function of LINK is to translate object modules into an executable program. LINK's input consists of one or more object files (.OBJ files) and, optionally, one or more libraries (.LIB files). LINK's output is an executable file (.EXE file) containing binary data that can be loaded directly from the file into memory and executed. LINK can also generate a symbolic address map listing (.MAP file)—a text file that describes the organization of the .EXE file and the correspondence of symbols declared in the object modules to addresses in the executable file.

Building an executable file

LINK builds two types of information into a .EXE file. First, it extracts executable code and data from the LEDATA and LIDATA records in object modules, arranges them in a specified order according to its rules for segment combination and relocation, and copies the result into the .EXE file. Second, LINK builds a header for the .EXE file. The header describes the size of the executable program and also contains a table of load-time segment relocations and initial values for certain CPU registers. *See* Pass 2 below.

Relocation and linking

In building an executable image from object modules, LINK performs two essential tasks: relocation and linking. As it combines and rearranges the executable code and data it extracts from the object modules it processes, LINK frequently adjusts, or relocates, address references to account for the rearrangements (Figure 20-3). LINK links object modules by resolving address references among them. It does this by matching the symbols declared in EXTDEF and PUBDEF object records (Figure 20-4). LINK uses FIXUPP records to determine exactly how to compute both address relocations and linked address references.

Object Module Order

LINK processes input files from three sources: object files and libraries specified explicitly by the user (in the command line, in response to LINK's prompts, or in a response file) and object libraries named in object module COMENT records.

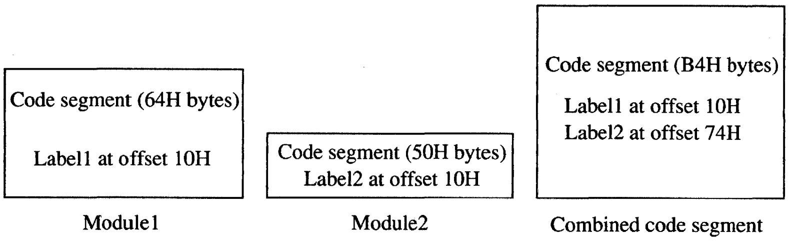


Figure 20-3. A simple relocation. Both object modules contain code that LINK combines into one logical segment. In this example, LINK appends the 50H bytes of code in Module2 *to the 64H bytes of code in* Module1. *LINK relocates all references to addresses in the code segment so that they apply to the combined segment.*

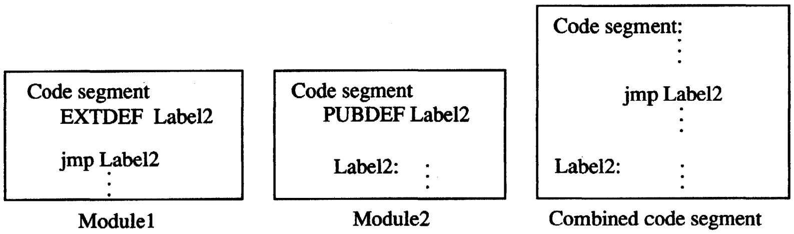


Figure 20-4. Resolving an external reference. LINK resolves the external reference in Module1 *(declared in an EXTDEF record) with the address of* Label2 *in* Module2 *(declared in a PUBDEF record).*

LINK always uses all the object modules in the object files it processes. In contrast, it extracts individual object modules from libraries—only those object modules needed to resolve references to public symbols are used. This difference is implicit in the order in which LINK reads its input files:

1. Object files specified in the command line or in response to the *Object Modules* prompt
2. Libraries specified in the command line or in response to the *Libraries* prompt
3. Libraries specified in COMENT records

The order in which LINK processes object modules influences the resulting executable file in three ways. First, the order in which segments appear in LINK's input files is reflected in the segment structure of the executable file. Second, the order in which LINK resolves external references to public symbols depends on the order in which it finds the public symbols in its input files. Finally, LINK derives the default name of the executable file from the name of the first input object file.

Segment order in the executable file

In general, LINK builds named segments into the executable file in the order in which it first encounters the SEGDEF records that declare the segments. (The /DOSSEG switch also affects segment order. *See* Using the /DOSSEG Switch below.) This means that the order in which segments appear in the executable file can be controlled by linking object modules in a specific order. In assembly-language programs, it is best to declare all the segments used in the program in the first object module to be linked so that the segment order in the executable file is under complete control.

Order in which references are resolved

LINK resolves external references in the order in which it encounters the corresponding public declarations. This fact is important because it determines the order in which LINK extracts object modules from libraries. When a public symbol required to resolve an external reference is declared more than once among the object modules in the input libraries, LINK uses the first object module that contains the public symbol. This means that the actual executable code or data associated with a particular external reference can be varied by changing the order in which LINK processes its input libraries.

For example, imagine that a C programmer has written two versions of a function named *myfunc()* that is called by the program MYPROG.C. One version of *myfunc()* is for debugging; its object module is found in MYFUNC.OBJ. The other is a production version whose object module resides in MYLIB.LIB. Under normal circumstances, the programmer links the production version of *myfunc()* by using MYLIB.LIB (Figure 20-5). To use the debugging version of *myfunc()*, the programmer explicitly includes its object module (MYFUNC.OBJ) when LINK is executed. This causes LINK to build the debugging version of *myfunc()* into the executable file because it encounters the debugging version in MYFUNC.OBJ before it finds the other version in MYLIB.LIB.

To exploit the order in which LINK resolves external references, it is important to know LINK's library search strategy: Each individual library is searched repeatedly (from first library to last, in the sequence in which they are input to LINK) until no further external references can be resolved.

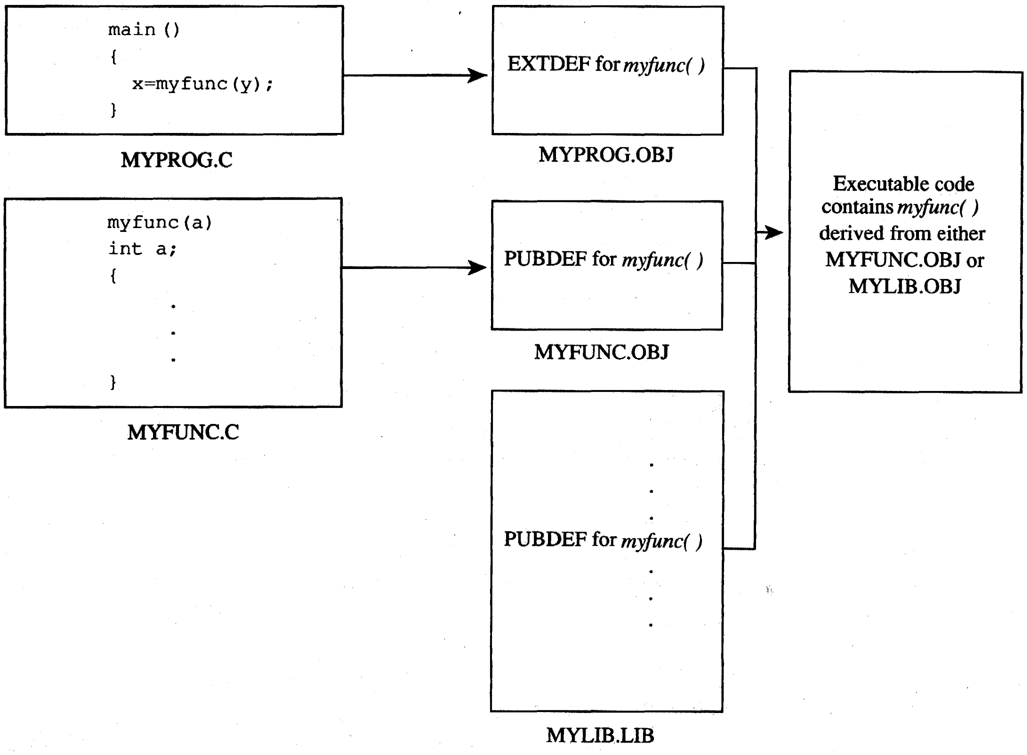


Figure 20-5. Ordered object module processing by LINK. (a) With the command LINK MYPROG,,,MYLIB, *the production version of* myfunc() *in MYLIB.LIB is used. (b) With the command* LINK MYPROG+ MYFUNC,,,MYLIB, *the debugging version of* myfunc() *in MYFUNC.OBJ is used.*

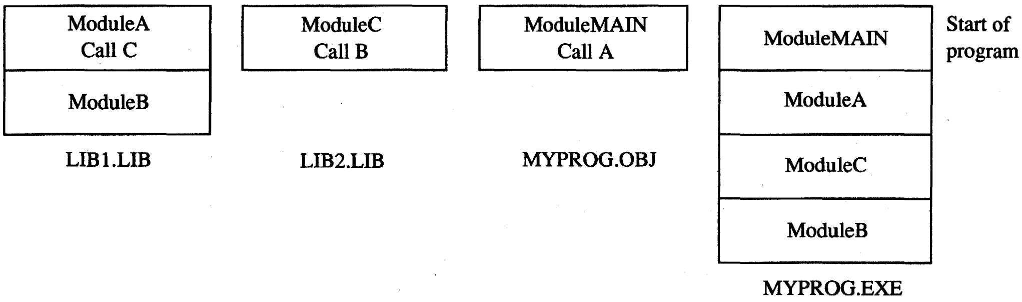


Figure 20-6. Library search order. Modules are incorporated into the executable file as LINK extracts them from the libraries to resolve external references.

The example in Figure 20-6 demonstrates this search strategy. Library LIB1.LIB contains object modules *A* and *B*, library LIB2.LIB contains object module *C*, and the object file MYPROG.OBJ contains the object module *MAIN*; modules *MAIN*, *A*, and *C* each contain an external reference to a symbol declared in another module. When this program is linked with

```
C>LINK MYPROG,,,LIB1+LIB2  <Enter>
```

LINK starts by incorporating the object module *MAIN* into the executable program. It then searches the input libraries until it resolves all the external references:

1. Process MYPROG.OBJ, find unresolved external reference to *A*.
2. Search LIB1.LIB, extract *A*, find unresolved external reference to *C*.
3. Search LIB1.LIB again; reference to *C* remains unresolved.
4. Search LIB2.LIB, extract *C*, find unresolved external reference to *B*.
5. Search LIB2.LIB again; reference to *B* remains unresolved.
6. Search LIB1.LIB again, extract *B*.
7. No more unresolved external references, so end library search.

The order in which the modules appear in the executable file thus reflects the order in which LINK resolves the external references; this, in turn, depends on which modules were contained in the libraries and on the order in which the libraries are input to LINK.

Name of the executable file

If no filename is specified in the command line or in response to the *Run File* prompt, LINK derives the name of the executable file from the name of the first object file it processes. For example, if the object files PROG1.OBJ and PROG2.OBJ are linked with the command

```
C>LINK PROG1+PROG2;  <Enter>
```

the resulting executable file, PROG1.EXE, takes its name from the first object file processed by LINK.

Segment Order and Segment Combinations

LINK builds segments into the executable file by applying the following sequence of rules:

1. Segments appear in the executable file in the order in which their SEGDEF declarations first appear in the input object modules.
2. Segments in different object modules are combined if they have the same name and class and a *public*, *memory*, *stack*, or *common* combine type. All address references within the combined segments are relocated relative to the start of the combined segment.
   - Segments with the same name and either the *public* or the *memory* combine type are combined in the order in which they are processed by LINK. The size of the resulting segment equals the total size of the combined segments.
   - Segments with the same name and the *stack* combine type are overlapped so that the data in each of the overlapped segments ends at the same address. The size of the resulting segment equals the total size of the combined segments. The resulting segment is always paragraph aligned.
   - Segments with the same name and the *common* combine type are overlapped so that the data in each of the overlapped segments starts at the same address. The size of the resulting segment equals the size of the largest of the overlapped segments.
3. Segments with the same class name are concatenated.
4. If the /DOSSEG switch is used, the segments are rearranged in conjunction with DGROUP. *See* Using the /DOSSEG Switch below.

These rules allow the programmer to control the organization of segments in the executable file by ordering SEGMENT declarations in an assembly-language source module, which produces the same order of SEGDEF records in the corresponding object module, and by placing this object module first in the order in which LINK processes its input files.

A typical MS-DOS program is constructed by declaring all executable code and data segments with the *public* combine type, thus enabling the programmer to compile the program's source code from separate source-code modules into separate object modules. When these object modules are linked, LINK combines the segments from the object modules according to the above rules to create logically unified code and data segments in the executable file.

Segment classes

LINK concatenates segments with the same class name after it combines segments with the same segment name and class. For example, Figure 20-7 shows the following compiling and linking:

```
C>MASM MYPROG1;   <Enter>
C>MASM MYPROG2;   <Enter>
C>LINK MYPROG1+MYPROG2;   <Enter>
```

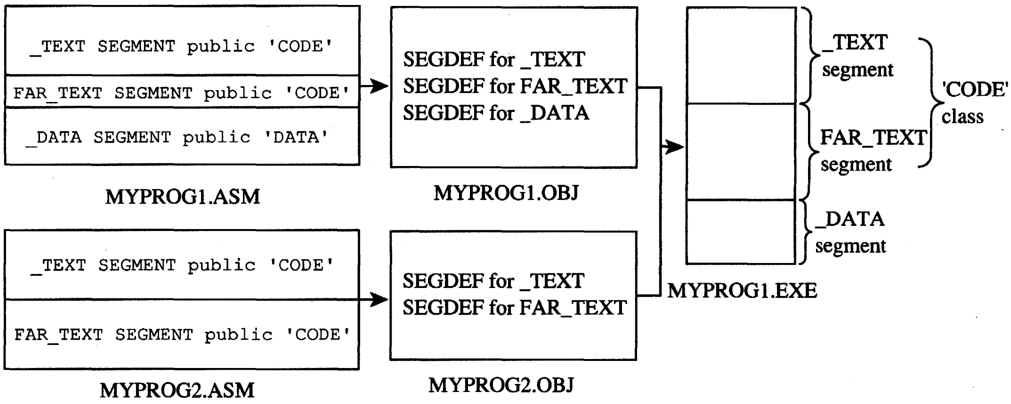


Figure 20-7. Segment order and concatenation by LINK. The start of each file, corresponding to the lowest address, is at the top.

After MYPROG1.ASM and MYPROG2.ASM have been compiled, LINK builds the *_TEXT* and *FAR_TEXT* segments by combining segments with the same name from the different object modules. Then, *_TEXT* and *FAR_TEXT* are concatenated because they have the same class name ('CODE'). *_TEXT* appears before *FAR_TEXT* in the executable file because LINK encounters the SEGDEF record for *_TEXT* before it finds the SEGDEF record for *FAR_TEXT*.

Segment alignment

LINK aligns the starting address of each segment it processes according to the alignment specified in each SEGDEF record. It adjusts the alignment of each segment it encounters regardless of how that segment is combined with other segments of the same name or class. (The one exception is *stack* segments, which always start on a paragraph boundary.)

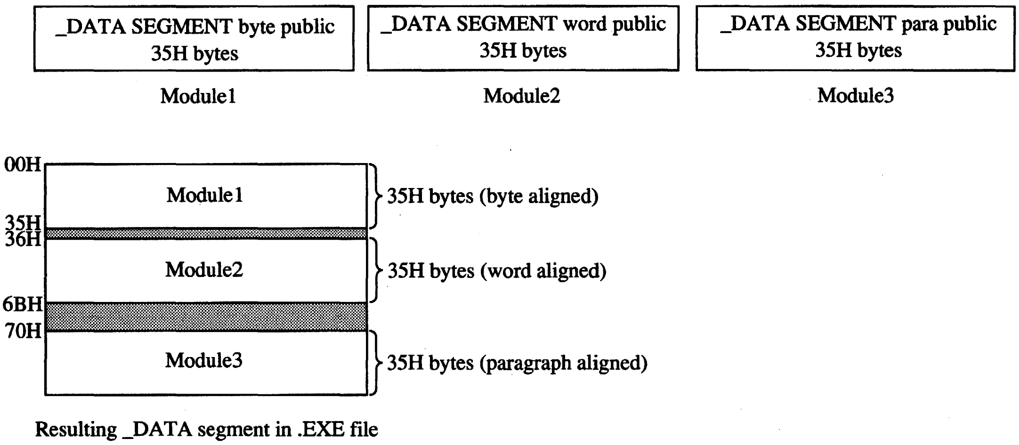

_DATA SEGMENT byte public
35H bytes
Module1
_DATA SEGMENT word public
35H bytes
Module2
_DATA SEGMENT para public
35H bytes
Module3
00H
35H
36H
6BH
70H
Module1
Module2
Module3
35H bytes (byte aligned)
35H bytes (word aligned)
35H bytes (paragraph aligned)
Resulting _DATA segment in .EXE file

Figure 20-8. Alignment of combined segments. LINK enforces segment alignment by padding combined segments with uninitialized data bytes.

Segment alignment is particularly important when public segments with the same name and class are combined from different object modules. Note what happens in Figure 20-8, where the three concatenated *_DATA* segments have different alignments. To enforce the word alignment and paragraph alignment of the *_DATA* segments in *Module2* and *Module3*, LINK inserts one or more bytes of padding between the segments.

Segment groups

A segment group establishes a logical segment address to which all offsets in a group of segments can refer. That is, all addresses in all segments in the group can be expressed as offsets relative to the segment value associated with the group (Figure 20-9). Declaring segments in a group does not affect their positions in the executable file; the segments in a group may or may not be contiguous and can appear in any order as long as all address references to the group fall within 64 KB of each other.

```
DataGroup       GROUP      DataSeg1,DataSeg2
CodeSeg         SEGMENT    byte public 'CODE'
                ASSUME     cs:CodeSeg

                mov        ax,offset DataSeg2:TestData
                mov        ax,offset DataGroup:TestData

CodeSeg         ENDS

DataSeg1        SEGMENT    para public 'DATA'
                DB         100h dup(?)
DataSeg1        ENDS

DataSeg2        SEGMENT    para public 'DATA'
TestData        DB         ?
DataSeg2        ENDS
                END
```

Figure 20-9. Example of group addressing. The first MOV loads the value 00H into AX (the offset of TestData *relative to* DataSeg2*); the second MOV loads the value 100H into AX (the offset of* TestData *relative to the group* DataGroup*).*

LINK reserves one group name, DGROUP, for use by Microsoft language translators. DGROUP is used to group compiler-generated data segments and a default stack segment. *See* DGROUP below.

LINK Internals

Many programmers use LINK as a "black box" program that transforms object modules into executable files. Nevertheless, it is helpful to observe how LINK processes object records to accomplish this task.

LINK is a two-pass linker; that is, it reads all its input object modules twice. On Pass 1, LINK builds an address map of the segments and symbols in the object modules. On Pass 2, it extracts the executable code and program data from the object modules and builds a memory image — an exact replica — of the executable file.

The reason LINK builds an image of the executable file in memory, instead of simply copying code and data from object modules into the executable file, is that it organizes the executable file by segments and not by the order in which it processes object modules. The most efficient way to concatenate, combine, and relocate the code and data is to build a map of the executable file in memory during Pass 1 and then fill in the map with code and data during Pass 2.

In versions 3.52 and later, whenever the /I (/INFORMATION) switch is specified in the command line, LINK displays status messages at the start of each pass and as it processes each object module. If the /M (/MAP) switch is used in addition to the /I switch, LINK also displays the total length of each segment declared in the object modules. This information is helpful in determining how the structure of an executable file corresponds to the contents of the object modules processed by LINK.

Pass 1

During Pass 1, LINK processes the LNAMES, SEGDEF, GRPDEF, COMDEF, EXTDEF, and PUBDEF records in each input object module and uses the information in these object records to construct a symbol table and an address map of segments and segment groups.

Symbol table

As each object module is processed, LINK uses the symbol table to resolve external references (declared in EXTDEF and COMDEF records) to public symbols. If LINK processes all the object files without resolving all the external references in the symbol table, it searches the input libraries for public symbols that match the unresolved external references. LINK continues to search each library until all the external references in the symbol table are resolved.

Segments and groups

LINK processes each SEGDEF record according to the segment name, class name, and attributes specified in the record. LINK constructs a table of named segments and updates it as it concatenates or combines segments. This allows LINK to associate each public symbol in the symbol table with an offset into the segment in which the symbol is declared.

LINK also generates default segments into which it places communal variables declared in COMDEF records. Near communal variables are placed in one paragraph-aligned public segment named c_common, with class name BSS (block storage space) and group

DGROUP. Far communal variables are placed in a paragraph-aligned segment named FAR_BSS, with class name FAR_BSS. The combine type of each far communal variable's FAR_BSS segment is private (that is, not *public*, *memory*, *common*, or *stack*). As many FAR_BSS segments as necessary are generated.

After all the object files have been read and all the external references in the symbol table have been resolved, LINK has a complete map of the addresses of all segments and symbols in the program. If a .MAP file has been requested, LINK creates the file and writes the address map to it. Then LINK initiates Pass 2.

Pass 2

In Pass 2, LINK extracts executable code and program data from the LEDATA and LIDATA records in the object modules. It builds the code and data into a memory image of the executable file. During Pass 2, LINK also carries out all the address relocations and fixups related to segment relocation, segment grouping, and resolution of external references, as well as any other address fixups specified explicitly in object module FIXUPP records.

If it determines during Pass 2 that not enough RAM is available to contain the entire image, LINK creates a temporary file in the current directory on the default disk drive. (LINK versions 3.60 and later use the environment variable TMP to find the directory for the temporary scratch file.) LINK then uses this file in addition to all the available RAM to construct the image of the executable file. (In versions of MS-DOS earlier than 3.0, the temporary file is named VM.TMP; in versions 3.0 and later, LINK uses Interrupt 21H Function 5AH to create the file.)

LINK reads each of the input object modules in the same order as it did in Pass 1. This time it copies the information from each object module's LEDATA and LIDATA records into the memory image of each segment in the proper sequence. This is when LINK expands the iterated data in each LIDATA record it processes.

LINK processes each LEDATA and LIDATA record along with the corresponding FIXUPP record, if one exists. LINK processes the FIXUPP record, performs the address calculations required for relocation, segment grouping, and resolving external references, and then stores binary data from the LEDATA or LIDATA record, including the results of the address calculations, in the proper segment in the memory image. The only exception to this process occurs when a FIXUPP record refers to a segment address. In this case, LINK adds the address of the fixup to a table of segment fixups; this table is used later to generate the segment relocation table in the .EXE header.

When all the data has been extracted from the object modules and all the fixups have been carried out, the memory image is complete. LINK now has all the information it needs to build the .EXE header (Table 20-1). At this point, therefore, LINK creates the executable file and writes the header and all segments into it.

Table 20-1. How LINK Builds a .EXE File Header.

Offset	Contents	Comments
00H	'MZ'	.EXE file signature
02H	Length of executable image MOD 512	Total size of all segments plus .EXE file header
04H	Length of executable image in 512-byte pages, including last partial page (if any)	
06H	Number of run-time segment relocations	Number of segment fixups
08H	Size of the .EXE header in 16-byte paragraphs	Size of segment relocation table
0AH	MINALLOC: Minimum amount of RAM to be allocated above end of the loaded program (in 16-byte paragraphs)	Size of uninitialized data and/or stack segments at end of program (0 if /HI switch is used)
0CH	MAXALLOC: Maximum amount of RAM to be allocated above end of the loaded program (in 16-byte paragraphs)	0 if /HI switch is used; value specified with /CP switch; FFFFH if /CP and /HI switches are not used
0EH	Stack segment (initial value for SS register); relocated by MS-DOS when program is loaded	Address of stack segment relative to start of executable image
10H	Stack pointer (initial value for register SP)	Size of stack segment in bytes
12H	Checksum	One's complement of sum of all words in file, excluding checksum itself
14H	Entry point offset (initial value for register IP)	MODEND object record that specifies program start address
16H	Entry point segment (initial value for register CS); relocated by MS-DOS when program is loaded	
18H	Offset of start of segment relocation table relative to start of .EXE header	
1AH	Overlay number	0 for resident segments; >0 for overlay segments
1CH	Reserved	

Using LINK to Organize Memory

By using LINK to rearrange and combine segments, a programmer can generate an executable file in which segment order and addressing serve specific purposes. As the following examples demonstrate, careful use of LINK leads to more efficient use of memory and simpler, more efficient programs.

Segment order for a TSR

In a terminate-and-stay-resident (TSR) program, LINK must be used carefully to generate segments in the executable file in the proper order. A typical TSR program consists of a resident portion, in which the TSR application is implemented, and a transient portion, which executes only once to initialize the resident portion. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities.

Because the transient portion of the TSR program is executed only once, the memory it occupies should be freed after the resident portion has been initialized. To allow the MS-DOS Terminate and Stay Resident function (Interrupt 21H Function 31H) to free this memory when it leaves the resident portion of the TSR program in memory, the TSR program must have its resident portion at lower addresses than its transient portion.

```
Low Memory   ResidentCodeSeg     SEGMENT    para         \
                                 .                        |
                                 .(executable code)       |
                                 .                        |
             ResidentCodeSeg     ENDS                     |
             ResidentDataSeg     SEGMENT    word          |
                                 .                        |
                                 .(program data)          >  Resident
                                 .                        |  portion
             ResidentDataSeg     ENDS                     |
             StackSeg            SEGMENT    para          |
                                 .                        |
                                 .(stack)                 |
                                 .                        |
             StackSeg            ENDS                     /
             TransientCodeSeg    SEGMENT    para         \
                                 .                        |
                                 .(executable code)       |
                                 .                        |
             TransientCodeSeg    ENDS                     >  Transient
             TransientDataSeg    SEGMENT    word          |  portion
                                 .                        |
                                 .(program data)          |
                                 .                        |
High Memory  TransientDataSeg    ENDS                     /
```

Figure 20-10. Segment order for a terminate-and-stay-resident program.

In Figure 20-10, the segments containing the resident code and data are declared before the segments that represent the transient portion of the program. Because LINK preserves this segment order, the executable program has the desired structure, with resident code and data at lower addresses than transient code and data. Moreover, the number of paragraphs in the resident portion of the program, which must be computed before Interrupt 21H Function 31H is called, is easy to derive from the segment structure: This value is the difference between the segment address of the program segment prefix, which immediately precedes the first segment in the resident portion, and the address of the first segment in the transient portion of the program.

Groups for unified segment addressing

In some programs it is desirable to maintain executable code and data in separate logical segments but to address both code and data with the same segment register. For example, in a hardware interrupt handler, using the CS register to address program data is generally simpler than using DS or ES.

In the routine in Figure 20-11, code and data are maintained in separate segments for program clarity, yet both can be addressed using the CS register because both code and data segments are included in the same group. (The SNAP.ASM listing in the Terminate-and-Stay-Resident Utilities article is another example of this use of a group to unify segment addressing.)

```
ISRgroup        GROUP       CodeSeg,DataSeg
CodeSeg         SEGMENT     byte public 'CODE'
                ASSUME      cs:ISRgroup
                mov         ax,offset ISRgroup:CodeLabel
CodeLabel:      mov         bx,ISRgroup:DataLabel
CodeSeg         ENDS

DataSeg         SEGMENT     para public 'DATA'
DataLabel       DW          ?
DataSeg         ENDS
                END
```

Figure 20-11. Code and data included in the same group. In this example, addresses within both CodeSeg *and* DataSeg *are referenced relative to the CS register by grouping the segments (using the assembler GROUP directive) and addressing the group through CS (using the assembler ASSUME directive).*

Uninitialized data segments

A segment that contains only uninitialized data can be processed by LINK in two ways, depending on the position of the segment in the program. If the segment is not at the end of the program, LINK generates a block of bytes initialized to zero to represent the segment in the executable file. If the segment appears at the end of the program, however, LINK does not generate a block of zeroed bytes. Instead, it increases the minimum run-time memory allocation by increasing MINALLOC (specified at offset 0AH in the .EXE header) by the amount of memory required for the segment.

Therefore, if it is necessary to reserve a large amount of uninitialized memory in a segment, the size of the .EXE file can be decreased by building the segment at the end of a program (Figure 20-12). This is why, for example, Microsoft high-level-language translators always build BSS and STACK segments at the end of compiled programs. (The loader does not fill these segments with zeros; a program must still initialize them with appropriate values.)

```
(a)          CodeSeg     SEGMENT     byte public 'CODE'
                         ASSUME      cs:CodeSeg,ds:DataSeg
                         ret
             CodeSeg     ENDS

             DataSeg     SEGMENT     word public 'DATA'
             BigBuffer   DB          10000 dup(?)
             DataSeg     ENDS
                         END

(b)          DataSeg     SEGMENT     word public 'DATA'
             BigBuffer   DB          10000 dup(?)
             DataSeg     ENDS

             CodeSeg     SEGMENT     byte public 'CODE'
                         ASSUME      cs:CodeSeg,ds:DataSeg
                         ret
             CodeSeg     ENDS
                         END
```

Figure 20-12. LINK processing of uninitialized data segments. (a) When DataSeg*, which contains only uninitialized data, is placed at the end of this program, the size of the .EXE file is only 513 bytes. (b) When* DataSeg *is not placed at the end of the program, the size of the .EXE file is 10513 bytes.*

Overlays

If a program contains two or more subroutines that are mutually independent—that is, subroutines that do not transfer control to each other—LINK can be instructed to build each subroutine into a separately loaded portion of the executable file. (This instruction is indicated in the command line when LINK is executed by enclosing each overlay subroutine or group of subroutines in parentheses.) Each of the subroutines can then be overlaid as it is needed in the same area of memory (Figure 20-13). The amount of memory required to run a program that uses overlays is, therefore, less than the amount required to run the same program without overlays.

A program that uses overlays must include the Microsoft run-time overlay manager. The overlay manager is responsible for copying overlay code from the executable file into memory whenever the program attempts to transfer control to code in an overlay. A program that uses overlays runs slower than a program that does not use them, because it takes longer to extract overlays separately from the .EXE file than it does to read the entire .EXE file into memory at once.

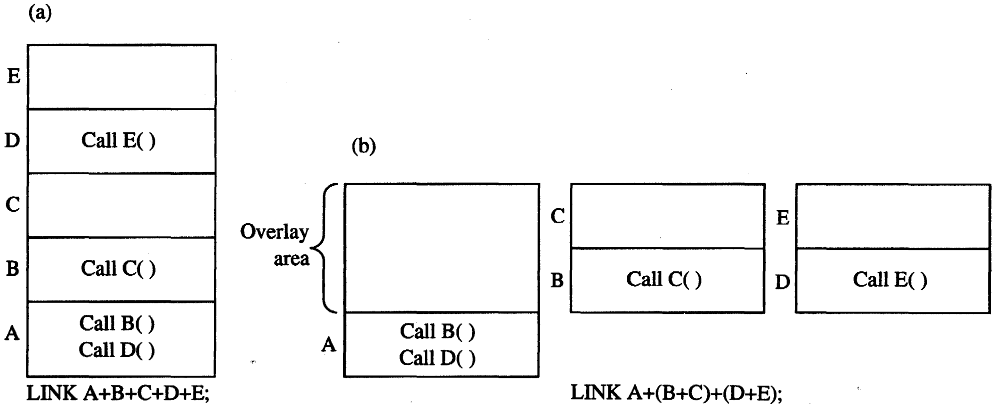


Figure 20-13. Memory use in a program linked (a) without overlays and (b) with overlays. In (b), either modules (B+C) or modules (D+E) can be loaded into the overlay area at run time.

The default object libraries that accompany Microsoft high-level-language compilers contain object modules that support the Microsoft run-time overlay manager. The following description of LINK's relationship to the run-time overlay manager applies to versions 3.00 through 3.60 of LINK; implementation details may vary in future versions.

Overlay format in a .EXE file

An executable file that contains overlays has a .EXE header preceding each overlay (Figure 20-14). The overlays are numbered in sequence, starting at 0; the overlay number is stored in the word at offset 1AH in each overlay's .EXE header. When the contents of the .EXE file are loaded into memory for execution, only the resident, nonoverlaid part of the program is copied into memory. The overlays must be read into memory from the .EXE file by the run-time overlay manager.

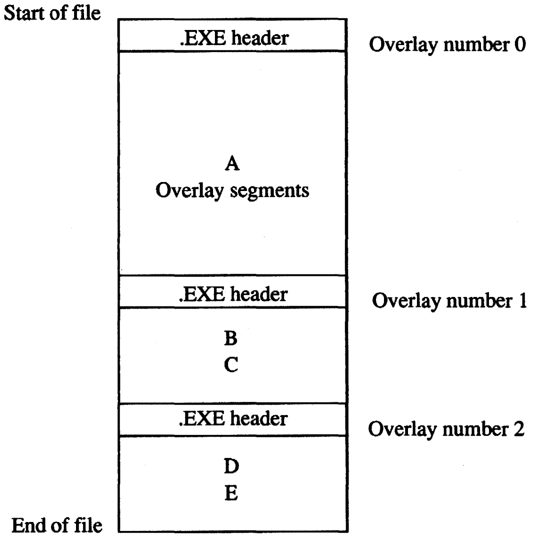


Figure 20-14. .EXE file structure produced by LINK A + (B+C) + (D+E).

Segments for overlays

When LINK produces an executable file that contains overlays, it adds three segments to those defined in the object modules: OVERLAY_AREA, OVERLAY_END, and OVERLAY_DATA. LINK assigns the segment class name 'CODE' to OVERLAY_AREA and OVERLAY_END and includes OVERLAY_DATA in the default group DGROUP.

OVERLAY_AREA is a reserved segment into which the run-time overlay manager is expected to load each overlay as it is needed. Therefore, LINK sets the size of OVERLAY_AREA to fit the largest overlay in the program. The OVERLAY_END segment is declared immediately after OVERLAY_AREA, so a program can determine the size of the OVERLAY_AREA segment by subtracting its segment address from that of OVERLAY_END. The OVERLAY_DATA segment is initialized by LINK with information about the executable file, the number of overlays, and other data useful to the run-time overlay manager.

LINK requires the executable code used in overlays to be contained in segments whose class names end in *CODE* and whose segment names differ from those of the segments used in the resident (nonoverlaid) portion of the program. In assembly language, this is accomplished by using the SEGMENT directive; in high-level languages, the technique of ensuring unique segment names depends on the compiler. In Microsoft C, for example, the /A switch in the command line selects the memory model and thus the segment naming defaults used by the compiler; in medium, large, and huge memory models, the compiler generates a unique segment name for each C function in the source code. In Microsoft FORTRAN, on the other hand, the compiler always generates a uniquely named segment for each SUBROUTINE and FUNCTION in the source code, so no special programming is required.

LINK substitutes all far CALL instructions from root to overlay or from overlay to overlay with a software interrupt followed by an overlay number and an offset into the overlay segment (Figure 20-15). The interrupt number can be specified with LINK's /OVERLAYINTERRUPT switch; if the switch is omitted, LINK uses Interrupt 3FH by default. By replacing calls to overlay code with a software interrupt, LINK provides a mechanism for the run-time overlay manager to take control, load a specified overlay into memory, and transfer control to a specified offset within the overlay.

```
(a)           EXTRN    OverlayEntryPoint:far
              call     OverlayEntryPoint   ; far CALL

(b)           int      IntNo               ; interrupt number
                                           ;  specified with /OVERLAYINTERRUPT
                                           ;  switch (default 3FH)
              DB       OverlayNumber       ; overlay number
              DW       OverlayEntry        ; offset of overlay entry point
                                           ;  (the address to which
                                           ;  the overlay manager transfers
                                           ;  control)
```

Figure 20-15. Executable code modification by LINK for accessing overlays. (a) Code as written. (b) Code as modified by LINK.

Run-time processing of overlays

The resident (nonoverlaid) portion of a program that uses overlays initializes the overlay interrupt vector specified by LINK with the address of the run-time overlay manager. (The OVERLAY_DATA segment contains the interrupt number.) The overlay manager then takes control wherever LINK has substituted a software interrupt for a far call in the executable code.

Each time the overlay manager executes, its first task is to determine which overlay is being called. It does this by using the return address left on the stack by the INT instruction that invoked the overlay manager; this address points to the overlay number stored in the byte after the interrupt instruction that just executed. The overlay manager then determines whether the destination overlay is already resident and loads it only if necessary. Next, the overlay manager opens the .EXE file, using the filename in the OVERLAY_DATA segment. It locates the start of the specified overlay in the file by examining the length (offset 02H and offset 04H) and overlay number (offset 1AH) in each overlay's .EXE header.

The overlay manager can then read the overlay from the .EXE file into the OVERLAY_AREA segment. It uses the overlay's segment relocation table to fix up any segment references in the overlay. The overlay manager transfers control to the overlay with a far call to the OVERLAY_AREA segment, using the offset stored by LINK 1 byte after the interrupt instruction (*see* Figure 20-15).

Interrupt 21H Function 4BH

LINK's protocol for implementing overlays is not recognized by Interrupt 21H Function 4BH (Load and Execute Program). This MS-DOS function, when called with AL = 03H, loads an overlay from a .EXE file into a specified location in memory. *See* SYSTEM CALLS: INTERRUPT 21H: Function 4BH. However, Function 4BH does not use an overlay number, so it cannot find overlays in a .EXE file formatted by LINK with multiple .EXE headers.

DGROUP

LINK always includes DGROUP in its internal table of segment groups. In object modules generated by Microsoft high-level-language translators, DGROUP contains both the default data segment and the stack segment. LINK's /DOSSEG and /DSALLOCATE switches both affect the way LINK treats DGROUP. Changing the way LINK manages DGROUP ultimately affects segment order and addressing in the executable file.

Using the /DOSSEG switch

The /DOSSEG switch causes LINK to arrange segments in the default order used by Microsoft high-level-language translators:

1. All segments with a class name ending in *CODE*. These segments contain executable code.
2. All other segments outside DGROUP. These segments typically contain far data items.

3. DGROUP segments. These are a program's near data and stack segments. The order in which segments appear in DGROUP is
   - Any segments of class BEGDATA. (This class name is reserved for Microsoft use.)
   - Any segments not of class BEGDATA, BSS, or STACK.
   - Segments of class BSS.
   - Segments of class STACK.

This segment order is necessary if programs compiled by Microsoft translators are to run properly. The /DOSSEG switch can be used whenever an object module produced by an assembler is linked ahead of object modules generated by a Microsoft compiler, to ensure that segments in the executable file are ordered as in the preceding list regardless of the order of segments in the assembled object module.

When the /DOSSEG switch is in effect, LINK always places DGROUP at the end of the executable program, with all uninitalized data segments at the end of the group. As discussed above, this placement helps to minimize the size of the executable file. The /DOSSEG switch also causes LINK to restructure the executable program to support certain conventions used by Microsoft language translators:

- Compiler-generated segments with the class name BEGDATA are placed at the beginning of DGROUP.
- The public symbols _edata and _end are generated to point to the beginning of the BSS and STACK segments.
- Sixteen bytes of zero are inserted in front of the _TEXT segment.

Microsoft compilers that rely on /DOSSEG conventions generate a special COMENT object record that sets the /DOSSEG switch when the record is processed by LINK.

Using the /HIGH and /DSALLOCATE switches

When a program has been linked without using LINK's /HIGH switch, MS-DOS loads program code and data segments from the .EXE file at the lowest address in the first available block of RAM large enough to contain the program (Figure 20-16). The value in the .EXE header at offset 0CH specifies the maximum amount of extra RAM MS-DOS must allocate to the program above what is loaded from the .EXE file. Above that, all unused RAM is managed by MS-DOS. With this memory allocation strategy, a program can use Interrupt 21H Functions 48H (Allocate Memory Block) and 4AH (Resize Memory Block) to increase or decrease the amount of RAM allocated to it.

When a program is linked with LINK's /HIGH switch, LINK zeros the words it stores in the .EXE header at offset 0AH and 0CH. Setting the words at 0AH and 0CH to zero indicates that the program is to be loaded into RAM at the highest address possible (Figure 20-16). With this memory layout, however, a program can no longer change its memory allocation dynamically because all available RAM is allocated to the program when it is loaded and the uninitialized RAM between the program segment prefix and the program itself cannot be freed.

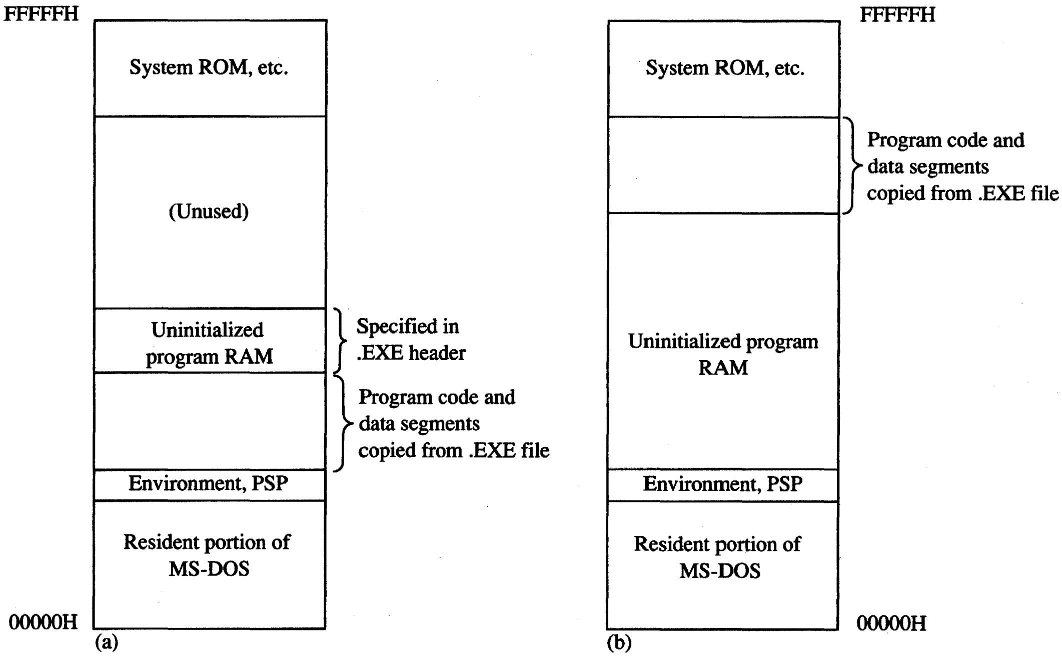

FFFFFH
System ROM, etc.
(Unused)
Uninitialized program RAM
Specified in .EXE header
Program code and data segments copied from .EXE file
Environment, PSP
Resident portion of MS-DOS
00000H
(a)
FFFFFH
System ROM, etc.
Program code and data segments copied from .EXE file
Uninitialized program RAM
Environment, PSP
Resident portion of MS-DOS
00000H
(b)

Figure 20-16. Effect of the /HIGH switch on run-time memory use. (a) The program is linked without *the /HIGH switch. (b) The program is linked* with *the /HIGH switch.*

The only reason to load a program with this type of memory allocation is to allow a program data structure to be dynamically extended toward lower memory addresses. For example, both stacks and heaps can be implemented in this way. If a program's stack segment is the first segment in its memory map, the stack can grow downward without colliding with other program data.

To facilitate addressing in such a segment, LINK provides the /DSALLOCATE switch. When a program is linked using this switch, all addresses within DGROUP are relocated in such a way that the last byte in the group has offset FFFFH. For example, if the program in Figure 20-17 is linked without the /DSALLOCATE and /HIGH switches, the value of offset *DGROUP:DataItem* would be 00H; if these switches are used, the linker adjusts the segment value of DGROUP downward so that the offset of *DataItem* within DGROUP becomes FFF0H.

Early versions of Microsoft Pascal (before version 3.30) and Microsoft FORTRAN (before version 3.30) generated object code that had to be linked with the /DSALLOCATE switch. For this reason, LINK sets the /DSALLOCATE switch by default if it processes an object module containing a COMENT record generated by one of these compilers. (Such a COMENT record contains the string *MS PASCAL* or *FORTRAN 77. See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules.) Apart from this special requirement of certain language translators, however, the use of /DSALLOCATE and /HIGH should probably be avoided because of the limitations they place on run-time memory allocation.

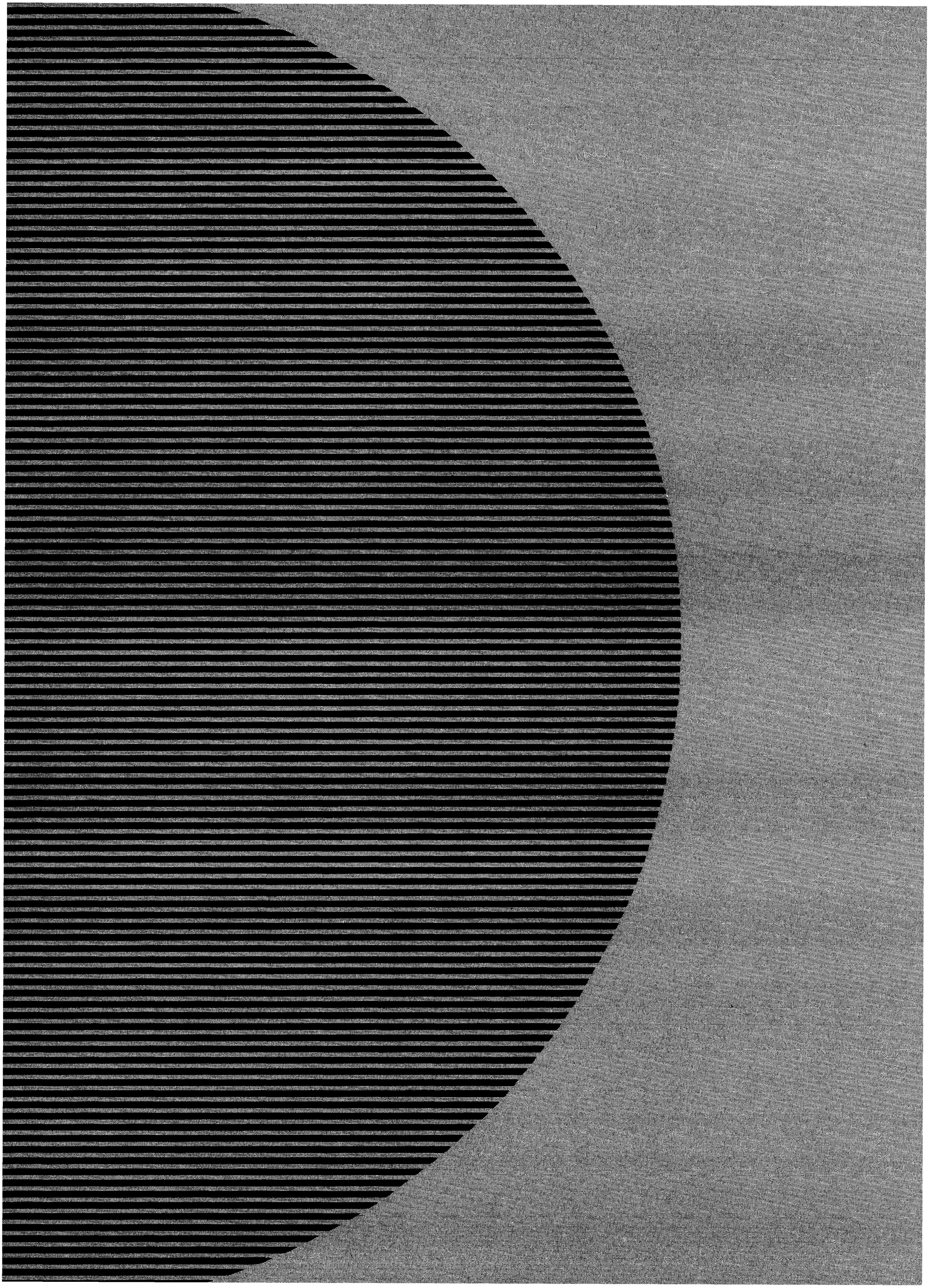
```
DGROUP          GROUP      _DATA
_DATA           SEGMENT    word public 'DATA'
DataItem        DB         10h dup (?)
_DATA           ENDS

_TEXT           SEGMENT    byte public 'CODE'
                ASSUME     cs:_TEXT,ds:DGROUP
                mov        bx,offset DGROUP:DataItem
_TEXT           ENDS
                END
```

Figure 20-17. The value of offset DGROUP:DataItem *in this program is FFF0H if the program is linked with the /DSALLOCATE switch or 00H if the program is linked without using the switch.*

Summary

LINK's characteristic support for segment ordering, for run-time memory management, and for dynamic overlays has an impact in many different situations. Programmers who write their own language translators must bear in mind the special conventions followed by LINK in support of Microsoft language translators. Application programmers must be familiar with LINK's capabilities when they use assembly language or link assembly-language programs with object modules generated by Microsoft compilers. LINK is a powerful program development tool and understanding its special capabilities can lead to more efficient programs.

Richard Wilton



Section III
User Commands

Introduction

This section of *The MS-DOS Encyclopedia* describes the standard internal and external MS-DOS commands available to the user who is running MS-DOS (versions 1.0 through 3.2). System configuration options, special batch-file directives, the line editor (EDLIN), and the installable device drivers normally included with MS-DOS are also covered.

Entries are arranged alphabetically by the name of the command or driver. The configuration, batch-file, and line-editor directives appear alphabetically under the headings CONFIG.SYS, BATCH, and EDLIN, respectively. Each entry includes

- Command name
- Version dependencies and network information
- Command purpose
- Prototype command and summary of options
- Detailed description of command
- One or more examples of command use
- Return codes (where applicable)
- Informational and error messages

The experienced user can find information with a quick glance at the first part of a command entry; a less experienced user can refer to the detailed explanation and examples in a more leisurely fashion. The next two pages contain an example of a typical entry from the User Commands section, with explanations of each component. This example is followed by listings of the commands by functional group.

The following terms are used for command-line variables in the sample syntax:

drive — a letter in the range A–Z, followed by a colon, indicating a logical disk drive.

path — a specific location in a disk's hierarchical directory structure; can include the special directory names **.** and **..**; elements are separated by backslash characters (\).

pathname — a file specification that can include a path and/or drive and/or filename extension.

filename — the name of a file, generally with its extension; cannot include a drive or path.

Note: PC-DOS, though not an official product name, is used in this section to indicate IBM's version of the disk operating system originally provided by Microsoft. Commands sometimes have slightly different options or appear for the first time in different versions of MS-DOS and PC-DOS. When a command appears only in the IBM versions, the abbreviation IBM appears in the heading area. Significant differences between MS-DOS and PC-DOS versions of a command are indicated in the *Syntax* and *Description* portions of the entry.

HEADING
The command name as the user would enter it or as it would be used in a batch or system-configuration file.

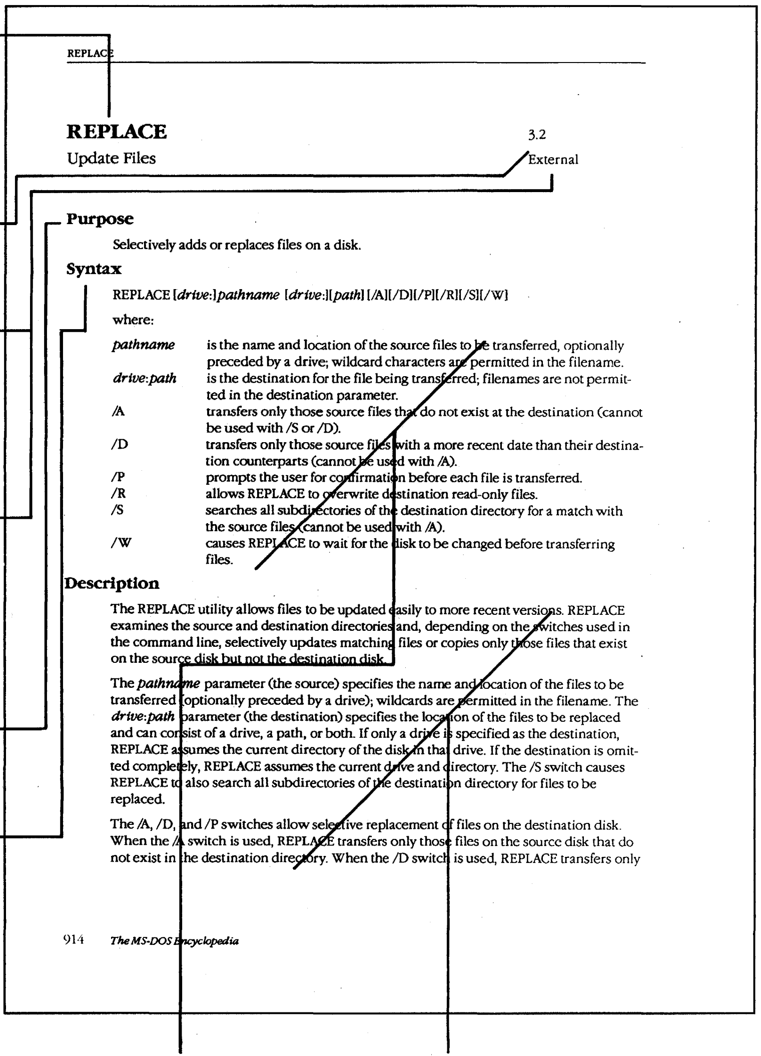
ICON-1
MS-DOS version dependency.

ICON-2
Whether the command is internal (built into COMMAND.COM) or external (loaded from a disk file when needed).

ICON-3
The abbreviation IBM if the command is present only in PC-DOS and the warning *No Net* if the command cannot be used across a network.

PURPOSE
An abstract of command purpose and usage.

SYNTAX
A prototype command line, with variable names in italic and optional parameters in square brackets. The various elements of the command line should be entered in the order shown. Any punctuation must be used exactly as shown; in commands that use commas as separators, the comma usually must be included as a placeholder even if the parameter is omitted. Except where noted, commands, parameters, and switches can be entered in either uppercase or lowercase. With MS-DOS versions 3.0 and later, external commands can be preceded by a drive and/or path.



REPLACE

REPLACE 3.2
Update Files External

Purpose

Selectively adds or replaces files on a disk.

Syntax

REPLACE [*drive*:]*pathname* [*drive*:][*path*] [/A][/D][/P][/R][/S][/W]

where:

pathname is the name and location of the source files to be transferred, optionally preceded by a drive; wildcard characters are permitted in the filename.
drive:path is the destination for the file being transferred; filenames are not permitted in the destination parameter.
/A transfers only those source files that do not exist at the destination (cannot be used with /S or /D).
/D transfers only those source files with a more recent date than their destination counterparts (cannot be used with /A).
/P prompts the user for confirmation before each file is transferred.
/R allows REPLACE to overwrite destination read-only files.
/S searches all subdirectories of the destination directory for a match with the source files (cannot be used with /A).
/W causes REPLACE to wait for the disk to be changed before transferring files.

Description

The REPLACE utility allows files to be updated easily to more recent versions. REPLACE examines the source and destination directories and, depending on the switches used in the command line, selectively updates matching files or copies only those files that exist on the source disk but not the destination disk.

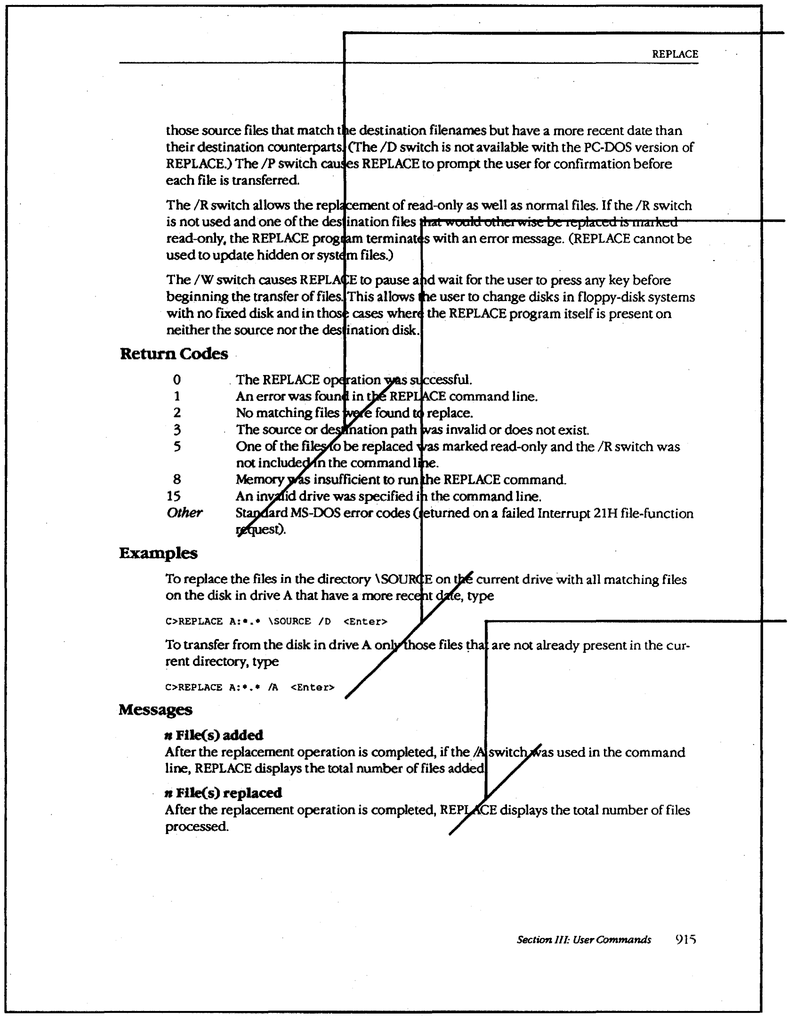
The *pathname* parameter (the source) specifies the name and location of the files to be transferred (optionally preceded by a drive); wildcards are permitted in the filename. The *drive:path* parameter (the destination) specifies the location of the files to be replaced and can consist of a drive, a path, or both. If only a drive is specified as the destination, REPLACE assumes the current directory of the disk in that drive. If the destination is omitted completely, REPLACE assumes the current drive and directory. The /S switch causes REPLACE to also search all subdirectories of the destination directory for files to be replaced.

The /A, /D, and /P switches allow selective replacement of files on the destination disk. When the /A switch is used, REPLACE transfers only those files on the source disk that do not exist in the destination directory. When the /D switch is used, REPLACE transfers only

914 *The MS-DOS Encyclopedia*

BELOW WHERE
A brief explanation of each command parameter and switch. Drives, paths, and filenames are always listed first, followed by the switches in alphabetic order. Any special position required for a filename or switch is shown in the syntax line and noted in the explanation.

DESCRIPTION
A detailed description of the command, including a full explanation of MS-DOS version dependencies, default values, possible interactions of command parameters and options, useful background information, and any applicable warnings.



those source files that match the destination filenames but have a more recent date than their destination counterparts. (The /D switch is not available with the PC-DOS version of REPLACE.) The /P switch causes REPLACE to prompt the user for confirmation before each file is transferred.

The /R switch allows the replacement of read-only as well as normal files. If the /R switch is not used and one of the destination files that would otherwise be replaced is marked read-only, the REPLACE program terminates with an error message. (REPLACE cannot be used to update hidden or system files.)

The /W switch causes REPLACE to pause and wait for the user to press any key before beginning the transfer of files. This allows the user to change disks in floppy-disk systems with no fixed disk and in those cases where the REPLACE program itself is present on neither the source nor the destination disk.

Return Codes

0	The REPLACE operation was successful.
1	An error was found in the REPLACE command line.
2	No matching files were found to replace.
3	The source or destination path was invalid or does not exist.
5	One of the files to be replaced was marked read-only and the /R switch was not included in the command line.
8	Memory was insufficient to run the REPLACE command.
15	An invalid drive was specified in the command line.
Other	Standard MS-DOS error codes (returned on a failed Interrupt 21H file-function request).

Examples

To replace the files in the directory \SOURCE on the current drive with all matching files on the disk in drive A that have a more recent date, type

```
C>REPLACE A:*.* \SOURCE /D  <Enter>
```

To transfer from the disk in drive A only those files that are not already present in the current directory, type

```
C>REPLACE A:*.* /A  <Enter>
```

Messages

n **File(s) added**
After the replacement operation is completed, if the /A switch was used in the command line, REPLACE displays the total number of files added.

n **File(s) replaced**
After the replacement operation is completed, REPLACE displays the total number of files processed.

RETURN CODES
Exit codes returned by the command (if any) that can be tested in a batch file or by another program.

EXAMPLES
One or more examples of the command at work, including examples of the resulting output where appropriate. User entry appears in color; do not type the prompt, which appears in black. Press the Enter key (labeled Return on some keyboards) as directed at the end of each command line.

MESSAGES
An alphabetic list of messages that may be displayed when the command is used in MS-DOS version 3.2 (may vary slightly in earlier versions). Both messages generated by the command itself and applicable messages generated by MS-DOS are included. Following each message is a brief explanation of the condition that produces the message and, where appropriate, any action that should be taken.

Contents by Functional Group

The MS-DOS commands can be divided into several distinct groups according to the functions they perform. These are listed on the following pages.

Command	Action
System Configuration and Control	
BREAK	Set Control-C check.
COMMAND	Install secondary copy of command processor.
DATE	Set date.
EXIT	Terminate command processor.
PROMPT	Define system prompt.
SELECT	Configure system disk for a specific country.
SET	Set environment variable.
SHARE	Install file-sharing support.
TIME	Set system time.
VER	Display version.
Character-Device Management	
CLS	Clear screen.
CTTY	Assign standard input/output.
GRAFTABL	Load graphics character set.
GRAPHICS	Print graphics screen-dump program.
KEYB*xx*	Define keyboard.
MODE	Configure device.
PRINT	Print file (background print spooler).
File Management	
ATTRIB	Change file attributes.
BACKUP	Back up files.
COMP	Compare files.
COPY	Copy file or device.
DEL/ERASE	Delete file.
EDLIN	Create or modify text file (see also commands below).
FC	Compare files.
RECOVER	Recover files.
RENAME	Change filename.
REPLACE	Update files.
RESTORE	Restore backup files.
TYPE	Display file.
XCOPY	Copy files.

(more)

Command	Action
Filters	
FIND	Find string.
MORE	Display by screenful.
SORT	Sort file or character stream alphabetically.
Directory Management	
APPEND	Set data-file search path.
CHDIR	Change current directory.
DIR	Display directory.
MKDIR	Make directory.
PATH	Define command search path.
RMDIR	Remove directory.
TREE	Display directory structure.
Disk Management	
ASSIGN	Assign drive alias.
CHKDSK	Check disk status.
DISKCOMP	Compare floppy disks.
DISKCOPY	Copy floppy disks.
FORMAT	Initialize disk.
FDISK	Configure fixed disk.
JOIN	Join disk to directory.
LABEL	Display volume label.
SUBST	Substitute drive for subdirectory.
SYS	Transfer system files.
VERIFY	Set verify flag.
VOL	Display disk name.
Installable Device Drivers	
ANSI.SYS	ANSI console driver.
DRIVER.SYS	Configurable external-disk-drive driver.
RAMDRIVE.SYS	Virtual disk.
VDISK.SYS	Virtual disk.
System-Configuration File Directives	
BREAK	Configure Control-C checking.
BUFFERS	Configure internal disk buffers.
COUNTRY	Set country code.
DEVICE	Install device driver.
DRIVPARM	Set block-device parameters.
FCBS	Set maximum open files using File Control Blocks (FCBs).

(more)

Command	Action
System-Configuration File Directives *(continued)*	
FILES	Set maximum open files using handles.
LASTDRIVE	Set highest logical drive.
SHELL	Specify command processor.
STACKS	Configure internal stacks.
Batch-File Directives	
AUTOEXEC.BAT	System startup batch file.
ECHO	Display text.
FOR	Execute command on file set.
GOTO	Jump to label.
IF	Perform conditional execution.
PAUSE	Suspend batch-file execution.
REM	Include comment line.
SHIFT	Shift replaceable parameters.
EDLIN Commands	
linenumber	Edit line.
A	Append lines from disk.
C	Copy lines.
D	Delete lines.
E	End editing session.
I	Insert lines.
L	List lines.
M	Move lines.
P	Display in pages.
Q	Quit.
R	Replace text.
S	Search for text.
T	Transfer another file.
W	Write lines to disk.

ANSI.SYS

2.0 and later

ANSI Console Driver

External

Purpose

Allows the user to employ a subset of the American National Standards Institute (ANSI) standard escape sequences for control of the console.

Syntax

DEVICE=[*drive*:][*path*]ANSI.SYS

where:

drive:*path*	is the drive and/or path to search for ANSI.SYS if it is not in the root directory of the startup disk.

Description

The ANSI.SYS file contains an installable character-device driver that supersedes the system's default driver for the console device (video display and keyboard). After ANSI.SYS is installed by means of a *DEVICE=ANSI.SYS* command in the CONFIG.SYS file of the disk used to start the system, programs can use a subset of the ANSI 3.64-1979 standard escape sequences to erase the display, set the display mode and attributes, and control the cursor in a hardware-independent fashion. (A supplementary set of escape sequences that are not part of the ANSI standard allows reprogramming of the keyboard.)

Programs that use ANSI.SYS for control of the screen can run on any MS-DOS machine without modification, regardless of its hardware configuration. However, most popular application programs for the IBM PC and compatibles circumvent ANSI.SYS and manipulate the video controller and its video buffer directly to achieve maximum performance.

The ANSI.SYS device driver detects ANSI escape sequences in a character stream and interprets them as commands to control the keyboard and display. An ANSI escape sequence is a sequence of ASCII characters, the first two of which must be the Escape character (1BH) and the left-bracket character (5BH). The characters following the Escape and left-bracket characters vary with the type of control function being performed; most consist of an alphanumeric code followed by a letter. In some cases this code is a single character; in others it is more than one character or a two-part string separated by a semicolon. Each ANSI escape sequence ends in a unique letter character that identifies the sequence; case is significant for these letters. The escape sequences supported by the ANSI.SYS driver are summarized in the tables on the following pages.

An escape sequence cannot be entered directly at the system prompt because each ANSI escape sequence must begin with an Escape character, and pressing the Esc key (or Alt-27 on the numeric keypad) causes MS-DOS to cancel the command line. There are three methods of executing ANSI escape sequences that do not require writing a program:

- Include the escape sequences in a PROMPT command.
- Enter the escape sequences into a word processor or text editor, save the file as an ASCII text file, and then execute the file by using the TYPE or COPY command (specifying CON as the destination for COPY) from the MS-DOS system prompt. (If the escape sequences are echoed on the screen when the file is executed, a *DEVICE=ANSI.SYS* command was not included in the CONFIG.SYS file when the system was turned on.)
- Place the escape sequences in a batch (.BAT) file as part of an ECHO command. When the batch file is executed, the sequences are sent to the console.

When escape sequences are entered using the PROMPT command, the Escape character is entered as $e. When escape sequences are entered using a word processor to create an ASCII text or batch file, the Escape character is usually entered by pressing the Esc key or by holding down the Alt key while typing 27 on the numeric keypad. (See the documentation provided with the word-processor for specific instructions.) In most cases, the escape character will appear in the word processor or text editor as a back-arrow character (←) or a caret–left bracket combination (^[).

Note: When the escape character is represented as ^[(as it is in EDLIN, for example), an additional left-bracket character must still be added to properly begin an ANSI escape sequence. Thus, the beginning of a valid ANSI escape sequence in EDLIN appears as ^[[.

The tables in this section use the abbreviation ESC to show where the ASCII escape character 27 (1BH) appears in the string.

Note: Case is significant for the terminal character in the string.

The following escape sequences control cursor movement:

Operation	Escape Sequence	Effect
Cursor Up	ESC[*number*A	Moves the cursor up *number* rows (1–24, default = 1). Has no effect if cursor is on the top row.
Cursor Down	ESC[*number*B	Moves the cursor down *number* rows (1–24, default = 1). Has no effect if cursor is on the bottom row.
Cursor Right	ESC[*number*C	Moves the cursor right *number* rows (1–79, default = 1). Has no effect if cursor is in the far right column.
Cursor Left	ESC[*number*D	Moves the cursor left *number* rows (1–79, default = 1). Has no effect if cursor is in the far left column.
Position Cursor	ESC[*row*;*column*H	Moves the cursor to the specified row (1–25, default = 1) and column (1–80, default = 1). If *row* is omitted, the semicolon before *column* must be specified.

(more)

Operation	Escape Sequence	Effect
Position Cursor	ESC[*row*;*column*f	Same as above.
Save Cursor Position	ESC[s	Stores the current row and column position of the cursor. Cursor can be restored to this position later with a Restore Cursor Position escape sequence.
Restore Cursor Position	ESC[u	Moves the cursor to the position of the most recent Save Cursor Position escape sequence.

The following two escape sequences are used to erase all or part of the display:

Operation	Escape Sequence	Effect
Erase Display	ESC[2J	Clears the screen and places the cursor at the home position.
Erase Line	ESC[K	Erases from the cursor position to the end of the same row.

The following escape sequences control the width and the color capability of the display. The use of any of these sequences clears the screen.

Operation	Escape Sequence	Effect
Set Mode	ESC[=0h	Sets display to 40 x 25 monochrome (text).
	ESC[=1h	Sets display to 40 x 25 color (text).
	ESC[=2h	Sets display to 80 x 25 monochrome (text).
	ESC[=3h	Sets display to 80 x 25 color (text).
	ESC[=4h	Sets display to 320 x 200 4-color (graphics).
	ESC[=5h	Sets display to 320 x 200 4-color (graphics, color burst disabled).
	ESC[=6h	Sets display to 640 x 200 2-color (graphics).

The following escape sequences control whether characters will wrap around to the first column of the next row after the rightmost column in the current row has been filled:

Operation	Escape Sequence	Effect
Enable Character Wrap	ESC[=7h	Sets character wrap.
Disable Character Wrap	ESC[=7l	Disables character wrap. (Note that the terminating letter is a lowercase L.)

The following escape sequence controls specific graphics attributes such as intensity, blinking, superscript, and subscript, as well as the foreground and background colors:

ESC[*attrib*;. . . ;*attrib*m

where:

attrib is one or more of the following values. Multiple values must be separated by semicolons.

Value	Attribute	Value	Foreground Color	Value	Background Color
0	All attributes off	30	Black	40	Black
1	High intensity (bold)	31	Red	41	Red
2	Normal intensity	32	Green	42	Green
4	Underline (monochrome only)	33	Yellow	43	Yellow
5	Blink	34	Blue	44	Blue
7	Reverse video	35	Magenta	45	Magenta
8	Concealed (invisible)	36	Cyan	46	Cyan
		37	White	47	White

Note: Values 30 through 47 meet the ISO 6429 standard.

The following escape sequence allows redefinition of keyboard keys to a specified *string*:

ESC[*code*;*string*;. . . p

where:

code is one or more of the following values that represent keyboard keys. Semicolons shown in this table must be entered in addition to the required semicolons in the command line.

string is either the ASCII code for a single character or a string contained in quotation marks. For example, both 65 and "A" can be used to represent an uppercase A.

Key	Code			
	Alone	Shift-	Ctrl-	Alt-
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;86	0;96	0;106
F4	0;62	0;87	0;97	0;107
F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109

(more)

Key	Code			
	Alone	Shift-	Ctrl-	Alt-
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
Home	0;71	55	0;119	–
Up Arrow	0;72	56	–	–
Pg Up	0;73	57	0;132	–
Left Arrow	0;75	52	0;115	–
Down Arrow	0;77	54	0;116	–
End	0;79	49	0;117	–
Down Arrow	0;80	50	–	–
Pg Dn	0;81	51	0;118	–
Ins	0;82	48	–	–
Del	0;83	46	–	–
PrtSc	–	–	0;114	–
A	97	65	1	0;30
B	98	66	2	0;48
C	99	67	3	0;46
D	100	68	4	0;32
E	101	69	5	0;18
F	102	70	6	0;33
G	103	71	7	0;34
H	104	72	8	0;35
I	105	73	9	0;23
J	106	74	10	0;36
K	107	75	11	0;37
L	108	76	12	0;38
M	109	77	13	0;50
N	110	78	14	0;49
O	111	79	15	0;24
P	112	80	16	0;25
Q	113	81	17	0;16
R	114	82	18	0;19
S	115	83	19	0;31
T	116	84	20	0;20
U	117	85	21	0;22
V	118	86	22	0;47
W	119	87	23	0;17
X	120	88	24	0;45

(more)

Key	Code			
	Alone	**Shift-**	**Ctrl-**	**Alt-**
Y	121	89	25	0;21
Z	122	90	26	0;44
1	49	33	–	0;120
2	50	64	–	0;121
3	51	35	–	0;122
4	52	36	–	0;123
5	53	37	–	0;124
6	54	94	–	0;125
7	55	38	–	0;126
8	56	42	–	0;127
9	57	40	–	0;128
0	48	41	–	0;129
–	45	95	–	0;130
=	61	43	–	0;131
Tab	9	0;15	–	–
Null	0;3	–	–	–

Examples

The following examples use ESC or $e to show where the ASCII escape character 27 (1BH) appears in the string. The PROMPT examples can be typed as shown, but for the examples that use ESC to denote the escape character, the actual escape character should be typed in its place.

To move the cursor to row 10, column 30 and display the string *Main Menu*, use the escape sequence

```
ESC[10;30fMain Menu
```

or

```
ESC[10;30HMain Menu
```

To move the cursor to row 5, column 10 and display the letter A (*ESC[5;10fA*), move the cursor down one row (*ESC[B*), move the cursor back one space and display the letter B (*ESC[DB*), move the cursor down one row (*ESC[B*), and move the cursor back one space and display the letter C (*ESC[DC*), use the escape sequence

```
ESC[5;10fAESC[BESC[DBESC[BESC[DC
```

To use ANSI escape sequences with the PROMPT command to save the current cursor position (*$e[s*), move the cursor to row 1, column 69 (*$e[1;69f*), display the current time using the PROMPT command's $t function, restore the cursor position (*$e[u*), and then display the current path using the PROMPT command's $p function and display a greater-than sign using the PROMPT command's $g function, use the escape sequence

```
C>PROMPT $e[s$e[1;69f$t$e[u$p$g  <Enter>
```

To erase the display (*ESC[2J*), then move the cursor to row 10, column 30 and display the string *Main Menu* (*ESC[10;30fMain Menu*), use the escape sequence

```
ESC[2JESC[10;30fMain Menu
```

To move the cursor to row 5, column 40 (*ESC[5;40f*) and erase the remainder of the row starting at the current cursor position (*ESC[K*), use the escape sequence

```
ESC[5;40fESC[K
```

To move the cursor to row 3 (*ESC[3;f*), erase the entire row (*ESC[K*), move the cursor down one row (*ESC[B*), erase that entire row (*ESC[K*), move the cursor down one row and erase that entire row, use the escape sequence

```
ESC[3;fESC[KESC[BESC[KESC[BESC[K
```

To set the display mode to 25 rows of 80 columns in color (*ESC[=3h*) and disable character wrap (*ESC[= 7l*), use the escape sequence

```
ESC[=3hESC[=7l
```

Note that ESC[=3h will also clear the screen.

To enable character wrap, use the escape sequence

```
ESC[=7h
```

To set the foreground color to black and the background color to blue (*ESC[30;44m*), clear the display (*ESC[2J*), then position the cursor at row 10, column 30 and display the string *Main Menu* (*ESC[10;30fMain Menu*), use the escape sequence

```
ESC[30;44mESC[2JESC[10;30fMain Menu
```

To (effectively) exchange the backslash and question-mark keys using literal strings to denote the keys, use the escape sequence

```
ESC["\";"?"pESC["?";"\"p
```

To exchange the backslash and question-mark keys using each key's ASCII value to denote the key, use the escape sequence

```
ESC[92;63pESC[63;92p
```

To restore the backslash and question-mark keys to their original meanings, use the escape sequence

```
ESC["\";"\"pESC["?";"?"p
```

or

```
ESC[92;92pESC[63;63p
```

To redefine the Alt-F9 key combination (*ESC[0;112*) so that it issues a CLS command (*;"CLS"*) plus a carriage return (*;13*) to execute the CLS command, then issues a DIR command piped through the SORT filter starting at column 24 (*;"DIR ¦ SORT /+24"*) followed by another carriage return, use the escape sequence

```
ESC[0;112;"CLS";13;"DIR ¦ SORT /+24";13p
```

To restore the Alt-F9 key combination to its original meaning, use the escape sequence

```
ESC[0;112;0;112p
```

APPEND

3.2

Set Data-File Search Path

External

Purpose

Specifies a search path for open operations on data files. (Also supported with some implementations of version 3.1, for use with networks.)

Syntax

APPEND [[*drive*:]*path*] [;[*drive*:]*path* ...]

or

APPEND ;

where:

path is the name of a valid directory, optionally preceded by a drive.

Description

APPEND is a terminate-and-stay-resident program that is used to specify a path or paths to be searched for data files (in contrast with the PATH command, which specifies a path to be searched for executable or batch files). The search path can include a network drive. If a program attempts to open a file and the file is not found in the current or specified directory, each path given in the APPEND command is searched.

If the APPEND command is entered with a path consisting of only a semicolon character (;), a "null" search path for data files is set; that is, no directory other than the current or specified directory is searched. This effectively cancels any search paths previously set with an APPEND command but does not free the memory used by APPEND.

An APPEND command without any parameters displays the current search path(s) for data files.

Note that a program cannot detect whether an opened file was found where it was expected (in the current or specified directory) or in some other directory specified in the APPEND command.

Warning: When an assigned drive is to be part of the search path, the ASSIGN command must be used before the APPEND command. Use of the ASSIGN command should be avoided whenever possible because it hides drive characteristics from those programs that require detailed knowledge of the drive size and format.

Examples

To cause the directories C:\SYSTEM and C:\SOURCE to be searched for a file during an open operation if the file is not found in the current or specified directory, type

```
C>APPEND C:\SYSTEM;C:\SOURCE  <Enter>
```

To display the current search path for data files, type

```
C>APPEND  <Enter>
```

MS-DOS then displays

```
APPEND=C:\SYSTEM;C:\SOURCE
```

To ensure that no directories other than the current or specified directory are searched during a file open operation, type

```
C>APPEND ;  <Enter>
```

Messages

APPEND / ASSIGN Conflict
APPEND was used before ASSIGN.

Incorrect DOS version
The version of APPEND is not compatible with the version of MS-DOS that is running.

No appended directories
The APPEND command had no parameters and no APPEND search path is active.

ASSIGN

Assign Drive Alias

3.0 and later

External

Purpose

Redirects requests for disk operations on one drive to a different drive. (Available with PC-DOS beginning with version 2.0.)

Syntax

ASSIGN [*x*=*y* [...]]

where:

x	is a valid designator (A, B, C, etc.) for a disk drive that physically exists in the system.
y	is a valid designator for the drive to be accessed by references to *x*.

Description

ASSIGN is a terminate-and-stay-resident program that redirects all references to drive *x* or files on drive *x* to drive *y*. The ASSIGN command is intended for use with application programs that require files to reside on drive A or B and have no provision within the program for changing those drives.

Multiple drive assignments can be requested in the same ASSIGN command line; the drive pairs must be separated with spaces, commas, or semicolons. Unlike the form in most other MS-DOS commands, the drive letters are not followed by colon characters (:). When a single drive is assigned, the equal sign is optional.

ASSIGN commands are not incremental. Each new ASSIGN command replaces assignments made with the previous ASSIGN command and cancels any assignments not specifically replaced. Entering ASSIGN with no parameters cancels all current drive assignments.

Warning: Use of the ASSIGN command should be avoided whenever possible because it hides drive characteristics from those programs that require detailed knowledge of the drive size and format; in particular, drives redirected with an ASSIGN statement should never be used with a BACKUP, RESTORE, LABEL, JOIN, SUBST, or PRINT command. ASSIGN can also defeat the checking performed by the COPY command to prevent a file from being copied onto itself. The FORMAT, SYS, DISKCOPY, and DISKCOMP commands ignore any drive reassignments made with ASSIGN.

With MS-DOS versions 3.1 and later, the SUBST command should be used instead of ASSIGN. For example, the command

```
C>ASSIGN A=C  <Enter>
```

should be replaced with the command

```
C>SUBST A: C:\  <Enter>
```

Examples

To redirect all requests for drive A to drive C, type

```
C>ASSIGN A=C  <Enter>
```

To redirect all requests for drives A and B to drive C, type

```
C>ASSIGN A=C B=C  <Enter>
```

To cancel all drive redirections currently in effect, type

```
C>ASSIGN  <Enter>
```

Messages

Incorrect DOS version
The version of ASSIGN is not compatible with the version of MS-DOS that is running.

Invalid parameter
One of the specified drive designators refers to a drive that does not exist in the system.

ATTRIB

3.0 and later

Change File Attributes

External

Purpose

Sets, removes, or displays a file's read-only and/or archive attributes.

Syntax

ATTRIB [+R¦–R] [+A¦–A] [*drive*:]*pathname*

where:

+R	marks the file read-only.
–R	removes the read-only attribute.
+A	sets the file's archive flag (version 3.2).
–A	removes the file's archive flag (version 3.2).
pathname	is the name and location, optionally preceded by a drive, of the file whose attributes are to be changed or displayed; wildcard characters are permitted in the filename.

Description

Each file has an entry in the disk's directory that contains its name, location, and size; the date and time it was created or last modified; and an attribute byte. For normal files, bits 0, 1, 2, and 5 in the attribute byte designate, respectively, whether the file is read-only, hidden, or system and whether it has been changed since it was last backed up.

The ATTRIB command provides a way to alter the read-only and archive bits from the MS-DOS command level. If a file is marked read-only, it cannot be deleted or modified; thus, crucial programs or data can be protected from accidental erasure. A file's archive flag can be used together with the /M switch of the BACKUP command or the /M or /A switch of the XCOPY command to allow an incremental or selective backup of files from one disk to another.

If the ATTRIB command is entered with only a pathname, the current attributes of the selected file are displayed. An R is displayed next to the name of a file that is marked read-only and an A is displayed if the file has the archive flag set.

Examples

To make the file MENUMGR.C in the current directory of the current drive a read-only file, type

```
C>ATTRIB +R MENUMGR.C  <Enter>
```

To display the attributes of the file LETTER.DOC in the directory \SOURCE on the disk in drive D, type

```
C>ATTRIB D:\SOURCE\LETTER.DOC  <Enter>
```

MS-DOS then displays

```
R A      D:\SOURCE\LETTER.DOC
```

to indicate that the file is marked read-only and the archive flag has been set.

To set the archive flag on all files in the directory \SYSTEM on drive C and mark them as read-only, type

```
C>ATTRIB +A +R C:\SYSTEM\*.*  <Enter>
```

Messages

Access denied
ATTRIB cannot be used to alter or replace the attributes of a file in use across a network.

DOS 2.0 or later required
ATTRIB does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of ATTRIB is not compatible with the version of MS-DOS that is running.

Invalid number of parameters
More than two attributes were used before the pathname.

Invalid path or file not found
The file named in the command line or one of the directories in the given path does not exist.

Syntax error
An invalid attribute was supplied or the attribute was not properly placed before the pathname in the command line.

BACKUP

2.0 and later

Back Up Files

External

Purpose

Creates backup copies of files, along with the associated directory information necessary to restore the files to their original locations.

Syntax

BACKUP *source destination* [/A] [/D:*date*] [/L:*filename*] [/M] [/P] [/S] [/T:*time*]

where:

source is the location (drive and/or path) and, optionally, the name of the files to be backed up; wildcard characters are permitted in the filename.

destination is the drive to receive the backup files.

/A adds the files to existing files on the destination disk without erasing the destination disk.

/D:*date* backs up only those files modified on or after *date.*

/L:*filename* creates a log file with the specified name in the root directory of the disk being backed up. If *filename* is not specified, BACKUP creates a file named BACKUP.LOG and places the log entries there. Use of the /L:*filename* switch may cause loss of IBM compatibility.

/M backs up only those files modified since the last backup.

/P packs the destination disk with as many files as possible, creating subdirectories, if necessary, to hold some of the files. Use of the /P switch causes loss of IBM compatibility.

/S backs up the contents of all subdirectories of the source directory.

/T:*time* backs up only those files modified on or after *time.*

Note: Not all switches are supported by all implementations of MS-DOS.

Description

The BACKUP command creates a backup copy of the specified file or files, transferring them from either a floppy disk or a fixed disk to another removable or fixed disk. The backup file is in a special format that includes information about the original file's location in the directory structure. Files created by BACKUP can be restored to their original form only with the RESTORE command.

BACKUP can back up a single file or many files in the same operation. If only a drive letter is given as the source, all the files in the current directory of that disk are backed up. If only a path is given as the source, all the files in the specified directory are backed up. If the /S switch is used, all the files in the current or specified directory are backed up, and the files in all its subdirectories as well. If both a path and a filename are entered as the source, the specified file or files in the named directory are backed up.

If the source file is marked read-only, the resulting backup file will also be marked read-only. If the source file's archive bit is set, it will be cleared for both the source and the destination files. BACKUP also backs up hidden files; the files will remain hidden on the destination disk.

If the destination disk is a floppy disk, its previous contents are erased as part of the backup operation (unless the /A switch is included in the command line and the destination disk has already been used as a backup disk—that is, the disk contains a valid BACKUPID.@@@ file). If the files being backed up do not fit onto a single floppy disk, the user will be prompted to insert additional disks until the backup operation is complete.

If the destination disk is a fixed disk, the backed-up files are placed in a directory named \BACKUP. If a \BACKUP directory already exists on the fixed disk, any files previously contained in it are erased as part of the backup operation (unless the /A switch is included in the command line and the destination disk has already been used as a backup disk—that is, the \BACKUP directory contains a valid BACKUPID.@@@ file). Other files on the destination fixed disk are not disturbed.

A control file named BACKUPID.@@@ is placed on every floppy disk onto which files are backed up or in the /BACKUP directory if the files are backed up onto a fixed disk. The BACKUPID.@@@ file has the following format:

Byte	Value	Use
00H	00 or FFH	Not last floppy disk/last floppy disk
01–02H	*nn*	Floppy disk number in low-byte/high-byte decimal format
03–04H	*nnnn*	Full year in low-byte/high-byte order
05H	1–31	Day of the month
06H	1–12	Month of the year
07–0AH	*nnnn*	Standard MS-DOS system time if the /T:*time* switch was used; otherwise 0
0B–7FH	00	Not used

Each backed-up file also has a 128-byte header added to it when it is created. The header has the following format:

Byte	Value	Use
00H	00 or FFH	Not last floppy disk/last floppy disk on which this file resides
01H	*nn*	Floppy disk number
02–04H	00	Not used

(more)

Byte	Value	Use
05–44H	*nn*	File's full pathname, except for drive designator
45–52H	00	Not used
53H	*nn*	Length of the file's pathname plus one
54–7FH	00	Not used

The /T:*time*, /D:*date*, and /M switches allow incremental or partial backups. The /T:*time* switch excludes files modified or created before a certain time and should be used in the form of the COUNTRY command in effect. For the USA, the format is /T:*hh*:*mm*:*ss*. (The /T:*time* switch is not supported in all implementations of BACKUP.) The /D:*date* switch excludes files modified or created before a certain date and should be used in the form of the COUNTRY command in effect. For the USA, the format is /D:*mm-dd-yy*. The /M switch selects only those files that have been modified since the last backup operation.

The /L:*filename* switch causes a log file to be created on the source disk. This file includes the name of each file backed up, the time and date, and the number of the destination disk that received that backup file. If *filename* is omitted, the name defaults to BACKUP.LOG. Use of the /L:*filename* switch can cause compatibility problems between MS-DOS and PC-DOS because the backup log file may match the search pattern and be backed up, too, resulting in an extra file on the backup disk.

The /P switch causes backup files to be packed as densely as possible on the destination disk. When many short files are being backed up to floppy disks, the number of files that fit on the destination disk may exceed the number of entries that will fit in the destination's root directory. If the /P switch is included in the command line, subdirectories are created on the destination disk as needed to use the disk space more effectively. The /P switch is not supported under PC-DOS; backup disks created with the /P switch will not be compatible with IBM's BACKUP and RESTORE commands.

Warning: BACKUP should not be used on disk directories or drives that have been redirected with an ASSIGN, JOIN, or SUBST command.

Return Codes

0 Backup operation was successful.
1 No files were found to back up.
2 Some files were not backed up because of sharing conflicts (versions 3.0 and later).
3 Backup operation was terminated by user.
4 Backup operation was terminated because of error.

Examples

To back up the file REPORT.TXT in the current directory on the current drive, placing the backup file on the disk in drive A, type

```
C>BACKUP REPORT.TXT A:  <Enter>
```

To back up all the files in the subdirectory B:\V2\SOURCE, placing the backup files on the disk in drive A, type

```
C>BACKUP B:\V2\SOURCE A:  <Enter>
```

To back up all the files with extension .C in the directory \V2\SOURCE on the current drive, placing the backup files on the disk in drive A, type

```
C>BACKUP \V2\SOURCE\*.C A:  <Enter>
```

To back up all the files with the extension .ASM from the current directory on the current drive and from all its subdirectories, placing the backup files on the disk in drive A, type

```
C>BACKUP *.ASM A: /S  <Enter>
```

To back up all the files that have been modified since the last backup from all the subdirectories on drive C, placing the backup files on the disk in drive A, type

```
C>BACKUP C:\ A: /S /M  <Enter>
```

To back up all the files with the extension .C from the directory C:\V2\SOURCE that were modified on or after October 16, 1985, placing the backup files on the disk in drive A, type

```
C>BACKUP C:\V2\SOURCE\*.C A: /D:10-16-85  <Enter>
```

Messages

*****Backing up files to drive *X*: *****
Diskette Number: *n*
This informational message informs the user of the progress of the BACKUP command.

*****Last file not backed up *****
The destination drive does not have enough space to back up the last file.

*****Not able to back up file *****
One of the system calls used by BACKUP failed unexpectedly; for example, a file could not be opened, read, or written.

Cannot create Subdirectory BACKUP on drive *X*:
Drive *X* is full or its root directory is full.

DOS 2.0 or later required
BACKUP does not work with versions of MS-DOS earlier than 2.0.

Error trying to open backup log file
Continuing without making log entries
The /L switch was used and BACKUP is unable to create the backup log file.

Files cannot be added to this diskette
unless the PACK (/P) switch is used
Set the switch (Y/N)?
The root directory of the destination disk is full and a subdirectory must be created to hold the remaining files. Respond with *Y* to cause BACKUP to create a subdirectory and continue backing up files into it; respond with *N* to return to MS-DOS.

Incorrect DOS version
The version of BACKUP is not compatible with the version of MS-DOS that is running.

Insert backup diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert a disk to receive the backup files into the specified destination drive.

Insert backup diskette *n* in drive *X*:
Strike any key when ready
The files being backed up will not fit onto a single floppy disk; this message prompts the user to insert the next floppy disk. Multiple-floppy-disk backup disks should be labeled and numbered to match the number displayed in this message.

Insert backup source diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert the floppy disk to be backed up into the specified source drive.

Insert last backup diskette in drive *X*:
Strike any key when ready
This message prompts the user to insert the final disk that will receive the backup files into the specified destination drive.

Insufficient memory
Available system memory is insufficient to run the BACKUP program.

Invalid argument
One of the switches specified in the command line is invalid or is not supported in the version of BACKUP being used.

Invalid Date/Time
An invalid date or time was given with the /D:*date* or /T:*time* switch.

Invalid drive specification
The source or destination drive specified in the command line is not available or is not valid.

Invalid number of parameters
At least two parameters, the source and the destination, must be specified in the command line; a maximum of seven switches can be specified after the source and destination.

Invalid parameter
One of the switches supplied in the command line is invalid.

Invalid path
The path specified as the source is invalid or does not exist.

Last backup diskette not inserted
Insert last backup diskette in drive *X*:
Strike any key when ready
The backup disk inserted as the last backup disk was not the correct disk. Insert the correct disk.

No space left on device
The destination disk is full.

No such file or directory
The source specified is invalid or does not exist.

Source and target drives are the same
The disks specified as the source and destination disks are identical.

Source disk is Non-removable
The disk containing the files to be backed up is a fixed disk.

Target can not be used for backup
The disk specified as the destination disk is damaged or the /A switch was used in the command line and the disk does not contain a valid BACKUPID.@@@ file.

Target disk is Non-removable
The disk that will contain the backed-up files is a fixed disk.

Target is a floppy disk

or

Target is a hard disk
This informational message indicates which type of disk was specified as the destination disk.

Too many open files
Too many files are open. Increase the value of the FILES command in the CONFIG.SYS file.

Unable to erase *filename*
BACKUP is unable to erase an older version of a backed-up file because the file is read-only or is in use by another program.

Warning! Files in the target drive
***X*:\root directory will be erased**
Strike any key when ready
The destination is a floppy-disk drive and this message warns the user that all files in its root directory will be erased before the backup operation.

Warning! Files in the target drive
C:\BACKUP directory will be erased
Strike any key when ready
BACKUP is ready to begin backing up files to the \BACKUP directory on drive C. All existing files in the \BACKUP directory will be deleted. Press Crtl-Break to terminate the backup operation or press any key to continue.

Warning! No files were found to back up
No files were found on the source disk in the current or specified directory or no files were found matching the filename supplied.

BATCH

1.0 and later

System Batch-File Interpreter

Internal

Purpose

Sequentially executes commands stored in a batch file (a text-only file with a .BAT extension).

Syntax

filename [[*parameter1* [*parameter2* [...]]]]

where:

filename is the name of the batch file to be executed, without the .BAT extension. (The filename is always %0 in the list of replaceable parameters.)

parameter1 is the filename, switch, or string that is the value of the first replaceable parameter (%1).

parameter2 is the filename, switch, or string that is the value of the second replaceable parameter (%2). As many additional replaceable parameters can be specified as the command line will hold.

Description

A batch file is an ASCII text file that contains one or more MS-DOS commands. It is a useful way to perform sequences of frequently used commands without having to type them all each time they are needed. When a batch file is invoked by entering its name, the commands it contains are carried out in sequence by a special batch-file interpreter built into COMMAND.COM. Additional information entered in the batch-file command line can be passed to other programs by means of replaceable parameters (*see below*).

A batch file must always have the extension .BAT. The file can contain any number of lines of ASCII text; each line can contain a maximum of 128 characters. Batch files can be created with EDLIN or another text editor or with a word processor in nondocument mode. (Formatted document files cannot be used as batch files because they contain special control codes or escape sequences that cannot be processed by the batch-file interpreter.) Batch files can also be created with the MS-DOS COPY command by specifying the CON device (keyboard) as the source file and the desired batch-file name as the destination file. For example, after the command

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

each line that is typed will be placed into MYFILE.BAT. This form of the COPY command is terminated by pressing Ctrl-Z or the F6 key, followed by the Enter key.

The commands in a batch file can be any combination of internal MS-DOS commands (such as DIR or COPY), external MS-DOS commands (such as CHKDSK or BACKUP), the names of other programs or batch files, or the following special batch-file directives:

Command	Action
ECHO	Displays a message on standard output (versions 2.0 and later).
FOR	Executes a command on each of a set of files (versions 2.0 and later).
GOTO	Transfers control to another point in a batch file (versions 2.0 and later).
IF	Conditionally executes a command based on the existence of a file, the equality of two strings, or the return code of a previously run program (versions 2.0 and later).
PAUSE	Waits for the user to press a key before executing the remainder of the batch file.
REM	Allows comment lines to be placed in batch files for internal documentation.
SHIFT	Provides access to more than 10 command-line parameters (versions 2.0 and later).

These special batch commands are discussed individually, with examples, in the following pages.

A batch file is executed by entering its name, without the .BAT extension, in response to the MS-DOS prompt. The system's command processor, COMMAND.COM, searches the current directory and then each directory named in the PATH environment variable for a file with the specified name and the extension .COM, .EXE, or .BAT, in that order. If a .COM or .EXE file is found, it is loaded into memory and receives control; if a .BAT file is found, it is assumed to be a text file and is passed to the batch-file interpreter. (If two files with the same name exist in the same directory, one with a .COM or .EXE extension and the other with a .BAT extension, it is not possible to execute the .BAT file—the .COM or .EXE file is always loaded instead.)

If the disk that contains a batch file is removed before all the commands in the batch file are executed, COMMAND.COM will prompt the user to replace the disk so that the batch file can be completed. Execution of a batch file can be terminated by pressing Ctrl-C or Ctrl-Break, causing COMMAND.COM to issue the message *Terminate batch job? (Y/N)*. If the user responds with *Y*, the batch file is abandoned and COMMAND.COM displays its usual prompt.

The input redirection (<), output redirection (> or >>), and piping (¦) characters have no effect when they are used in a command line that invokes a batch file. However, they can be used in individual command lines *within* the file.

Ordinarily, if a batch file includes the name of another batch file, control passes to the second batch file and never returns. That is, when the commands in the second batch file are completed, the batch-file interpreter terminates and any remaining commands in the first batch file are not processed. However, a batch file can execute another batch file without itself being terminated by first loading a secondary copy of the system's command processor. To accomplish this, the first batch file must contain a command of the form

COMMAND /C *batch2*

where *batch2* is the name of the second batch file. When all the commands in the second batch file have been processed, the secondary copy of COMMAND.COM exits and the first batch file continues where it left off. (*See* USER COMMANDS: COMMAND for details on the use of the /C switch with COMMAND.COM.)

A batch file can be made more flexible by including replaceable parameters inside the file. A replaceable parameter takes the form *%n*, where *n* is a numeral in the range 0 through 9. Replaceable parameters simply hold places in the batch file for filenames or other information that the user will supply in the command line when the batch file is invoked.

When a batch file is interpreted and a command containing a replaceable parameter is encountered, the corresponding value specified in the batch-file command line is substituted for the replaceable parameter and the command is then executed. The %0 replaceable parameter is replaced by the name of the batch file itself; parameters %1 through %9 are replaced sequentially with the remaining values specified in the command line. If a replaceable parameter references a command-line entry that does not exist, the parameter is replaced with a null (zero-length) string.

For example, if the batch file MYBATCH.BAT contains the single line

```
COPY %1.COM %2.SAV
```

and is executed by entry of

```
C>MYBATCH FILE1 FILE2  <Enter>
```

the actual command that is carried out is

```
COPY FILE1.COM FILE2.SAV
```

(The SHIFT batch command makes it possible to use more than 10 replaceable parameters. *See* USER COMMANDS: BATCH:SHIFT)

An environment variable is a special case of a replaceable parameter. If the SET command is used in the form

SET *name=value*

to add an environment variable to the system's environment block, the string *value* will be substituted for the string %*name*% wherever the latter is encountered during the interpretation of a batch file. This capability is available only in versions 2.x, 3.1, and 3.2.

BATCH: AUTOEXEC.BAT

1.0 and later

System Startup Batch File

Description

The AUTOEXEC.BAT file is an optional batch file containing a series of MS-DOS commands that automatically execute when the system is turned on or restarted.

When the system's default command processor, COMMAND.COM, is first loaded, it looks in the root directory of the current drive for a file named AUTOEXEC.BAT. If AUTOEXEC.BAT is not found, COMMAND.COM prompts the user to enter the current time and date and then displays the MS-DOS copyright notice and command prompt. If AUTOEXEC.BAT is found, COMMAND.COM sequentially executes the commands within the file. No prompts to enter the time and date are issued unless the TIME and DATE commands are explicitly included in the batch file; no copyright notice is displayed.

Typical uses of the AUTOEXEC.BAT file include

- Running a program to set the system time and date from a real-time clock/calendar located on a multipurpose expansion board (IBM PC, PC/XT, or compatibles only)
- Using the MODE command to configure a serial port or to redirect printing
- Executing SET commands to configure environment variables
- Setting display colors on a color monitor (if the command *DEVICE=ANSI.SYS* has been included in the CONFIG.SYS file)
- Installing terminate-and-stay-resident (TSR) utilities
- Using the PATH command to tell COMMAND.COM where to find executable program files if they are not in the current drive and/or directory
- Defining a custom prompt using the PROMPT command
- Invoking an application program such as a database, spreadsheet, or word processor

A secondary copy of the command processor can also be loaded from within the AUTOEXEC.BAT file. If this copy of COMMAND.COM is loaded with the /P switch, it too searches for an AUTOEXEC.BAT file on the current drive and processes the file if it is found. This feature can be useful for performing special operations. For example, on very old PCs that are unable to start from a fixed disk, a secondary copy of the command processor can be used to make the fixed disk's copy of COMMAND.COM the copy used by the system from that point on (at the expense of some system memory). If the AUTOEXEC.BAT file containing the lines

```
C:
COMMAND C:\ /P
```

is stored on the floppy disk in drive A when the system is turned on or restarted, the first line of the file causes drive C to become the current drive; then the second line permanently loads a secondary copy of COMMAND.COM from drive C and instructs COMMAND.COM to reload its transient portion from the root directory of drive C when necessary. This in turn triggers the execution of the AUTOEXEC.BAT file on the fixed disk to perform the actual system configuration. Because the transient part of COMMAND.COM will be reloaded from the fixed disk when necessary, rather than from the floppy disk, system performance is improved considerably.

Example

The following example illustrates several common uses of the AUTOEXEC.BAT file to configure the MS-DOS system at startup time. (The line numbers are included for reference and are not part of the actual file.)

```
1    ECHO OFF
2    SETCLOCK
3    PROMPT $p$g
4    MD D:\BIN
5    COPY C:\SYSTEM\*.* D:\BIN > NUL
6    PATH=D:\BIN;C:\WP\WORD;C:\MSC\BIN;C:\ASM
7    APPEND D:\BIN;C:\WP\WORD;C:\ASM
8    SET INCLUDE=C:\MSC\INCLUDE
9    SET LIB=C:\MSC\LIB
10   SET TMP=C:\TEMP
11   MODE COM1:9600,n,8,1,p
12   MODE LPT1:=COM1:
```

Line 1 causes the batch-file processor to operate silently; that is, the commands in the batch file are not displayed on the screen as they are executed.

Line 2 runs a utility program called SETCLOCK, which reads the current time and date from a real-time clock chip on a multifunction board and sets the system time and date accordingly.

Line 3 configures COMMAND.COM's user prompt so that it displays the current drive and directory.

Line 4 creates a directory named \BIN on drive D, which in this case is a RAMdisk that was created by an entry in the system's CONFIG.SYS file.

Line 5 copies all the programs in the \SYSTEM directory on drive C to the \BIN directory on drive D. The normal output of this COPY command is redirected to the NUL device — in effect, the output is thrown away — to avoid cluttering the screen.

Line 6 sets the search path for executable files and line 7 sets the search path for data files. Note that the RAMdisk directory D:\BIN is specified as the first directory in the PATH command; therefore, if the name of a program is entered and it cannot be found in the current directory, COMMAND.COM will look next in the directory D:\BIN. This strategy allows commonly used programs (in this example, the programs in the \SYSTEM directory that were copied into D:\BIN) to be located and loaded quickly.

Lines 8 through 10 add the environment variables INCLUDE, LIB, and TMP to the system's environment. These variables are used by the Microsoft C Compiler and the Microsoft Object Linker.

Line 11 configures the first serial communications port (COM1) and line 12 causes program output to the system's first parallel port (LPT1) to be redirected to the first serial port. This pair of commands allows a serial-interface Hewlett Packard LaserJet printer to be used as the system list device.

Note: Depending on the version of MS-DOS in use, some commands in this example may not be available or may support different options. See the individual command entries for more detailed information.

BATCH: ECHO

2.0 and later

Display Text

Internal

Purpose

Displays a message during the execution of a batch file and controls whether or not batch-file commands are listed on the screen as they are executed.

Syntax

ECHO [ON¦OFF¦*message*]

where:

ON	enables the display of all subsequent batch-file commands as they are executed.
OFF	disables the display of all subsequent batch-file commands as they are executed.
message	is a text string to be displayed on standard output.

Description

Each command line of a batch file is ordinarily displayed on the screen as it is executed. The ECHO command has a dual usage: to control the display of these commands and to display a message to the user.

ECHO is used with ON or OFF to enable or disable the display of commands during batch-file processing. If the ECHO command is used with no parameter, the current status of the batch processor's ECHO flag is displayed. Note that the ECHO flag is always forced on at the start of any batch-file processing, even if that batch file was invoked by another batch file.

The ECHO command is not limited to batch files; an ECHO command can also be issued at the command prompt. ECHO OFF entered at the command prompt prevents the prompt from subsequently being displayed. ECHO ON entered interactively restores the display. If ECHO is entered interactively without a parameter, the current status of the ECHO flag is displayed.

ECHO can also be followed by a message to be sent to standard output regardless of the status of the ECHO flag (on or off). Note that if ECHO is on, two copies of the message are actually displayed, the first copy preceded by the word *ECHO.* ECHO *message* is frequently used to display prompts and informative text during the execution of a batch file because text following REM or PAUSE commands is not displayed if ECHO is off.

ECHO *message* can also be used to build lists or other batch files dynamically while the batch file is executing. For example, the messages in the following ECHO commands are used to build the file STARTUP.BAT:

```
ECHO CHKDSK > STARTUP.BAT
ECHO DIR /W >> STARTUP.BAT
ECHO PROMPT $p$g >> STARTUP.BAT
```

The first ECHO command causes the message *CHKDSK* to be redirected to the file STARTUP.BAT. The second and third ECHO commands cause the messages *DIR/W* and *PROMPT pg* to be appended to the existing contents of STARTUP.BAT. The completed STARTUP.BAT file contains the following:

```
CHKDSK
DIR /W
PROMPT $p$g
```

Note: When the pipe symbol (¦) is used in *message*, the symbol and any characters following it are ignored until a redirection symbol (<, >, or >>) is encountered, at which point the redirection symbol and the remaining characters are recognized. For example, if the line

```
ECHO DIR ¦ SORT > STARTUP.BAT
```

was placed in a batch file and subsequently executed, the only characters echoed to the file STARTUP.BAT would be *DIR*; the pipe symbol and the characters between it and the redirection symbol > would be ignored.

Examples

To disable the display of each batch-file command as it is executed, include the following line as the first line in the batch file:

```
ECHO OFF
```

To display the message *Now formatting disk* on standard output, include the following line in the batch file:

```
ECHO Now formatting disk
```

To display the current status of the ECHO flag, include the following line in the batch file:

```
ECHO
```

If the ECHO flag is currently off, MS-DOS displays:

```
ECHO is off
```

To echo a blank line to the screen with versions 2.x, type a space after the ECHO command and press Enter. To echo a blank line with versions 3.x, type the ECHO command and a space, then hold down Alt and type 255 on the numeric keypad; finally, release the Alt key and press Enter.

Messages

ECHO is off

or

ECHO is on
If the ECHO command is entered without a parameter, one of these lines is displayed to give the current status of the batch processor's ECHO flag.

BATCH: FOR

2.0 and later

Execute Command on File Set

Internal

Purpose

Executes a command or program for each file in a set of files.

Syntax

FOR %%*variable* IN (*set*) DO *command* (batch processing)

or

FOR %*variable* IN (*set*) DO *command* (interactive processing)

where:

variable is a variable name that can be any single character except the numerals 0 through 9, the redirection symbols (<, >, and >>), and the pipe symbol (¦); case is significant.

set is one or more filenames, pathnames, character strings, or metacharacters, separated by spaces, commas, or semicolons; wildcard characters are permitted in filenames.

command is any MS-DOS command or program except the FOR command; the variable name %%*variable* (or %*variable* in interactive mode) can be part of the command.

Description

The FOR command allows sequential execution of the same command or program on each member of a set of files.

The *set* parameter can contain multiple filenames (including wildcards), pathnames, character strings, or metacharacters such as the replaceable parameters %0 through %9. Each of the following lines is an example of a valid set:

```
(FILE1.TXT %1 %2 B:\PROG\LISTING?.TXT)
(A:\%1 A:\%2 C:\LETTERS\*.TXT C:MEMO?.*)
(%PATH%)
```

Each filename from *set* is assigned in turn to %*variable* and then the specified command or program is executed. (When the FOR command line is executed in a batch file, the leading percent sign of %%*variable* is removed, leaving %*variable*.) If a filename in *set* contains wildcards, each matching file is used before the batch processor goes on to the next member of *set*.

Note: In versions 2.x, *set* can consist only of a list of single filenames, a single filename with wildcard characters, or a combination of single filenames and metacharacters. In versions 3.x, however, all combinations of these are allowed in the same set.

The FOR command can also be used interactively at the MS-DOS prompt to perform a single command on several files without entering the same command for each file. When FOR is used in this manner, only one percent sign (%) should be used before the dummy alphabetic variable; in this case, the percent sign is not removed during processing. When the FOR command is used interactively, environment variables such as %PATH% cannot be used as part of the filename set.

Examples

To view all the files with the extension .TXT in the current directory, include the following line in the batch file:

```
FOR %%X IN (*.TXT) DO TYPE %%X
```

To perform the same function interactively, type

```
C>FOR %X IN (*.TXT) DO TYPE %X  <Enter>
```

To copy up to nine files to the disk in drive A, specifying the names of the files in the batch-file command line, include the following line in the batch file:

```
FOR %%Y IN (%1 %2 %3 %4 %5 %6 %7 %8 %9) DO COPY %%Y A:
```

(Recall that %0 is the name of the batch file.)

To execute successive batch files under the control of one batch file, use the /C switch with COMMAND, as in the following batch-file line:

```
FOR %%Z IN (BAT1 BAT2 BAT3) DO COMMAND /C %%Z
```

Message

FOR cannot be nested
The command or program performed by a FOR command cannot be another FOR command.

BATCH: GOTO

2.0 and later

Jump to Label

Internal

Purpose

Transfers program control to the batch-file line following the specified label.

Syntax

GOTO *name*

where:

name is a batch-file label declared elsewhere in the file in the form *:name.*

Description

The GOTO command causes the batch-file processor to transfer its point of execution to the line following the specified label. If the label does not exist in the file, execution of the batch file is terminated with the message *Label not found.*

A batch-file label is defined as a line with a colon character (:) in the first column, followed by any text (including spaces but not other separator characters such as semicolons or equal signs). Only the first eight characters following the colon are significant; spaces are not counted in the eight characters.

Examples

The GOTO command is frequently used in combination with the IF and SHIFT batch commands to perform some action based on the return code from a program. For example, the following batch file will back up a variable number of files or directories, whose names are specified in the batch-file command line, to a floppy disk in drive A. The batch file accomplishes this by executing the BACKUP program with successive pathnames specified in the command line until BACKUP returns a nonzero (error) code. Control is then transferred to the label *:DONE,* and the batch file is terminated.

```
1   ECHO OFF
2   :START
3   BACKUP %1 A:
4   IF ERRORLEVEL 1 GOTO DONE
5   SHIFT
6   GOTO START
7   :DONE
```

Note that the batch file includes two labels, *:START* and *:DONE,* in lines 2 and 7, respectively. It also includes two GOTO commands, in lines 4 and 6. (The line numbers in the listing above are included only for reference and are not present in the actual batch file.) If the condition in line 4 is true (the BACKUP program returned an exit code of 1 or higher), the remainder of line 4 is executed and program control passes to the *:DONE* label in line 7. If the condition is false, program control passes to line 5, the SHIFT command is executed, and program control goes to line 6, where the GOTO statement returns program control to line 2.

Message

Label not found
The specified label does not exist in the batch file.

BATCH: IF

2.0 and later

Perform Conditional Execution

Internal

Purpose

Tests a condition and executes a command or program if the condition is met.

Syntax

IF [NOT] *condition command*

where:

condition is one of the following:

ERRORLEVEL *number*
The condition is true if the exit code of the program last executed by COMMAND.COM was equal to or greater than *number*. Note that not all MS-DOS commands return explicit exit codes.

string1==string2
The condition is true if *string1* and *string2* are identical after parameter substitution; case is significant. The strings cannot contain separator characters such as commas, semicolons, equal signs, or spaces.

EXIST *pathname*
The condition is true if the specified file exists. The pathname can include metacharacters.

command is the command or program to be executed if the condition is true.

Description

The IF command provides conditional execution of a command or program in a batch file. When *condition* is true, IF executes the specified command, which can be another IF command, any other MS-DOS internal command, or a program. When *condition* is not true, MS-DOS ignores *command* and proceeds to the next line in the batch file. The sense of any condition can be reversed by preceding the test or expression with NOT.

Examples

To branch to the label *:ERROR* if the file LEDGER.DAT does not exist, include the following line in the batch file:

```
IF NOT EXIST LEDGER.DAT GOTO ERROR
```

To branch to the label *:ONEPAR* if the batch-file command line does not contain at least two parameters, include the following line in the batch file:

```
IF "%2"==""GOTO ONEPAR
```

or

```
IF %2~==~ GOTO ONEPAR
```

Note that the existence of a replaceable parameter can be determined by concatenating it to another string. In the first example, quotation marks are concatenated on either side of the replaceable parameter; if %2 doesn't exist, *"%2"==""* evaluates to *""==""*, which is true and will allow *GOTO ONEPAR* to be executed. In the second example, a tilde character is concatenated to the end of the replaceable parameter; if %2 doesn't exist, the argument becomes ~==~.

To copy the file specified by the first replaceable batch-file parameter to drive A only if it does not already exist on the disk in drive A, include the following line in the batch file:

```
IF NOT EXIST A:%1 COPY %1 A:
```

To branch to the label *:DONE* if the first replaceable batch-file parameter exists in the \PROG directory on drive C *and* in the \BACKUP directory on drive C, include the following line in the batch file:

```
IF EXIST C:\PROG\%1 IF EXIST C:\BACKUP\%1 GOTO DONE
```

Messages

Bad command or filename
The command following the condition in the IF statement was misspelled, does not exist, or was represented by a replaceable parameter that was not supplied in the command line that invoked the batch file.

Syntax error
The condition specified in the IF statement cannot be tested.

BATCH: PAUSE

1.0 and later

Suspend Batch-File Execution

Internal

Purpose

Displays a message, suspends execution of a batch file, and waits for the user to press a key.

Syntax

PAUSE [*message*]

where:

message is a text string to be displayed on standard output.

Description

The PAUSE command displays the message *Strike a key when ready...* and suspends execution of a batch file until the user presses a key. This command can be used to allow time for the operator to change disks, change the type of forms on the printer, or take some other action that is necessary before the batch file can continue.

If the batch processor's ECHO flag is on when the PAUSE command is executed, the entire line containing the PAUSE statement is displayed on the screen so that the optional message is visible to the user. The message *Strike a key when ready...* is then displayed on a new line and the system waits. Note that *Strike a key when ready...* is *always* displayed, even if the ECHO flag is off. When the user presses a key, execution of the batch file resumes.

Note: Redirection symbols should not be used within *message*. They prevent the message *Strike a key when ready...* from being displayed on the screen.

If the user presses Ctrl-C or Ctrl-Break while a PAUSE command is waiting for a key to be pressed, a prompt is displayed that gives the user the opportunity to terminate the execution of the batch file. This same message is displayed whenever the user presses Ctrl-C or Ctrl-Break during the execution of a batch file; however, using PAUSE commands supplemented by appropriate ECHO commands at strategic points within a batch file provides the user with clearly defined breakpoints for terminating the file.

Examples

To display the message *Put an empty disk in drive A* and then wait until the user has pressed a key, include the following line in the batch file:

```
PAUSE Put an empty disk in drive A
```

When this line of the batch file is executed, if the ECHO flag is on, the user sees the following messages on the screen:

```
C>PAUSE Put an empty disk in drive A
Strike a key when ready . . .
```

If the ECHO flag is off, only the message *Strike a key when ready*... appears.

To display the message without the prompt and command, the PAUSE command can be used immediately after an ECHO command, as follows:

```
ECHO OFF
CLS
ECHO Put an empty disk in drive A
PAUSE
```

This batch file will display the following message on the screen:

```
Put an empty disk in drive A
Strike a key when ready . . .
```

Note that the message must be included in an ECHO command. With ECHO off, a PAUSE message is not displayed.

BATCH: REM

Include Comment Line

1.0 and later

Internal

Purpose

Designates a remark, or comment, line in a batch file.

Syntax

REM [*message*]

where:

message is any text.

Description

The REM command allows inclusion of remarks, or comments, within a batch file. Remarks are often used to document the purpose of other commands within the file for the benefit of those who may wish to modify the file later.

If the ECHO flag is on, remarks are displayed on the screen during the execution of a batch file. Thus, remarks can also be used to provide information, guidance, or prompts to the user; however, the ECHO and PAUSE commands are more suitable for these purposes.

REM can also be used alone to insert blank lines in a batch file to improve readability. (If ECHO is on, the word *REM* will still be displayed.)

Note: The redirection symbols (<, >, and >>) and piping character (¦) produce no meaningful results with the REM command and should not be used.

Example

To document a batch file's revision history with the internal comment *This batch file last modified on 6/18/87*, include the following line in the batch file:

```
REM This batch file last modified on 6/18/87
```

BATCH: SHIFT

2.0 and later

Shift Replaceable Parameters

Internal

Purpose

Changes the position of the replaceable parameters in a batch-file command line, thereby allowing more than 10 replaceable parameters.

Syntax

SHIFT

Description

Ordinarily only 10 replaceable parameters (%0 through %9, where %0 is the name of the batch file) can be referenced within a batch file. The SHIFT command allows access to additional parameters specified in the command line by shifting the contents of each of the previously assigned parameters to a lower number (%1 becomes %0, %2 becomes %1, and so on). The previous contents of %0 are lost and are not recoverable. The eleventh parameter in the batch-file command line is then moved into %9. This allows more than 10 parameters to be specified in the batch-file command line and subsequently processed in the batch file.

Example

The following batch file will copy a variable number of files, whose names are entered in the batch-file command line, to the disk in drive A:

```
ECHO OFF
:NEXT
IF "%1"=="" GOTO DONE
COPY %1 A:
SHIFT
GOTO NEXT
:DONE
```

BREAK

2.0 and later

Set Control-C Check

Internal

Purpose

Sets or clears MS-DOS's internal flag for Control-C checking.

Syntax

BREAK [ON¦OFF]

Description

Pressing Ctrl-C or Ctrl-Break while a program is running ordinarily terminates the program, unless the program itself contains instructions that disable MS-DOS's Control-C handling. As a rule, MS-DOS checks the keyboard for a Control-C only when a character is read from or written to a character device (keyboard, screen, printer, or auxiliary port). Therefore, if a program executes for long periods without performing such character I/O, detection of the user's entry of a Control-C may be delayed. The BREAK ON command causes MS-DOS to also check the keyboard for a Control-C at the time of each system call (which slows the system somewhat); the BREAK OFF command disables such extended Control-C checking. The default setting for BREAK is off.

If the BREAK command is entered alone, the current status of MS-DOS's internal BREAK flag is displayed.

Examples

To display the current status of the MS-DOS internal flag for extended Control-C checking, type

```
C>BREAK  <Enter>
```

MS-DOS displays

```
BREAK is off
```

or

```
BREAK is on
```

depending on the status of the BREAK flag.

To enable extended checking for Control-C during disk operations, type

```
C>BREAK ON  <Enter>
```

Messages

BREAK is on

or

BREAK is off
Extended Control-C checking is enabled or disabled, respectively. These messages occur in response to a BREAK status check.

Must specify ON or OFF
An invalid parameter was supplied in a BREAK command.

CHDIR or CD

2.0 and later

Change Current Directory

Internal

Purpose

Changes the current directory or displays the current path of the specified or default disk drive.

Syntax

CHDIR [*drive*:][*path*]

or

CD [*drive*:][*path*]

where:

drive is the letter of the drive for which the current directory will be changed or displayed, followed by a colon. Note that use of the *drive* parameter does not change the currently active drive.

path is one or more directory names, separated by backslash characters (\), that define an existing path.

Description

The CHDIR command, when followed by an existing path, is used to set the working directory for the default or specified disk drive.

The *path* parameter consists of the name of an existing directory, optionally followed by the names of existing subdirectories, each separated from the next by a backslash character. If *path* begins with a backslash, CHDIR assumes that the first named directory is a subdirectory of the root directory; otherwise, CHDIR assumes that the first named directory is a subdirectory of the current directory. The special directory name **..** , which is an alias for the parent directory of the current directory, can be used as the path.

When CHDIR is entered alone or with only a drive letter followed by a colon, the full path of the current directory for the default or specified drive is displayed.

CD is simply an alias for CHDIR; the two commands are identical.

Examples

To change the current directory for the current (default) disk drive to the path \V2\SOURCE, type

```
C>CD \V2\SOURCE  <Enter>
```

To display the name of the current directory for the disk in drive D, type

```
C>CD D:  <Enter>
```

To return to the parent directory of the current directory, type

```
C>CD ..  <Enter>
```

Messages

Invalid directory
One of the directories in the specified path does not exist.

Invalid drive specification
An invalid drive letter was given or the named drive does not exist in the system.

CHKDSK

Check Disk Status

1.0 and later

External

Purpose

Analyzes the allocation of storage space on a disk and displays a summary report of the space occupied by files and directories.

Syntax

CHKDSK [*drive*:][*pathname*] [/F] [/V]

where:

drive	is the letter of the drive containing the disk to be analyzed, followed by a colon.
pathname	is the location and, optionally, the name of the file(s) to be checked for fragmentation; wildcard characters are permitted in the filename.
/F	repairs errors (versions 2.0 and later).
/V	"verbose mode," reports the name of each file as it is checked (versions 2.0 and later).

Description

The CHKDSK command analyzes the disk directory and file allocation table for consistency and reports any errors. If the /V switch is included in the command line, the name of each file processed is displayed as the disk is being analyzed.

After analyzing the disk, CHKDSK displays a summary of the disk and RAM space used and available. The disk-space report includes

- Total disk space in bytes
- Number of bytes allocated to hidden files
- Number of bytes contained in directories
- Number of bytes contained in user files
- Number of bytes contained in bad (unusable) sectors
- Number of available bytes on the disk

(Hidden files are files that do not appear in a directory listing. A bootable MS-DOS or PC-DOS disk always contains two hidden files—MSDOS.SYS and IO.SYS or IBMDOS.COM and IBMBIO.COM, respectively—that contain the operating system. A volume label, if present, counts as a hidden file. In addition, some application programs create hidden files for copy protection or other purposes.)

Directory errors detected by CHKDSK include

- Invalid pointers to data areas
- Bad file attributes in directory entries

- Damage to a portion of the directory that makes it impossible to check one or more paths
- Damage to an entire directory that makes the files contained in that directory inaccessible

File allocation table (FAT) errors detected by CHKDSK include

- Defective disk sectors in the FAT
- Invalid cluster (disk allocation unit) numbers in the FAT
- Lost clusters
- Cross-linking of files on the same cluster

If the /F switch is included in the command line, CHKDSK will attempt to repair errors in disk allocation and recover as much data as possible. Because repairs usually involve changes to the disk's file allocation table that may cause a loss of information, the user is prompted for confirmation. Lost clusters are collected into files in the root directory with names of the form FILE*nnnn*.CHK.

If the command line contains a file specification, CHKDSK will examine all files that match the specification and report on their fragmentation — that is, on whether or not their sectors are contiguous on the disk. (Fragmented files can degrade the performance of the system because of the time required to move the drive head back and forth across the disk to reach the various parts of the file.) Files on a floppy disk can be collected into contiguous sectors by copying them to an empty floppy disk. Files on a fixed disk can be collected into contiguous sectors by backing them all up to floppy disks, erasing all files and subdirectories on the fixed disk, and then restoring the files from the floppy disk.

Warning: CHKDSK should not be used on a network drive or on a drive created or affected by an ASSIGN, JOIN, or SUBST command.

Examples

To check the disk in the current drive, type

```
C>CHKDSK <Enter>
```

If CHKDSK finds no errors, a report such as the following is displayed:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free
```

Note that the line containing the volume name and creation date does not appear if the disk has not been assigned a volume name.

If CHKDSK finds errors, a message such as the following is displayed:

```
Errors found, F parameter not specified.
Corrections will not be written to disk.

10 lost clusters found in 3 chains.
Convert lost chains to files  (Y/N)?
```

A *Y* response at this point does not convert the lost chains to files; to do this, enter the CHKDSK command again with the /F switch specified.

To correct any allocation errors found by the CHKDSK command, type

```
C>CHKDSK /F  <Enter>
```

In this example, CHKDSK displays its usual report, followed by an error message:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free

10 lost clusters found in 3 chains.
Convert lost chains to files (Y/N) ?
```

A *Y* response causes CHKDSK to recover the lost chains of clusters into files in the root directory, giving the files the names FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, and so on. An *N* response causes CHKDSK to free the lost chains of clusters without saving the contents to files.

To check all files in the directory C:\SYSTEM with the extension .COM for fragmentation, type

```
C>CHKDSK C:\SYSTEM\*.COM  <Enter>
```

CHKDSK displays its usual report, followed by a list of fragmented files:

```
Volume HARDDISK    created Jun 8, 1986 9:34a

 21204992 bytes total disk space
    38912 bytes in 3 hidden files
   116736 bytes in 53 directories
 17055744 bytes in 715 user files
    20480 bytes in bad sectors
  3973120 bytes available on disk

   655360 bytes total memory
   566576 bytes free

C:\SYSTEM\ALUSQ.COM
   Contains 2 non-contiguous blocks.

C:\SYSTEM\EJECT.COM
   Contains 4 non-contiguous blocks.
```

Messages

. Does not exist.

or

.. Does not exist.
The **.** (alias for the current directory) or the **..** (alias for the parent directory) entry is missing.

filename* Is cross linked on cluster *n
Two or more files have been assigned the same cluster. Make a copy of both files on another disk and then delete them from the disk containing the error. One or both of the resulting files may contain information belonging to the other file.

***x* lost clusters found in *y* chains.**
Convert lost chains to files (Y/N)?
Clusters have been identified that are not assigned to any existing file. If the /F switch was included in the original command line, respond with *Y* to convert the lost clusters to files in the root directory of the disk with names of the form FILE*nnnn*.CHK. If desired, the recovered clusters can then be returned to the free-disk-space pool by erasing the .CHK files.

Allocation error, size adjusted.
The size of the file indicated in the disk directory is not consistent with the number of clusters allocated to the file. If the /F switch was included in the command line, the file is truncated to the size indicated in the disk directory.

All specified file(s) are contiguous.
The clusters belonging to the specified file(s) are allocated contiguously (without fragmentation).

Cannot CHDIR to *pathname*
tree past this point not processed.
The tree directory structure of the disk being checked cannot be traveled to the specified directory. This message indicates severe damage to the disk's directories or files.

Cannot CHDIR to root
Processing cannot continue.
In traversing the tree directory structure of the disk being checked, CHKDSK was unable to return to the root directory. This message indicates severe damage to the disk's directories or files.

Cannot CHKDSK a Network drive
The drive containing the disk to be checked has been assigned to a network.

Cannot CHKDSK a SUBSTed or ASSIGNed drive
The drive containing the disk to be checked has been substituted or assigned.

Cannot recover . entry, processing continued.
The special directory entry **.** (alias for the current directory) is defective.

Cannot recover .. entry,
Entry has a bad attribute

or

Cannot recover .. entry,
Entry has a bad link

or

Cannot recover .. entry,
Entry has a bad size
The special directory entry **..** (alias for the parent directory of the current directory) is defective due to a bad attribute, link, or size.

CHDIR .. failed, trying alternate method.
While checking the tree structure, CHKDSK was unable to return to the parent directory of the current directory. It will attempt to return to that directory by starting over at the root directory and searching again.

Contains *n* non-contiguous blocks.
The clusters assigned to the specified file are not allocated contiguously on the disk.

Directory is joined
CHKDSK cannot process directories that have been joined using the JOIN command. Use the JOIN /D command to unjoin the directories, then run CHKDSK again.

Directory is totally empty, no . or ..
The specified directory does not contain the usual aliases for the current and parent directories. This message indicates severe damage to the disk's directories or files. Delete the directory and recreate it.

Disk error reading FAT *n*

or

Disk error writing FAT *n*
One of the file allocation tables for the disk being checked contains a defective sector. MS-DOS will use the alternate FAT if one is available. It is advisable to copy all the files on the disk containing the defective sector to another disk.

Errors found, F parameter not specified.
Corrections will not be written to disk.
Errors were found on the disk being checked, but the /F switch was not included in the command line.

File allocation table bad drive *X*:
The disk is not an MS-DOS disk. Repeat CHKDSK with the /F option; if this message is displayed again, reformat the disk.

File not found.
CHKDSK was unable to find the specified file.

First cluster number is invalid, entry truncated.
The directory entry for the specified file contains an invalid pointer to the disk's data area. If the /F switch was included in the command line, the file is truncated to a zero-length file.

General Failure error reading drive *X*:
The format of the disk being checked is not compatible with MS-DOS or the disk has not been formatted for use by MS-DOS.

Has invalid cluster, file truncated.
The file directory contains an invalid pointer to the disk's data area. If the /F switch was included in the command line, the file is truncated to a zero-length file.

Incorrect DOS version
The version of CHKDSK is not compatible with the version of MS-DOS that is running.

Insufficient memory
Processing cannot continue.
The computer does not have enough memory to contain the tables necessary for CHKDSK to process the specified disk.

Insufficient room in root directory.
Erase files in root and repeat CHKDSK.
The root directory is full and does not have room for the entries for recovered files. Delete some files from the root directory of the disk being checked and rerun the CHKDSK program.

Invalid current directory
Processing cannot continue.
The directory structure of the disk is so badly damaged that the disk is unusable.

Invalid drive specification
The CHKDSK command contained an invalid disk drive.

Invalid parameter
One of the switches in the command line is invalid.

Invalid sub-directory entry.
The directory name specified in the command line does not exist or is invalid.

Path not found.
One of the directories in the path specified in the command line does not exist or is invalid.

Probable non-DOS disk
Continue (Y/N) ?
The disk being checked was not formatted by MS-DOS or the file allocation table has been severely damaged or destroyed.

Unrecoverable error in directory.
Convert directory to file (Y/N)?
The specified directory is damaged and unusable. If the /F switch was included in the original command line, respond with *Y* to convert the damaged directory to a file in the root directory of the disk with a name of the form FILE*nnnn*.CHK. If desired, the .CHK file can then be deleted. Any files that were previously reached through the damaged directory will be lost.

CLS

Clear Screen

2.0 and later

Internal

Purpose

Clears the video display.

Syntax

CLS

Description

The CLS command clears the video display and displays the current prompt.

In some implementations of MS-DOS, proper operation of the CLS command may require installation of the ANSI.SYS console driver with a *DEVICE=ANSI.SYS* command in the CONFIG.SYS file.

Examples

To clear the screen, type

```
C>CLS  <Enter>
```

To save the ANSI escape sequence used by the CLS command (ESC[2J) into a file named CLEAR.TXT, type

```
C>CLS > CLEAR.TXT  <Enter>
```

COMMAND

1.0 and later

Command Processor

External

Purpose

Loads a secondary copy of the MS-DOS default command processor.

Syntax

COMMAND [*drive*:][*path*] [*device*] [/E:*n*] [/P] [/C *string*]

where:

path is the name of the directory to be searched for COMMAND.COM when the transient portion needs to be reloaded; a drive letter can be included with versions 2.0 and later.

device is the name of a character device to be used instead of CON for the command processor's input and output (versions 2.0 and later).

/E:*n* is the initial size, in bytes, of the command processor's environment block (160–32768, default = 160) (version 3.2).

/P fixes the newly loaded command processor permanently in memory (versions 2.0 and later).

/C *string* causes the command processor to behave as a transient program and execute the command or program specified by *string* (versions 2.0 and later).

Description

The command processor is the module of the operating system that is responsible for issuing prompts to the user, interpreting commands, loading and executing transient application programs, and interpreting batch files. The file COMMAND.COM contains the MS-DOS default command processor, or shell. It is ordinarily loaded from the root directory of the system disk when the system is turned on or restarted, unless the SHELL command is used in the CONFIG.SYS file to specify another command processor or an alternate location for COMMAND.COM.

With versions 1.x, COMMAND.COM is invoked by the COMMAND command in response to a shell prompt or within a batch file. A second copy of the resident portion of COMMAND.COM is loaded and the memory occupied by the original resident portion is lost. The second copy of the transient portion simply overlays the original transient portion. (Versions 1.x of COMMAND support no switches or other parameters and any specified in the command line are ignored.) With versions 2.0 and later, the new copy of COMMAND.COM is loaded *in addition to* the parent command processor and serves as a secondary command processor.

The *path* parameter specifies the location of the COMMAND.COM file that is used to reload the transient part of the command processor if it is overlaid by application programs. If absent, *path* defaults to the root directory of the system (startup) disk.

The *device* parameter allows a character device other than CON to be used by the command processor for input and output. For example, use of AUX as the *device* parameter allows a personal computer to be controlled from a terminal attached to a serial port, instead of from the usual built-in keyboard and memory-mapped video display.

The secondary copy of COMMAND.COM ordinarily remains in memory and serves as the active command processor until an EXIT command is entered. If a /P switch is used with the COMMAND command, the new copy of COMMAND.COM is fixed in memory and the EXIT command is disabled. In such cases, the memory occupied by previously loaded copies of COMMAND.COM is simply lost.

The /E:*n* switch controls the size of the environment block initially allocated for the command processor. The default size of the block is 160 bytes, but the /E:*n* switch allows the initial allocation to be as large as 32768 bytes. This switch is frequently used when *COMMAND.COM* is included in the SHELL command in the CONFIG.SYS file.

When the /C *string* switch is included in the command line, followed by a string designating a command or program name, the new copy of COMMAND.COM carries out the operation specified by *string* and then exits, returning control to its parent command processor or other program. This option allows a batch file to invoke another batch file and then resume its own execution. (If a batch file names another batch file directly without using COMMAND /C *string* as an intermediary, the first batch file is terminated.) Note that when the /C *string* switch is used in combination with other switches, it must be the last switch in the command line.

A secondary copy of COMMAND.COM always inherits a copy of the environment of the command processor or other program that loaded it. Changes made to the new COMMAND.COM's environment with a SET, PROMPT, or PATH command do not affect the environment of any previously loaded program or command processor.

Examples

To execute the batch file MENU2.BAT from the batch file MENU1.BAT and then resume execution of MENU1.BAT, include the following line in MENU1.BAT:

```
COMMAND /C MENU2
```

To cause COMMAND.COM to be loaded from the directory \SYSTEM on drive C rather than from the root directory and to allocate an initial environment block of 1024 bytes, include the following line in the CONFIG.SYS file:

```
SHELL=C:\SYSTEM\COMMAND.COM C:\SYSTEM /P /E:1024
```

Messages

Bad or missing command interpreter
The file COMMAND.COM is not present in the root directory of the system disk and no SHELL command is present to specify an alternate command processor file or location, or the location specified for COMMAND.COM in a SHELL command is not correct. This message may also be seen if COMMAND.COM is moved from its original location after the system is booted.

Invalid device
The character device specified in the command line is not valid or does not exist.

Invalid environment size specified
The value supplied with the /E:*n* switch was less than 160 bytes or greater than 32768 bytes.

COMP

Compare Files

IBM

External

Purpose

Compares two files or sets of files. This command is available only with PC-DOS.

Syntax

COMP [*primary*] [*secondary*]

where:

primary is the name of the file to be compared against and can be preceded by a drive and/or path; wildcard characters are permitted in the filename.

secondary is the name of the file to be compared with *primary* and can be preceded by a drive and/or path; wildcard characters are permitted in the filename.

Description

The COMP command compares one file or set of files with another. As each pair of files is compared, the program reports whether the files are identical, different in size, or the same size but different in content.

The *primary* and *secondary* parameters can be any combination of drive, path, and filename, optionally including wildcards to allow sets of files to be compared. (With versions 1.x, using wildcards does not cause multiple file comparisons — only the first secondary file whose name matches the first primary filename is compared.) The *primary* parameter generally designates the specific files to be compared; the *secondary* parameter is usually only a drive and/or path, except when the files being compared have different names or extensions.

If both *primary* and *secondary* are omitted from the command line, the COMP program prompts for them interactively. If *primary* is given as a drive or path only, COMP assumes *.* to be the primary file. If *secondary* is given as a drive or path only, COMP compares all files on that drive or path whose filenames match those of the primary files.

The COMP command is included only with PC-DOS. MS-DOS versions 2.0 and later provide a similar function in the FC command, which also displays the differences between files.

Examples

To compare the file MYFILE.DAT on the disk in drive A with the file LEDGER.DAT on the disk in drive B, type

```
C>COMP A:MYFILE.DAT B:LEDGER.DAT  <Enter>
```

To compare all the files in the current directory of the disk in drive A with the corresponding files in the current directory of the disk in drive D, type

```
C>COMP A:*.* D:  <Enter>
```

To compare all the files with the extension .ASM in the directory C:\SOURCE with the corresponding files with extension .BAK on the disk in drive B, type

```
C>COMP C:\SOURCE\*.ASM B:*.BAK  <Enter>
```

Messages

10 mismatches - ending compare
The primary and secondary files are the same size but have more than 10 internal differences. The compare operation on this pair of files is aborted and COMP proceeds to the next pair of files, if any.

filename* and *filename
This informational message shows the full filenames of the two files currently being compared.

Access Denied
An attempt was made to compare a locked file.

Cannot compare file to itself
An attempt was made to compare a file with itself.

Compare error at OFFSET *nn*
File 1 = *nn*
File 2 = *nn*
This informational message itemizes the first 10 differences in data between the two files being compared (if the files are the same size), displaying the file offset and the differing bytes from each file as hexadecimal values.

Compare more files (Y/N)?
After all specified pairs of files have been compared, the COMP program allows the entry of another pair of file specifications. Respond with *Y* or press Enter to continue; respond with *N* to terminate the COMP program.

Enter 2nd file name or drive id
If the secondary filename was not specified in the COMP command, this message prompts the user to enter it (or a path, if the secondary file has the same name as the primary file).

Enter primary file name
If no parameter was entered after COMP, this message prompts the user to enter the primary filename. If a drive or path is specified, COMP assumes *.* for the primary filename.

EOF mark not found
The last byte at the logical end of the file was not a Control-Z character (^Z, or 1AH). This message is commonly seen during comparison of two files that are not ASCII text files, such as executable program files.

Files compare OK
The files being compared were the same length and contained identical data.

File not found
The specified filename was invalid or the file does not exist.

Files are different sizes
The two files being compared have different sizes recorded in the directory. No comparison on the data within the files is attempted.

File sharing conflict
COMP is unable to compare the two current files because one of the files is in use by another process.

Incorrect DOS version
The version of COMP is not compatible with the version of PC-DOS that is running.

Insufficient memory
The available system memory is insufficient to run the COMP program.

Invalid drive specification
The drive specification in *primary* or *secondary* is invalid or does not exist.

Invalid path
The path or directory in *primary* or *secondary* is invalid or does not exist.

Too many files open
No more system file handles are available. Increase the value of the FILES command in the CONFIG.SYS file and restart the system.

CONFIG.SYS

2.0 and later

System Configuration File

Purpose

Allows the user to configure the operating system.

Description

The CONFIG.SYS file is an ASCII text file that MS-DOS processes during initialization (when the system is turned on or restarted). It allows the user to configure certain aspects of the operating system, such as the number of internal disk buffers allocated, the number of files that can be open at one time, the formats for date and currency, and the name and location of the executable file containing the command processor. CONFIG.SYS can also contain commands that extend the system with installable device drivers for terminal emulation, virtual disks or RAMdisks, extended or expanded memory, and other special peripheral devices.

The CONFIG.SYS file can be created or modified with EDLIN or with any other editor or word processor that can produce ordinary ASCII text files (nondocument files) and save them to disk. The CONFIG.SYS file must be in the root directory of the disk that is used to start the operating system in order for it to be processed during system initialization. When changes are made to the CONFIG.SYS file, they do not take effect until the system is restarted.

Commands in the CONFIG.SYS file take the form

command [=] *value*

(Note that the equal sign is optional; any other valid MS-DOS separator [semicolon, tab, or space] can be used instead.) The commands supported are

Command	**Action**
BREAK	Controls extended checking for Control-C.
BUFFERS	Specifies the number of internal disk-sector buffers available for use by MS-DOS when reading from or writing to a disk.
COUNTRY	Controls date, time, and currency formatting.
DEVICE	Specifies the filename of an installable device driver.
DRIVPARM	Redefines the default characteristics of the resident MS-DOS block device(s) (version 3.2).
FCBS	Specifies the maximum number of simultaneously open file control blocks (versions 3.0 and later).

(more)

Command	Action
FILES	Specifies the maximum number of simultaneously open files controlled by handles.
LASTDRIVE	Sets the highest valid drive letter (versions 3.0 and later).
SHELL	Specifies the filename (and optionally the drive and/or path) of the system command processor.
STACKS	Sets the number and size of stack frames for the system.

Each of these commands is discussed in detail on the following pages.

Message

Unrecognized command in CONFIG.SYS
A command in the CONFIG.SYS file was misspelled, an invalid parameter was used, or a command was included that is not compatible with the version of MS-DOS that is running. Correct the CONFIG.SYS file and restart the system.

CONFIG.SYS: BREAK

2.0 and later

Configure Control-C Checking

Purpose

Sets or clears MS-DOS's internal flag for Control-C checking.

Syntax

BREAK=ON ¦OFF

Description

Pressing Ctrl-C or Ctrl-Break while a program is running ordinarily terminates the program, unless the program itself contains instructions that disable MS-DOS's Control-C handling. As a rule, MS-DOS checks the keyboard for a Control-C only when a character is read from or written to a character device (keyboard, screen, printer, or auxiliary port). Therefore, if a program executes for long periods without performing such character I/O, detection of the user's entry of a Control-C may be delayed. The BREAK=ON command causes MS-DOS to also check the keyboard for a Control-C at the time of each system call (which slows the system somewhat); the BREAK=OFF command disables such extended Control-C checking. The default setting for BREAK is off.

Extended Control-C checking can also be enabled or disabled at the command prompt with the interactive form of the BREAK command whenever the system is running.

Example

To enable extended Control-C checking during MS-DOS disk operations, insert the line

```
BREAK=ON
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
The setting supplied for the BREAK command was not ON or OFF. Correct the CONFIG.SYS file and restart the system.

CONFIG.SYS: BUFFERS

2.0 and later

Configure Internal Disk Buffers

Purpose

Sets the number of MS-DOS's internal disk buffers.

Syntax

BUFFERS=*nn*

where:

nn is the number of buffers (1–99, default = 2; default = 3 for IBM PC/AT and compatibles).

Description

MS-DOS maintains a set of internal buffers (sometimes referred to as a disk cache) in which it keeps copies of the sectors most recently read from or written to the disk. Whenever a program requests a disk read, MS-DOS first searches the disk buffers to determine whether a copy of the disk sector containing the required data is already present in RAM. If the sector is found, the actual disk access is bypassed. This technique can significantly improve the overall performance of the disk operating system.

By using the BUFFERS command in the CONFIG.SYS file, the user can control the number of buffers in MS-DOS's disk cache. The default number of buffers is 2 for an IBM PC, PC/XT, or compatible and 3 for an IBM PC/AT or compatible. The optimum number of buffers varies, depending in part on the characteristics and types of the system disk drives, the types of application programs used on the system, the number and levels of subdirectories in the file structure, and the amount of RAM in the system.

If the system has only floppy-disk drives, the default setting of 2 buffers is sufficient. If the system includes a fixed disk, increasing the number of buffers to 10 or so typically speeds up overall system operation. Configuring the system for too many buffers, however, can actually degrade the performance of the system.

Increases in the number of buffers should be tailored to the type of application most frequently used. For example, allocation of extra disk buffers will not improve the performance of programs that use primarily sequential file access but may considerably enhance the execution times of programs that perform random access on a relatively small number of disk records (such as the index for a database file). In addition, if the system has many subdirectories organized in several levels, increasing the number of buffers can significantly increase the speed of disk operations.

The ideal number of buffers for a given system is difficult to predict because of the interactions between the access time of the disk, the speed of the central processing unit, and the

RAM requirements and disk access behavior of the mix of application programs. However, a reasonably optimal number of buffers can be quickly estimated experimentally by increasing the number of buffers in increments of five or so, restarting the system, performing some simple timing tests on the most frequently used application programs, and observing at what number of buffers system performance begins to degrade.

Example

To allocate 20 internal disk buffers, insert the line

```
BUFFERS=20
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
The value supplied for the BUFFERS command was not a number in the range 1 through 99.

CONFIG.SYS: COUNTRY

2.1 and later

Set Country Code

Purpose

Configures MS-DOS's internationalization support for a specific country.

Syntax

COUNTRY=*nnn*

where:

nnn is the international telephone dialing prefix for the country (001–999, default = 001):

Australia	061
Belgium	032
Denmark	045
Finland	358
France	033
Israel	972
Italy	039
Netherlands	031
Norway	047
Spain	034
Sweden	046
Switzerland	041
United Kingdom	044
USA	001
West Germany	049

Note: In versions 2.x (except 2.0), *nnn* is 01 through 99. Individual computer manufacturers determine the specific codes supported by their versions of MS-DOS.

Description

The COUNTRY command enables the user to tailor MS-DOS's date, time, and currency displays for a specific country. This capability, termed internationalization support, is achieved through use of a country code that controls the contents of the table MS-DOS uses to format these displays (including numeric separators). (The internationalization table is made available to application programs through Interrupt 21H Function 38H.) Beginning with version 3.0, PC-DOS also supports the COUNTRY command.

Example

In West Germany, the format for the date is *dd.mm.yy*. To configure MS-DOS to use this date format, insert the line

```
COUNTRY=049
```

into the CONFIG.SYS file and restart the system.

Message

Invalid country code
The specified country code is not supported by the version of MS-DOS that is running.

CONFIG.SYS: DEVICE

2.0 and later

Install Device Driver

Purpose

Loads and links an installable device driver into the operating system during initialization.

Syntax

DEVICE=[*drive*:][*path*]*filename* [*options*]

where:

filename	is the name of the device-driver file, optionally preceded by a drive and/or path.
options	specifies any switches or other parameters needed by the device driver; the DEVICE command itself has no switches.

Description

Device drivers are the modules of the operating system that control the interface between the operating system and peripheral devices such as disk drives, magnetic-tape drives, CRT terminals, and printers.

As supplied, MS-DOS already contains device drivers for the keyboard, video display, serial port, printer, real-time clock, and disk devices. Device drivers for additional peripheral devices can be linked into the operating system by adding a DEVICE command to the CONFIG.SYS file, placing the file containing the device driver on the system startup disk (or at the location specified by the *drive*: and/or *path* parameter), and restarting the computer.

If a drive other than the one containing the system disk is named as the location of the device driver, that drive must either be accessible via the system's default disk driver or be a drive configured with a previous DEVICE command.

Most OEM implementations of version 3.2 provide three installable device drivers: ANSI.SYS, which allows the video display and keyboard to be controlled by ANSI standard escape sequences; DRIVER.SYS, which supports external disk drives; and RAMDRIVE.SYS (VDISK.SYS with PC-DOS), which uses a portion of the machine's RAM to emulate a disk drive. *See* USER COMMANDS: ANSI.SYS; DRIVER.SYS; RAMDRIVE.SYS; VDISK.SYS.

Many manufacturers of add-on products for MS-DOS machines (such as network interfaces or Lotus/Intel/Microsoft Expanded Memory boards) also supply installable device drivers for use with their hardware. For information concerning these drivers, see the product manufacturer's user's manual.

Examples

To load the ANSI standard console driver, insert the line

```
DEVICE=ANSI.SYS
```

into the CONFIG.SYS file, place the file ANSI.SYS in the root directory of the system disk, and restart the system.

To load the RAMDRIVE.SYS driver located in the \DRIVERS directory on the disk in drive A, configuring it for 1024 KB in extended memory, insert the line

```
DEVICE=A:\DRIVERS\RAMDRIVE.SYS 1024 /E
```

into the CONFIG.SYS file and restart the system.

Messages

Bad or missing *filename*
The filename specified in the DEVICE command is invalid or does not exist or the file does not contain a valid MS-DOS installable device driver.

Sector size too large in file *filename*
The specified installable device driver uses a sector size that is larger than the sector size used by any of the system's default disk drivers. Such a driver cannot be used because MS-DOS's internal disk buffers will not be large enough to hold a sector read from the device.

CONFIG.SYS: DRIVPARM

3.2

Set Block-Device Parameters

Purpose

Alters the system's list of characteristics for an existing block device.

Syntax

DRIVPARM=/D:*n* [/C] [/F:*n*] [/H:*n*] [/N] [/S:*n*] [/T:*n*]

where:

/D:*n* is the drive number (0–255; 0 = A, 1 = B, etc.) and must always be the first switch in the command line.

/C indicates that the device provides door-lock-status support.

/F:*n* is a form-factor index from the following table (default = 2 if the DRIVPARM command is present but this switch is omitted):

0	320 KB or 360 KB
1	1.2 MB
2	720 KB
3	8-inch single-density floppy disk
4	8-inch double-density floppy disk
5	Fixed disk
6	Tape drive
7	Other

/H:*n* is the number of read/write heads (1–99).

/N indicates that the block device is not removable.

/S:*n* is the number of sectors per track (1–99).

/T:*n* is the number of tracks per side (1–999).

Note: The DRIVPARM command must not be used to specify device characteristics that the device driver is not capable of supporting.

Description

Whenever the device driver for a block device such as a disk drive or magnetic-tape drive performs input or output, it refers to an internal table of characteristics for the device that allows it to convert logical addresses to physical addresses. The DRIVPARM command modifies the default MS-DOS values in the table of characteristics for a particular block device during system initialization (when the computer is turned on or restarted). Multiple DRIVPARM commands, each modifying the characteristics of a different block device, can be included in the same CONFIG.SYS file. Any characteristics not specifically altered in the DRIVPARM command for a particular device retain their original values, except for /F:*n*, which defaults to 2.

DRIVPARM commands that alter the characteristics for block devices controlled by *installable* device drivers must follow the DEVICE command that loads the device driver itself.

Example

Assume that drive B is a floppy-disk drive originally configured for 40 tracks with 8 sectors per track. To reconfigure the drive to read or write 80 tracks of 9 sectors each, insert the line

```
DRIVPARM=/D:1 /S:9 /T:80
```

into the CONFIG.SYS file and restart the system. For this command to be valid the drive must be capable of supporting these parameters.

Message

Unrecognized command in CONFIG.SYS
An invalid parameter was specified in a DRIVPARM command.

CONFIG.SYS: FCBS

3.0 and later

Set Maximum Open Files Using File Control Blocks (FCBs)

Purpose

Configures the maximum number of files that can be open concurrently using file control blocks (FCBs). This command has no practical effect unless either the file-sharing support module SHARE.EXE or networking support has been loaded.

Syntax

FCBS=*m*,*p*

where:

m is the maximum number of files that can be open concurrently using FCBs (1–255, default = 4).

p is the number of files opened with FCBs that are protected against automatic closure (0–*m*, default = 0).

Description

MS-DOS supports two methods of file access: file control blocks and file handles. A file control block is a data structure that stores information about an open file. It resides inside an application program's memory space and is accessed by both MS-DOS and the application. (*See* USER COMMANDS: CONFIG.SYS: FILES for information on file handles.)

In a network environment, a large number of active FCBs or improper use of FCBs by an application can seriously degrade the performance of the network as a whole. Consequently, MS-DOS versions 3.0 and later provide the FCBS command to enable the user to limit the number of files that can be open concurrently using FCBs if either the file-sharing support module SHARE.EXE (*see* USER COMMANDS: SHARE) or network support has been loaded. If an application program attempts to exceed the specified number of files, MS-DOS closes the file with the least recently used FCB.

The *p* parameter in the FCBS command line allows the user to protect files from unilateral closure by MS-DOS. The value of *p* is the number of files, counting from the first file opened using an FCB, that cannot be closed automatically.

If the current value of FCBS is *4,0* (the default) when the file-sharing module SHARE.EXE or network support is loaded, MS-DOS automatically increases the maximum number of files that can be open concurrently to 16 and the number of files protected against automatic closure to 8. (When multiple FCBs refer to the same file, the file is counted only once.)

Examples

To set the maximum number of files that can be concurrently open using FCBs to 10 and protect none of the FCB-opened files against automatic closure by MS-DOS, insert the line

```
FCBS=10,0
```

into the CONFIG.SYS file and restart the system.

To set the maximum number of files that can be concurrently open using FCBs to 8 but protect the first 4 FCB-opened files against automatic closure by MS-DOS, insert the line

```
FCBS=8,4
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified as one of the parameters in the FCBS command.

CONFIG.SYS: FILES

2.0 and later

Set Maximum Open Files Using Handles

Purpose

Configures the maximum number of files and/or devices that can be open concurrently using file handles.

Syntax

FILES=*n*

where:

n is the maximum number of files and devices that can be open concurrently using file handles (8–255, default = 8).

Description

MS-DOS supports two methods of file access: file handles and file control blocks (FCBs). During initialization, MS-DOS allocates a data structure that holds information about files and/or devices opened with the handle, or extended-file-management, function calls. This structure resides inside the operating system's memory space and is accessed only by MS-DOS. (*See* USER COMMANDS: CONFIG.SYS: FCBS.) The default size of this data structure allows 8 files and/or devices to be open concurrently using the file-handle functions. The FILES command enables the user to change the size of the data structure. (Note that increasing the size of the data structure decreases the amount of RAM available to application programs.)

The FILES command controls the maximum number of files and/or devices opened with handles for *all* active processes in the system combined. The limit on the number of files and/or devices opened for a single process using handles is 20 or the number of entries in the allocated data structure, whichever is less. Five of the 20 possible handles for a given process are automatically assigned to standard input, standard output, standard error, standard auxiliary, and standard list. However, since standard input, standard output, and standard error all default to the same device (CON), only three of the allocated data-structure entries are actually expended. In addition, the preassigned standard device handles for a process can be closed and reused for other files and devices, if necessary.

Example

To set the maximum number of files and/or devices that can be concurrently open using the handle functions to 20, insert the line

```
FILES=20
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified in the FILES command.

CONFIG.SYS: LASTDRIVE

3.0 and later

Set Highest Logical Drive

Purpose

Defines the highest letter that MS-DOS will recognize as a disk-drive code.

Syntax

LASTDRIVE=*drive*

where:

drive is a single letter (A–Z).

Description

MS-DOS block devices (floppy-disk drives, fixed-disk drives, and magnetic-tape drives) are referred to by logical drive codes consisting of a single letter from A through Z. In most MS-DOS systems, drives A and B are floppy-disk drives, drive C is a fixed disk, and drives D and above are such devices as additional fixed disks, RAMdisks, or network volumes. In some cases, a single physical drive (such as a very large fixed disk) is partitioned into two or more logical drives, each of which is assigned a drive letter.

MS-DOS validates the drive code in a command or filename before carrying out a command. In the default case, MS-DOS recognizes a maximum of five drives (A–E), depending on the total number of default devices and devices incorporated into the system using installable device drivers. (MS-DOS does not consider a drive letter valid unless it refers to a physical or logical device.) The LASTDRIVE command configures MS-DOS to accept additional drive codes, to a total of 26 (A–Z). This also makes it possible to use fictitious drive letters with the SUBST command to assign a drive letter to a subdirectory.

If the letter code for a LASTDRIVE command specifies fewer drives than are physically present in the system (including installed device drivers), MS-DOS uses the actual number of physical drives.

Example

To configure MS-DOS to recognize a maximum of eight logical disk drives, insert the line

```
LASTDRIVE=H
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An illegal value was specified in the LASTDRIVE command.

CONFIG.SYS: SHELL

2.0 and later

Specify Command Processor

Purpose

Defines the name and, optionally, the location of the file that contains the operating system's command processor.

Syntax

SHELL=[*drive*:][*path*]*filename* [*options*]

where:

filename is the name of the file containing the command processor, optionally preceded by a drive and/or path.

options specifies any switches and other parameters needed by the designated command processor; the SHELL command itself has no switches.

Description

The command processor, or shell, is the user's interface to the operating system. It is responsible for parsing and carrying out the user's commands, including the loading and execution of other programs from the disk. MS-DOS uses the SHELL command in the CONFIG.SYS file to locate and load the command interpreter for the system during its initialization process.

The default shell for MS-DOS is the file COMMAND.COM. This file is loaded by MS-DOS from the root directory of the system disk if no SHELL command is found in the CONFIG.SYS file or if no CONFIG.SYS file exists.

The most common use of the SHELL command is simply to advise MS-DOS that COMMAND.COM is stored in a location other than the root directory; MS-DOS then sets the COMSPEC variable in the environment block to COMMAND.COM, preceded by the location specified in the SHELL command. (This can be verified by typing the SET command at the command prompt.) Another common use of SHELL is to specify switches or other parameters for COMMAND.COM itself (*see* USER COMMANDS: COMMAND).

Example

To specify the file VISUAL.COM in the root directory of drive C as the system's command processor, insert the line

```
SHELL=C:\VISUAL.COM
```

into the CONFIG.SYS file and restart the system.

Message

Bad or missing command interpreter
The path or filename in the SHELL command is invalid or the file does not exist.

CONFIG.SYS: STACKS

3.2

Configure Internal Stacks

Purpose

Defines the number and size of stacks for system interrupt handlers.

Syntax

STACKS=*number,size*

where:

number	is the number of stacks allocated for use by interrupt handlers (8–64, default = 9).
size	is the size of each stack in bytes (32–512, default = 128).

Description

Each time certain hardware interrupts occur (02H, 08–0EH, 70H, and 72–77H), MS-DOS version 3.2 switches to an internal stack before transferring control to the handler that will service the interrupt. In the case of nested interrupts, MS-DOS checks to ensure that both interrupts do not get the same stack. After the interrupt has been processed, the stack is released. This protects the stacks owned by application programs or system device drivers from overflowing when several interrupts occur in rapid succession.

The STACKS command configures the number and size of internal stacks available for interrupt handling and thus controls the number of interrupts that can exist only partially processed while still allowing another interrupt to occur.

The *number* parameter sets the number of internal stacks to be allocated; *number* must be in the range 8 through 64. The *size* parameter is the number of bytes allocated per stack frame; *size* must be in the range 32 through 512.

If too many interrupts occur too quickly and the pool of internal stack frames is exhausted, the system halts with the message *Internal Stack Overflow*. Increasing the *number* parameter in the STACKS command usually corrects the problem.

Example

To configure 10 stacks of 256 bytes each for use by interrupt handlers, insert the line

```
STACKS=10,256
```

into the CONFIG.SYS file and restart the system.

Message

Unrecognized command in CONFIG.SYS
An invalid number was specified in the STACKS command.

COPY

1.0 and later

Copy File or Device

Internal

Purpose

Copies one or more files from one disk, directory, or filename to another. Can also copy files to or from character devices.

Syntax

COPY *source* [/A] [/B] [+*source* [/A] [/B]...] [*destination*] [/A] [/B] [/V]

where:

source	is the names of the file(s) to be copied, optionally preceded by a drive and/or path; wildcard characters are permitted in filenames. The source can also be a device.
destination	is the location and, optionally, the name(s) for the copied file(s) and can be preceded by a drive; wildcard characters are permitted in the filename. The destination can also be a device.
/A	indicates that the previous file is an ASCII text file.
/B	indicates that the previous file is a binary file.
/V	performs read-after-write verification of destination file(s).

Description

The COPY command copies one or more source files to one or more destination files. When multiple files are copied, the name of each source file is displayed as it is processed. The COPY command can also be used to send the contents of a file to a character device or to copy input from a character device into a file.

The *source* parameter identifies the file or files to be copied. It can consist of any combination of drive, path, and filename or it can be a device name. If a path without a filename is specified, all files in the named directory are copied. Several source files can be concatenated into a single destination file by placing a + operator between their names; if the source filename contains a wildcard but the destination name does not, all the source files are concatenated into the specified destination.

Warning: When multiple source files are concatenated into a destination file with the same name as one of the source files, that filename should be specified as the *first* source file. Otherwise, the contents of the source file will be destroyed before the file is copied.

When a device is specified as the source, it is usually the console (CON), for copying keyboard input to a file or another device. Keyboard input is terminated by pressing Ctrl-Z or F6 (on IBM PCs or compatibles) and then the Enter key.

The *destination* parameter also can consist of any combination of drive, path, and filename or be a device name. Unless the source files are being renamed as part of the operation, *destination* is usually simply a drive and/or path specifying where to place the copied files. If no destination is specified, the source file is copied to a file with the same name in the current directory of the default disk drive; if the source file in this case is itself in the current directory of the current drive, an error message is displayed and the copy operation is aborted. If files are being concatenated and no destination is specified, the source files are copied sequentially into one file in the current directory with the same name as the first source file. If the first source file already exists, the second file and any additional specified files are appended sequentially to the first source file.

The /A and /B switches control the manner in which the COPY command operates on a file. Both switches affect the file specification immediately preceding them and any subsequent file specifications in the command until another /A or /B switch is encountered, at which point the new /A or /B switch takes effect for the file immediately preceding it and for any subsequent files.

The /A switch indicates that a file is an ASCII text file. When the /A switch is applied to a source file, the file is copied up to, but not including, the first Control-Z (^Z) character in the file. When the /A switch is applied to a destination file, a Control-Z character is appended by the COPY command as the last character of the new file.

The /B switch indicates a binary file. When /B is applied to a source file, the exact number of bytes in the original file are copied without regard to Control-Z or any other control characters. When the /B switch is applied to a destination file, no Control-Z character is appended to the newly created file.

The default values for the /A and /B switches for file-to-file copies are /A when source files are being concatenated and /B otherwise. When a file is being copied to or from a character device, the /A switch is the default.

The /V switch causes a read-after-write verification of each block of the destination file. Its effect is equivalent to that of the VERIFY ON command. No comparison is made between the source and destination files — the /V switch simply causes MS-DOS to verify that the destination file has been written correctly.

Examples

To copy the file REPORT.TXT from the root directory of the disk in drive B to a file named FINAL.RPT in the \WP\DOCS directory on the current drive, type

```
C>COPY B:\REPORT.TXT \WP\DOCS\FINAL.RPT  <Enter>
```

To make a copy of the file A:\V2\SOURCE\MENUMGR.C in the current directory of the current drive, type

```
C>COPY A:\V2\SOURCE\MENUMGR.C  <Enter>
```

To copy all files with the extension .DOC in the current directory of the disk in drive A to files with the same filenames but a .TXT extension in the current directory of the current drive, type

```
C>COPY A:*.DOC *.TXT  <Enter>
```

To combine the files PROLOG.C, MENUMGR.C, and EPILOG.C in the current directory of the current drive into a single file named VISUAL.C in the current directory of the current drive, type

```
C>COPY PROLOG.C+MENUMGR.C+EPILOG.C VISUAL.C  <Enter>
```

To append the files MENUMGR.C and EPILOG.C to an existing file named PROLOG.C in the current directory of the current drive, type

```
C>COPY PROLOG.C+MENUMGR.C+EPILOG.C  <Enter>
```

To copy the file MENUMGR.MAP in the current directory of the current drive to the system printer, type

```
C>COPY MENUMGR.MAP PRN  <Enter>
```

To copy input from the keyboard (CON) to a file named MENU.BAT in the current directory of the current drive, type

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

Text subsequently entered from the keyboard is placed into the file MENU.BAT until a Ctrl-Z or F6 is pressed.

To copy all files in the \MEMOS directory on the current drive to the \ARCHIVE directory on the disk in drive B, type

```
C>COPY \MEMOS\*.* B:\ARCHIVE  <Enter>
```

or

```
C>COPY \MEMOS B:\ARCHIVE  <Enter>
```

Messages

***n* File(s) copied**
This informational message is displayed at the completion of a COPY command and indicates the total number of source files processed.

Cannot do binary reads from a device
The COPY command specified a copy from a character device in binary mode. Reenter the command without a /B switch.

Content of destination lost before copy
One of the source files specified as a destination file was overwritten prior to completion of the copy. When the destination name is the same as one of the source names, that file should be specified as the first source file.

File cannot be copied onto itself
The source directory and filename of a file being copied are the same as the destination directory and filename.

File not found
A file specified in the COPY command is invalid or does not exist.

Invalid directory
A directory specified in the COPY command is invalid or does not exist.

CTTY

2.0 and later

Assign Standard Input/Output Device

Internal

Purpose

Specifies the character device to be used as standard input and output.

Syntax

CTTY *device*

where:

device is the logical character-device name.

Description

MS-DOS ordinarily uses the computer's built-in keyboard and screen (CON) as standard input and output. The CTTY command allows another character device to be assigned instead.

CTTY allows MS-DOS commands to be issued from a terminal attached to the computer's serial port or from another custom device with a screen and keyboard. Although PRN and NUL are valid MS-DOS device names, they should not be used with this command, as they have no input capability.

Programs that do not use MS-DOS function calls to perform their input and output will not be affected by the CTTY command. Microsoft BASIC is an example of such a program.

Examples

To use a terminal connected to the serial port as standard input and output for programs, type

```
C>CTTY AUX  <Enter>
```

To reinstate the normal keyboard and video display (CON) as standard input and output for programs, type

```
C>CTTY CON  <Enter>
```

on the currently assigned console device.

Message

Invalid device
The specified device is not a legal character-device name or does not exist in the system.

DATE

1.0 and later

Set Date

Internal

Purpose

Sets or displays the system date.

Syntax

DATE *mm-dd-yy*

or

DATE *mm/dd/yy*

or

DATE *mm.dd.yy* (versions 3.0 and later)

where:

mm	is the month (1–12).
dd	is the day (1–31).
yy	is the year (80–99 or 1980–1999; 80–79 or 1980–2079 with versions 3.0 and later).

Description

All computers that run MS-DOS have as part of their hardware configuration a timer, or clock, that maintains the current system date and time. Among other uses, the current date and time are inserted into a file's directory entry when the file is created or modified.

The DATE command allows the user to display or modify the current date that is being maintained by the system's real-time clock. The command is executed automatically by MS-DOS when the system is initialized, unless there is an AUTOEXEC.BAT file on the system disk, in which case DATE is executed only if it is included in the file.

A date entered using the DATE command does not permanently change the system date; the newly entered date will be lost when the system is turned off or reset. On IBM PC/ATs and compatibles, which have a built-in battery-backed clock/calendar, the system setup program (found on the Diagnostics for IBM Personal Computer AT disk or equivalent) must be used to permanently alter the date stored in the machine. On IBM PCs, PC/XTs, and compatibles equipped with add-on cards containing battery-backed clock/calendar circuitry, it is generally necessary to run a time/date installation program (included with the card) when the system is turned on to set the system date and time from the clock/calendar on the card. The DATE command usually has no effect on these card-mounted clock/calendars.

The order of the day, month, and year in the DATE command depends on the country code, which is set with the COUNTRY command in the CONFIG.SYS file. The format shown here is for the USA.

Examples

To set the system date to October 15, 1987, type

```
C>DATE 10-15-87  <Enter>
```

or

```
C>DATE 10/15/87  <Enter>
```

or

```
C>DATE 10.15.87  <Enter>
```

To display the current system date, type

```
C>DATE  <Enter>
```

and MS-DOS will respond in the form

```
Current date is Thu 10-15-1987
Enter new date (mm-dd-yy):
```

To leave the date unchanged, press the Enter key.

Messages

Current date is *day mm-dd-yyyy*
Enter new date (mm-dd-yy):
This informational message and prompt are displayed when MS-DOS is started and there is no AUTOEXEC.BAT file on the system disk, when the DATE command is entered alone, or when the DATE command is included in the AUTOEXEC.BAT file.

Invalid date
Enter new date (mm-dd-yy):
The date entered in the command line or in response to the prompt from the DATE command was not formatted properly or was invalid.

DEL or ERASE

Delete File

1.0 and later

Internal

Purpose

Deletes a file or set of files. DEL and ERASE are synonymous.

Syntax

DEL [*drive*:][*path*]*filename*

or

ERASE [*drive*:][*path*]*filename*

where:

filename is the name of the file(s) to be deleted, optionally preceded by a drive and/or path; wildcard characters are permitted in the filename.

Description

The DEL command marks the directory entry for the specified file as deleted and frees the disk sectors occupied by the file. If the command line ends with *.* or a directory name (including the special directory names . and ..), MS-DOS prompts the user for confirmation before deleting all the files in the current or specified directory. Note that in the case of a directory name, the directory itself is not removed; only the files within it are deleted.

Warning: If the filename specification begins with an * wildcard and the extension is also * (for example, *xyz.*), DEL interprets the specification as *.* and prompts the user for confirmation before deleting all files from the current or specified directory.

Examples

To delete the file HELLO.C from the current directory on the current drive, type

```
C>DEL HELLO.C  <Enter>
```

To delete all files with the extension .OBJ from the \SOURCE directory on the disk in drive D, type

```
C>DEL D:\SOURCE\*.OBJ  <Enter>
```

To delete all files from the current directory on the current drive, type

```
C>DEL *.*  <Enter>
```

or

```
C>DEL .  <Enter>
```

In this case, MS-DOS will prompt for confirmation that all files should be deleted.

To delete all files from the directory \WORD\LETTERS on the current drive, type

```
C>DEL \WORD\LETTERS  <Enter>
```

Again, MS-DOS will prompt for confirmation that all files should be deleted.

Messages

Access denied
The specified file is read-only. Use the ATTRIB command with the –R switch to remove the file's read-only status.

Are you sure (Y/N)?
This message prompts the user for confirmation if the command would delete all files in a directory (if the command line ends with a directory name or *.*). Respond with *Y* to delete all files in the directory; respond with *N* to terminate the command.

File not found
The filename in the command is invalid or the file does not exist in the specified directory.

Invalid directory
One of the directories named in the file specification is invalid or does not exist.

Invalid drive specification
The drive code in the file specification is invalid or the named drive does not exist in the system.

DIR

Display Directory

1.0 and later

Internal

Purpose

Displays a list of a directory's files and subdirectories.

Syntax

DIR [*drive*:][*path*][*filename*] [/P] [/W]

where:

filename is the name of the file, optionally preceded by a drive and/or path, whose directory entry is to be displayed; wildcard characters are permitted.

/P causes a pause after each screen page of display.

/W causes a wide display of filenames formatted five across.

Description

The DIR command displays information about the files in a directory. It also displays information about the volume name of the disk that contains the directory, the total number of files and subdirectories in the directory, and the amount of free space remaining on the disk.

The normal format of the DIR command's output is

```
 Volume in drive C is HARDDISK
 Directory of  C:\ASM
.            <DIR>       9-19-85   7:09p
..           <DIR>       9-19-85   7:09p
LIB          <DIR>       9-17-86  11:31p
SOURCE       <DIR>       9-17-86  11:31p
AT86     EXE    41146    5-13-85   5:18p
CREF     EXE    15028   10-16-85   4:00a
DEBUG    COM    15552    3-07-85   1:43p
EXE2BIN  EXE     2816    3-07-85   1:43p
EXEMOD   EXE    11034   10-16-85   4:00a
EXEPACK  EXE    10848   10-16-85   4:00a
LIB      EXE    28716   10-16-85   4:00a
LINK     EXE    43988   10-16-85   4:00a
MAKE     EXE    24300   10-16-85   4:00a
MAPSYM   EXE    18026   10-16-85   4:00a
MASM     EXE    85566   10-16-85   4:00a
SYMDEB   EXE    37021   10-16-85   4:00a
T86      EXE    49024   12-06-84   4:03p
       17 File(s)   4022272 bytes free
```

The first line shows the volume label of the disk that contains the directory being displayed; the second line gives the full pathname of the directory. The subsequent lines are the names of the files and subdirectories within the current or specified directory. Each entry includes the time and date the file or subdirectory was created or last modified.

Files are shown with their exact size in bytes; directories are shown with the symbol <DIR>. If the directory being listed is not the root directory of the disk, it always contains the two special directory entries . and .., which are aliases for the current directory and the parent directory, respectively. These aliases are included in the total file count in the last line of the display.

Subsets of the files and subdirectories in the current or specified directory of the current or specified drive can be listed by including a filename with wildcards in the command line. For example, the filename *.DOC will cause DIR to list only the files with a .DOC extension.

If the command line ends with a drive or path, DIR automatically appends an *.*, causing all files and subdirectories in the current or specified directory of the current or specified drive to be listed. If a filename is included but no extension is given, DIR appends a .* to the filename, causing all files with that name to be listed, regardless of their extension. If a filename ending with a . is included, nothing is appended and all matching subdirectories and filenames without extensions are listed.

The /P switch causes a pause in the display after each screen page (23 lines plus a message). The listing resumes when the user presses a key.

The /W switch causes the list to be in a more compact format by omitting size and date/time information and by displaying the filenames five across:

```
 Volume in drive C is HARDDISK
 Directory of  C:\ASM
.             ..            LIB           SOURCE        AT86     EXE
CREF     EXE  DEBUG    COM  EXE2BIN  EXE  EXEMOD   EXE  EXEPACK  EXE
LIB      EXE  LINK     EXE  MAKE     EXE  MAPSYM   EXE  MASM     EXE
SYMDEB   EXE  T86      EXE
       17 File(s)   4022272 bytes free
```

When the /W form of the listing is displayed, subdirectories are not easily distinguished from files because the <DIR> symbol is not shown.

Examples

To list all files in the current directory on the current drive, type

```
C>DIR  <Enter>
```

To list all files in the current directory on the disk in drive B, type

```
C>DIR B:  <Enter>
```

or

```
C>DIR B:*.*  <Enter>
```

To list all files in the directory \SOURCE on the current drive, type

```
C>DIR \SOURCE  <Enter>
```

or

```
C>DIR \SOURCE\*.*  <Enter>
```

To list all files with the extension .OBJ in the \LIB directory on the disk in drive D, type

```
C>DIR D:\LIB\*.OBJ  <Enter>
```

To list all files in the parent directory of the current directory on the current drive, type

```
C>DIR ..  <Enter>
```

To list all files in the current directory on the current drive, sorted by filename and extension, type

```
C>DIR ¦ SORT  <Enter>
```

To list all files in the current directory on the current drive, sorted by extension, type

```
c>DIR ¦ SORT /+10  <Enter>
```

The */+10* instructs SORT to sort the directory entries starting at the tenth column, which is the first column of the filename extension.

To list the subdirectories and files without extensions in the current directory, type

```
C>DIR *.  <Enter>
```

To print the directory on an attached printer instead of displaying it on the screen, type

```
C>DIR > PRN  <Enter>
```

To make a copy of the directory in a file called FILES.TXT, type

```
C>DIR > FILES.TXT  <Enter>
```

Messages

File not found
A filename was included in the command line and no matching files were found.

Invalid directory
An element of the path included in the command line does not exist.

Invalid drive specification
The specified drive is invalid or is not present in the system.

Strike a key when ready...
If the DIR command includes the /P switch, the display is suspended after each 23 lines and this message prompts the user to press a key to see the next screenful of entries.

DISKCOMP

3.2

Compare Floppy Disks

External

Purpose

Compares two entire floppy disks on a sector-by-sector basis and reports any differences. This command was included with PC-DOS beginning with version 1.0. To compare individual files, *see* USER COMMANDS: COMP; FC.

Syntax

DISKCOMP [*drive1*:] [*drive2*:] [/1] [/8]

where:

drive1	is the drive containing the first disk to be compared.
drive2	is the drive containing the second disk to be compared.
/1	compares only the first sides of the disks.
/8	compares only the first eight sectors of each track.

Description

The DISKCOMP command compares the physical sectors of one floppy disk with those of another. The *drive1* and *drive2* parameters designate the drives holding the two disks to be compared; the drives should always be of the same type. If *drive2* is omitted, DISKCOMP uses the current drive. If both *drive1* and *drive2* are omitted or are identical, DISKCOMP performs the comparison using a single drive, prompting the user to swap disks as required.

Ordinarily, DISKCOMP determines the disk format by inspecting the disk in *drive1*. The /1 and /8 switches override this check so that only one side of the disks or only the first eight sectors of each track are compared, regardless of the actual format of the disks.

If all the sectors on all the tracks are identical, DISKCOMP displays the message *Compare OK.* If differences are found, DISKCOMP reports them by issuing a message that includes the numbers of the track and disk side (read/write head) where the differences occur. Because DISKCOMP works at the level of the disks' physical sectors and is ignorant of the control areas and file structures imposed on a disk by MS-DOS, it also reports as errors bad sectors that were marked during the FORMAT process.

When DISKCOMP finishes comparing two disks, it displays a prompt that allows the user to choose between comparing another pair of disks and returning to the MS-DOS command level.

DISKCOMP cannot be used with a network drive or with a drive created or affected by an ASSIGN, JOIN, or SUBST command, nor can it be used with fixed disks.

Return Codes

0 Compared disks were identical.
1 Differences were found between the compared disks.
2 DISKCOMP was terminated with a Control-C.
3 Bad sector was found on one of the disks being compared.
4 Initialization error was encountered: not enough memory, syntax error in command line, or invalid drive specified in command line.

Note: Return codes are not present in the PC-DOS version of DISKCOMP.

Examples

To compare the disk in drive A with the disk in drive B, type

```
C>DISKCOMP A: B:  <Enter>
```

To compare two disks using only drive A, type

```
C>DISKCOMP A: A:  <Enter>
```

To compare only the first side of the disk in drive A with the first side of the disk in drive B, type

```
C>DISKCOMP A: B: /1  <Enter>
```

To compare only the first eight sectors of each track on one side of one disk with the first eight sectors of each track on one side of another disk using only drive A, type

```
C>DISKCOMP A: A: /1 /8  <Enter>
```

Messages

Cannot DISKCOMP to or from
an ASSIGNed or SUBSTed drive
One of the specified drives has been affected by an ASSIGN or SUBST command.

Cannot DISKCOMP to or from
a network drive
One of the specified drives is a network device.

Compare another diskette (Y/N) ?
This prompt allows comparison of another pair of disks. Respond with *Y* to cause DISKCOMP to prompt for insertion of the next pair of disks to be compared; respond with *N* to exit to MS-DOS.

Compare error on side *n*, track *n*
A difference was detected between the two disks being compared.

Compare OK
The two disks being compared are identical.

Compare process ended
The disk comparison was terminated as the result of a fatal error.

Comparing *n* tracks,
***n* sectors per track, *n* side(s)**
This informational message specifies the format of the two disks being compared.

DEVICE Support Not Present
The disk drive does not support MS-DOS 3.2 device control.

Drive *X* not ready
Make sure a diskette is inserted into
the drive and the door is closed
DISKCOMP was unable to read the disk in the specified drive.

Drive types or diskette types
not compatible
Single-sided disks cannot be compared with double-sided disks, nor high-density disks with double-density disks.

FIRST diskette bad or incompatible
DISKCOMP is unable to determine the format of the first disk.

Incorrect DOS version
The version of DISKCOMP is not compatible with the version of MS-DOS that is running.

Insert diskette with directory that contains
COMMAND.COM in drive *X* and strike any key when ready
If the system was booted from a floppy disk and the system disk was then removed in order to use DISKCOMP, the user must replace the system disk after the compare operation is complete.

Insert FIRST diskette in drive *X*:
Press any key when ready...
This message prompts the user to insert the first disk of a pair to be compared.

Insert SECOND diskette in drive *X*:
Press any key when ready...
This message prompts the user to insert the second disk of a pair to be compared.

Insufficient memory
The available system memory is insufficient to load and execute the DISKCOMP program.

Invalid drive specification
Specified drive does not exist
or is non-removable
One of the drives specified in the command line is invalid or does not exist.

Invalid parameter
Do not specify filename(s)
Command format: DISKCOMP d: d: [/1][/8]
A syntax error was detected in the command line, usually caused by an incorrect switch.

SECOND diskette bad or incompatible
The second disk of a pair to be compared does not have the same format as the first disk or has bad sectors preventing DISKCOMP from determining its format.

Unrecoverable read error on drive *X*:
The disk in the specified drive contains an unreadable sector.

DISKCOPY

2.0 and later

Copy Floppy Disks

External

Purpose

Performs a sector-by-sector copy of one entire floppy disk to another floppy disk. This command was included with PC-DOS beginning with version 1.0. To copy individual files, *see* USER COMMANDS: COPY.

Syntax

DISKCOPY [*drive1*:] [*drive2*:] [/1]

where:

drive1	is the drive containing the disk to be copied.
drive2	is the drive containing the disk that will become the copy.
/1	copies only the first side of the disk in *drive1* (MS-DOS version 3.2).

Description

The DISKCOPY command duplicates a floppy disk, performing the copy on a physical sector-by-sector basis. The *drive1* parameter specifies the location of the disk to be copied (the source disk). The *drive2* parameter specifies the location of the disk that will become the copy (the destination disk). If *drive2* is omitted, the current drive is used as the destination drive; if both *drive1* and *drive2* parameters are omitted or are the same, DISKCOPY performs the copy operation using a single drive, prompting the user to swap the disks as necessary.

DISKCOPY examines the destination disk before writing any information and terminates with an error message if it does not have the same format as the source disk. If the destination disk is not formatted, DISKCOPY formats it with the same format as the source disk, as part of the DISKCOPY operation.

Note: With MS-DOS versions 2.0 through 3.1, the destination disk must be formatted using the FORMAT command before DISKCOPY can be used. All PC-DOS versions of DISKCOPY will automatically format the destination disk, if necessary.

When DISKCOPY finishes copying a disk, it displays a prompt that allows the user to choose between copying another disk and returning to the MS-DOS command level.

Because DISKCOPY creates an exact duplicate of the source disk, any file fragmentation present on the source disk is also present on the destination disk after the DISKCOPY process is complete. To eliminate fragmentation of the source files, they should be copied to the destination disk individually using COPY or XCOPY.

The DISKCOPY command cannot be used with a network drive or with a drive created or affected by an ASSIGN, JOIN, or SUBST command, nor can it be used with fixed disks.

Return Codes

0 Disk was copied successfully.
1 Nonfatal but unrecoverable read or write error occurred (no Interrupt 24H generated).
2 DISKCOPY was terminated with a Control-C.
3 Fatal error was encountered: unreadable source disk or unformattable destination disk.
4 Initialization error was encountered: not enough memory, syntax error in command line, or invalid drive specified in command line.

Note: Return codes are not present in the PC-DOS version of DISKCOPY.

Examples

To copy the contents of the disk in drive A to the disk in drive B, type

```
C>DISKCOPY A: B:  <Enter>
```

To copy the contents of the disk in drive A using only one drive, type

```
C>DISKCOPY A: A:  <Enter>
```

To copy only the first side of the disk in drive A to the first side of the disk in drive B, type

```
C>DISKCOPY A: B: /1  <Enter>
```

Messages

Cannot DISKCOPY to or from
an ASSIGNed or SUBSTed drive
One of the specified drives has been affected by an ASSIGN or SUBST command.

Cannot DISKCOPY to or from
a network drive
One of the specified drives is a network device.

Copy another diskette (Y/N) ?
This prompt allows copying of another disk. Respond with *Y* to cause DISKCOPY to prompt for insertion of the next set of disks; respond with *N* to exit to MS-DOS.

Copying *n* tracks
***n* sectors per track, *n* side(s)**
This informational message specifies the format of the source disk being copied.

Copy process ended
The DISKCOPY process has been successfully completed or has been terminated by a fatal error. In the latter case, this message is preceded by another message explaining the error.

DEVICE Support Not Present
The disk drive does not support MS-DOS version 3.2 device control.

Disk error while reading drive *X*:
Abort, Retry, Ignore?
A bad sector was detected on the source disk. This does not necessarily invalidate the disk copy; the bad sector may originally have been detected and flagged by the FORMAT program and therefore not included in any file. One solution is to copy the files individually using the COPY command.

Drive *X*: not ready
Make sure a diskette is inserted into
the drive and the door is closed
DISKCOPY was unable to read the disk in the specified drive.

Drive types or diskette types
not compatible
Single-sided disks cannot be copied to or from double-sided disks, nor high-density disks to or from double-density disks.

Formatting while copying
The destination disk was not previously formatted. It is given the same format as the source disk as part of the DISKCOPY operation (MS-DOS version 3.2).

Incorrect DOS version
The version of DISKCOPY is not compatible with the version of MS-DOS that is running.

Insert diskette with directory that contains
COMMAND.COM in drive *X* and strike any key when ready
If the system was booted from a floppy disk and the system disk was then removed in order to use DISKCOPY, the user must replace the system disk after the copy operation is complete.

Insert SOURCE diskette in drive *X*:
Press any key when ready...

or

Insert TARGET diskette in drive *X*:
Press any key when ready...
These messages prompt the user to insert the source and destination disks before beginning the copy operation.

Insufficient memory
The available system memory is insufficient to load and execute the DISKCOPY program.

Invalid drive specification
Specified drive does not exist,
or is non-removable
One of the drives specified in the command line is invalid or does not exist. A fixed disk cannot be the source or destination disk for a DISKCOPY operation.

Invalid parameter
Do not specify filename(s)
Command Format: DISKCOPY d: d: [/1]
A syntax error was detected in the command line, usually caused by an incorrect switch or by the use of a filename instead of (or in addition to) a disk drive.

SOURCE diskette bad or incompatible

or

TARGET diskette bad or incompatible
The source disk could not be read or the destination disk could not be formatted.

Target diskette is write protected
The destination disk has a write-protect tab on it.

Target diskette may be unusable
Unrecoverable read or write errors were encountered while copying the source disk to the destination disk. The newly copied disk may not be an accurate copy.

Unrecoverable read error on drive *X*:
side *n*, track *n*

or

Unrecoverable write error on drive *X*:
side *n*, track *n*
The disk in the specified drive contained a sector that could not be successfully read or written.

DRIVER.SYS

3.2

Configurable External-Disk-Drive Driver

External

Purpose

Installs and configures external disk drives or assigns logical drive letters to existing floppy-disk drives.

Syntax

DEVICE=DRIVER.SYS /D:*n* [/C] [/F:*n*] [/H:*n*] [/N] [/S:*n*] [/T:*n*]

where:

/D:*n*	is the drive number (0–127 for floppy disks, 128–255 for fixed disks) and must always be the first switch in the command line.
/C	specifies that door-lock-status support is available.
/F:*n*	is the form-factor index for the device (default = 2):

0	320/360 KB
1	1.2 MB
2	720 KB
3	8" single-density floppy disk
4	8" double-density floppy disk
5	fixed disk
6	magnetic-tape drive
7	other

/H:*n*	is the number of heads supported by the disk drive (1–99).
/N	specifies a nonremovable block device.
/S:*n*	is the number of sectors per track (1–40).
/T:*n*	is the tracks per read/write head (1–999).

Description

When the computer is turned on or restarted, MS-DOS assigns numbers to all existing internal disk drives. The DRIVER.SYS file—an installable, configurable block-device driver for external disk drives and other mass-storage devices—allows installation of peripheral devices that are not supported by the resident drivers in the MS-DOS BIOS module. DRIVER.SYS can also assign a logical drive letter to an existing disk drive, thus giving the device two drive letters. (This allows such activities as copying files between like media—for example, copying files from one 1.2 MB 5.25-inch disk to another—using the same drive.)

The /D:*n* switch assigns a unit number to the additional disk drive or specifies the number of the existing disk drive that is to be assigned a logical drive letter. (Floppy-disk unit numbers begin at 0; fixed-disk numbers begin at 80H.) For example, if the system contains two floppy-disk drives (0 and 1), an external floppy-disk drive requiring DRIVER.SYS would be assigned the value 2; MS-DOS would then assign that drive the next available drive letter. If the number used with the /D:*n* switch references an existing drive (for example, 0, the first floppy-disk drive), MS-DOS assigns the drive the next available drive letter, allowing the one drive unit to be referenced by two drive letters. The /D:*n* switch is not optional and must precede all other switches in the command line.

The /C, /F:*n*, and /N switches describe characteristics of the disk drive that is being selected for use with DRIVER.SYS. The /C switch is included only if the device has a status line indicating whether the disk in the drive has been changed. (This information is used by the driver to optimize disk accesses to the directory and file allocation table.) If the device does not have a status line, /C will have no effect. The /F:*n* option describes the form-factor index used by the device. The permissible values for *n* are given in the preceding table; the default type is a 720 KB disk. The /N switch indicates that the block device is nonremovable. Access to such devices is more efficient than access to removable media because MS-DOS can eliminate calls to the driver for a media-change check.

The /H:*n*, /S:*n*, and /T:*n* switches describe the physical layout of the recording medium. /H:*n* specifies the number of recording surfaces, or read-write heads, supported by the drive (1–99). /S:*n* is the number of sectors per track (1–40) and /T:*n* is the tracks per side (1–999). (The total number of physical sectors on a given disk is found by multiplying the number of heads by the tracks per side and the sectors per track.)

Note: The values used with these switches must be supported by the device being installed. If DRIVER.SYS is used to assign a logical drive letter to an existing physical device, the values used with the switches must be identical to the characteristics imposed by the default device driver.

Examples

To install a driver for an external 720 KB disk drive in a system that already has two 5.25-inch floppy-disk drives, insert the line

```
DEVICE=DRIVER.SYS /D:02
```

into the CONFIG.SYS file and restart the system.

Assume that an IBM PC/AT or compatible has three disk drives installed: Drive A is a 1.2 MB 5.25-inch floppy-disk drive; drive B is a 360 KB 5.25-inch floppy-disk drive; drive C is a 30 MB fixed-disk drive. To assign the logical drive letter D to the existing drive A, effectively giving the one drive two drive letters, insert the line

```
DEVICE=DRIVER.SYS /D:0 /F:1 /H:2 /S:15 /T:80 /C
```

into the CONFIG.SYS file and restart the system.

Messages

Bad or missing DRIVER.SYS
The file DRIVER.SYS could not be found in the root or specified directory or has been damaged.

ERROR - Incorrect DOS version
The version of DRIVER.SYS is not compatible with the version of MS-DOS that is running.

ERROR - No drive specified
The /D:*n* switch was not included in the command line.

Loaded External Disk Driver for Drive *X*
The device driver has been successfully installed and this message informs the user of the drive letter assigned to the device.

Sector size too large in file DRIVER.SYS
DRIVER.SYS uses a sector size that is larger than the sector size used by any of the system's default disk drivers. The driver cannot be used because MS-DOS's internal disk buffers will not be large enough to hold a sector read from the device.

EDLIN

1.0 and later

Line Editor

External

Purpose

Creates and changes ASCII text files.

Syntax

EDLIN [*drive*:][*path*]*filename* [/B]

where:

filename is the name of an ASCII text file to be created or edited, optionally preceded by a drive and/or path.

/B causes logical end-of-file marks within the file to be ignored (versions 2.0 and later).

Description

The EDLIN program is a simple line-oriented editor that can be used to create or maintain short text files. The user references and edits text by line number; EDLIN displays these numbers for convenience but they do not become part of the file. Each line of the file being edited can be a maximum of 253 characters.

The *filename* parameter specifies a plain ASCII text file; if the file does not already exist, EDLIN creates it. (EDLIN cannot be used on most files created by word-processing programs because such document files have embedded formatting codes and other formatting information that EDLIN cannot interpret.) EDLIN does not assume any extensions; the user must type the complete filename. (EDLIN does not permit editing of a .BAK file.)

If *filename* is a previously existing text file, EDLIN loads lines from the file into memory until the editing buffer is 75 percent full or until a logical end-of-file mark or the physical end of the file is reached. The /B switch forces EDLIN to ignore any logical end-of-file marks (1AH, or Control-Z) the file may contain. If the file is too large for the edit buffer, the Write Lines to Disk (W) and Append Lines from Disk (A) commands are used during the edit session to process the remaining portions of the file.

Once the file is created or loaded into the editing buffer, EDLIN displays its asterisk prompt (*) and the user can begin entering editing commands.

EDLIN commands consist of a single character, in either uppercase or lowercase, usually preceded by one or more line numbers. More than one command can be entered on a single line by separating the commands with semicolons. EDLIN does not execute a command until the Enter key is pressed.

The EDLIN commands are

Command	Action
linenumber	Edit line.
A	Append lines from disk.
C	Copy lines (versions 2.0 and later).
D	Delete lines.
E	End editing session.
I	Insert lines.
L	List lines.
M	Move lines (versions 2.0 and later).
P	Display in pages (versions 2.0 and later).
Q	Quit without saving changes.
R	Replace text.
S	Search for text.
T	Transfer another file into the edit buffer (versions 2.0 and later).
W	Write lines to disk.

Each of these commands is discussed in detail in the following pages.

All EDLIN commands that accept a line number or range of line numbers can also recognize the following symbolic references:

Symbol	Meaning
#	The line after the last line in the edit buffer
.	The current line
+*n* or −*n*	A line number relative to the current line (for example, +5 = five lines past the current line)

When the user terminates the editing session with the E command, EDLIN gives the new file the same name as the original file and renames the original (unchanged) file with the extension .BAK. Any previous file with the same name and the extension .BAK is lost. When the user terminates the editing session with the Q command, the original filename remains unchanged.

Example

To edit the file AUTOEXEC.BAT in the root directory of the current drive, type

```
C>EDLIN \AUTOEXEC.BAT  <Enter>
```

Messages

Cannot edit .BAK file — rename file
Files with the extension .BAK cannot be edited with EDLIN. Rename the file or copy it to a file with the same name but a different extension.

End of input file
The entire file has been read into memory.

File is READ-ONLY
Files marked with the read-only attribute cannot be edited. Remove the read-only attribute with the ATTRIB command or copy the file to a file with a different name.

File name must be specified
The command line did not include a filename.

File not found
The file named in the command line could not be found or does not exist.

Incorrect DOS version
The version of EDLIN is not compatible with the version of MS-DOS that is running.

Insufficient memory
Not enough memory is available to carry out the requested command.

Invalid drive or file name
The command line included a drive that is invalid or does not exist in the system or the filename is not valid.

Invalid Parameter
The command line contained an illegal switch or other invalid parameter.

New file
The file named in the command line did not previously exist. The file is created and the edit buffer is emptied.

Read error in: *filename*
MS-DOS was unable to read the entire file. Run CHKDSK to determine whether the file or disk has been damaged.

EDLIN: *linenumber*

1.0 and later

Edit Line

Purpose

Selects a line of text for editing.

Syntax

linenumber

where:

linenumber is the number assigned by EDLIN to the text line to be edited (1–65534).

Description

The command to edit a particular line of text is simply the line's number or one of the special symbols or expressions that evaluate to a line number, followed by the Enter key. EDLIN displays the current contents of the specified line and copies them to a special editing buffer called the template, then moves the cursor to a new line and displays a prompt in the form of the line number followed by a colon and an asterisk. If a line number is not specified (that is, if the Enter key alone is pressed in response to the EDLIN prompt), EDLIN displays the line following the current line and makes it the current line.

The user can change the text of the specified line by simply entering new text followed by a press of the Enter key, leave the text unchanged by pressing Enter alone, or modify the text by using special editing keys to change a portion of the text that has been placed in the template. These editing keys and their actions are

Key	Action
F1	Copies one character from the template to the new line.
F2*char*	Copies all characters up to the specified character from the template to the new line.
F3	Copies all remaining characters in the template to the new line.
Del	Does not copy (skips over) one character.
F4*char*	Does not copy (skips over) all characters up to the specified character.
Esc	Restarts editing for the current line, leaving the template unchanged.
Ins	Enters/exits character-insert mode.
F5	Makes the newly edited line the new template.
→	Copies one character from the template to the new line.
←	Deletes one character from the new line.
Backspace	Deletes one character from the new line.

Note: Computers that are not IBM-compatible may use a different set of editing keys to perform these actions.

Control characters (those characters with ASCII codes in the range 0–1FH) cannot be inserted into text with the usual Control-key combinations. Instead, the user must press the sequence Ctrl-V, followed by an uppercase character or symbol. For example, Ctrl-C (ASCII code 03H) is entered into text by pressing Ctrl-V followed by a capital C; the Escape character (ASCII code 1BH) is generated by pressing Ctrl-V followed by a left square-bracket character ([).

Examples

To edit line 4, type

```
*4  <Enter>
```

To edit the line two lines ahead of the current line, type

```
*+2  <Enter>
```

EDLIN: A

1.0 and later

Append Lines from Disk

Purpose

Reads lines from the file being edited into the edit buffer.

Syntax

[*n*]A

where:

n is the number of lines to be read from the file.

Description

If the file being edited is too large to fit into the edit buffer, EDLIN ordinarily reads only enough text to fill 75 percent of the buffer when it opens the file, reserving 25 percent of the buffer for additions and changes to the text. The user must then employ the Write Lines to Disk (W) and Append Lines from Disk (A) commands to write and read successive blocks of text until the entire file has passed through the edit buffer.

The A command alone has no effect if the edit buffer is 75 percent or more full. The W command must be used to write lines to the output file and delete them from the buffer; then the A command can read new lines from the input file and append them to the end of the text remaining in the buffer.

The *n* parameter specifies the number of lines to be read from the file. If *n* is omitted or is too large, EDLIN reads only enough lines to fill the editing buffer to 75 percent of its capacity.

Examples

To append 200 lines from the disk file to the edit buffer, type

```
*200A  <Enter>
```

To append as many lines from the file as possible (until the edit buffer is 75 percent full), type

```
*A  <Enter>
```

Message

End of input file
The last section of the file being edited has been read into the edit buffer.

EDLIN: C

2.0 and later

Copy Lines

Purpose

Copies one or more lines from one location in the edit buffer to another.

Syntax

[*first*],[*last*],*destination*[,*count*]C

where:

first is the number of the first line to be copied.
last is the number of the last line to be copied.
destination is the number of the line before which the copied lines are to appear.
count is the number of times to execute the copy operation.

Description

The Copy Lines (C) command copies one or more text lines, inserting the copied lines at another location in the edit buffer. The original lines that were copied are unchanged. EDLIN then renumbers the edit buffer and makes the first copied line at the destination the new current line.

The *first* and *last* line-number parameters define the block of lines to be copied. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but the commas must still be entered as placeholders. The *destination* parameter specifies the line before which the copied lines are to be inserted; it is not optional and must not fall within the range of line numbers specified by *first* and *last*. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

To replicate the line or lines multiple times, the copy operation can be repeated automatically with the optional parameter *count*. The default value for *count* is one.

Examples

If the current line is line 10, to copy lines 10 through 15 and place the copied lines before line 5, type

```
*10,15,5C  <Enter>
```

or

```
*,15,5C  <Enter>
```

or

```
*,+5,-5C  <Enter>
```

If the current line is line 10, to place three copies of lines 10 through 15 before line 1, type

```
*10,15,1,3C  <Enter>
```

or

```
*,15,1,3C  <Enter>
```

or

```
*,+5,1,3C  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number or a destination line number that fell within the range *first,last.*

Insufficient memory
The edit buffer does not have sufficient room for EDLIN to carry out the specified command.

Must specify destination line number
No destination line number was specified in the command line; therefore, no changes were made to the edit buffer.

EDLIN: D

1.0 and later

Delete Lines

Purpose

Deletes one or more lines from the edit buffer.

Syntax

[*first*][,*last*]D

where:

first is the number of the first line to delete.
last is the number of the last line to delete.

Description

The Delete Lines (D) command removes one or more text lines from the edit buffer. The line after the last line deleted becomes the new current line.

The *first* and *last* line-number parameters define the block of lines to be deleted. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but a leading comma is required as a placeholder if *first* is omitted when *last* is present. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

Examples

If the current line is line 10, to delete the current line, type

```
*10D  <Enter>
```

or

```
*D  <Enter>
```

If the current line is line 10, to delete lines 10 through 15, type

```
*10,15D  <Enter>
```

or

```
*,15D  <Enter>
```

or

```
*,+5D  <Enter>
```

If the current line is line 10, to delete all lines from the current line to the end of the buffer, type

```
*10,#D  <Enter>
```

or

```
*,#D  <Enter>
```

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: E

1.0 and later

End Editing Session

Purpose

Saves the edited file to disk and exits from EDLIN.

Syntax

E

Description

The End Editing Session (E) command writes the contents of the edit buffer to the current directory of the disk in the current drive. If a previously existing file was being edited and there is any text remaining in the original file that has not yet passed through the edit buffer, EDLIN copies this text to the output file. EDLIN gives the newly edited file the same name as the original file and renames the original (unchanged) file with the extension .BAK. Any previous file with the same name and the extension .BAK is lost. EDLIN then returns to MS-DOS.

If the disk does not have enough space to hold the edited file in addition to the original file, EDLIN writes as much of the edited file as possible into a file with the extension .$$$; the remainder of the edited text is lost. The name and contents of the original file are left unchanged.

Example

To end an editing session, type

```
*E  <Enter>
```

Messages

Disk full. Edits lost.
The disk does not contain enough free space for the edited file. A partial file may have been created with the extension .$$$.

File Creation Error
The .BAK file is marked read-only, the root directory is full or cannot contain any more files, or the filename is the same as a volume label or directory name.

No room in directory for file
The file could not be saved because its destination was the root directory and the root directory is full.

Too many files open
MS-DOS was unable to open the .BAK file due to a lack of available system file handles. Increase the value of the FILES command in the CONFIG.SYS file.

EDLIN: I

1.0 and later

Insert Lines

Purpose

Inserts new lines into the edit buffer.

Syntax

[*destination*]I

where:

destination is the number of the line before which text is to be inserted.

Description

The Insert Lines (I) command enables insert mode and allows new text to be placed between previously existing lines of text. When insert mode is terminated, the first line following the inserted lines becomes the new current line.

EDLIN places the new text before the line specified by the *destination* parameter. If *destination* is omitted, EDLIN assumes the current line; if *destination* is larger than the number of lines in the edit buffer, EDLIN simply appends the new text after the actual last line. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of an absolute line number.

After an I command, EDLIN issues a prompt consisting of the line number for the inserted text followed by a colon and an asterisk and continues to issue such prompts each time the Enter key is pressed until the user terminates insert mode by pressing Ctrl-C or Ctrl-Break.

Examples

If the current line is line 10, to insert text before line 7, type

```
*7I  <Enter>
```

or

```
*-3I  <Enter>
```

To insert lines at the beginning of the buffer, type

```
*1I  <Enter>
```

To insert lines at the end of the buffer, type

```
*#I  <Enter>
```

Message

Insufficient memory
The edit buffer does not have sufficient room for EDLIN to complete the specified command.

EDLIN: L

1.0 and later

List Lines

Purpose

Displays one or more lines from the edit buffer.

Syntax

[*first*][,*last*]L

where:

first	is the number of the first line to be displayed.
last	is the number of the last line to be displayed.

Description

The List Lines (L) command displays text lines on standard output. If the current line lies within the range of lines listed, EDLIN displays an asterisk next to its number. The current line is not changed.

The *first* and *last* line-number parameters define the block of lines to be listed. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. One of the special symbols **.** (current line) and # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

If only the first line number is specified, EDLIN displays text in 23-line increments starting with that number. If only the last line number is specified, EDLIN displays text beginning 11 lines before the current line and continuing to the specified last line. If no line numbers are specified in the command, EDLIN lists the 23 lines centered around the current line; if the current line number is less than 13, EDLIN lists the first 23 lines in the buffer.

Examples

To display lines 20 through 30, type

```
*20,30L  <Enter>
```

If the current line is 20, to display the 23 lines centered around the current line, type

```
*L  <Enter>
```

EDLIN displays lines 9 through 31.

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: M

2.0 and later

Move Lines

Purpose

Moves lines from one place in the edit buffer to another.

Syntax

[*first*],[*last*],*destination*M

where:

first is the number of the first line to be moved.
last is the number of the last line to be moved.
destination is the number of the line before which the moved lines are to be inserted.

Description

The Move Lines (M) command transfers one or more text lines from one location in the edit buffer to another. EDLIN then deletes the original lines and renumbers the edit buffer. The first moved line becomes the new current line.

The *first* and *last* line-number parameters define the block of lines to be moved. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted (in which case the current line number is used), but the commas must still be entered as placeholders. The *destination* parameter specifies the line before which the moved lines are to be inserted; it is not optional and must not fall within the range of line numbers specified by *first* and *last*. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of absolute line numbers.

Example

If the current line is line 10, to move lines 10 through 15 and place them before line 5, type

```
*10,15,5M  <Enter>
```

or

```
*,15,5M  <Enter>
```

or

```
*,+5,-5M  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number or a destination line number that fell within the range *first,last.*

Must specify destination line number
No destination line number was specified in the command line; therefore, no changes were made to the edit buffer.

EDLIN: P

2.0 and later

Display in Pages

Purpose

Displays lines for viewing in successive screenfuls (pages).

Syntax

[*first*][,*last*]P

where:

first	is the number of the first line to be displayed.
last	is the number of the last line to be displayed.

Description

The Display in Pages (P) command displays text lines on standard output one screenful at a time. Unlike the List Lines (L) command, which has no effect on the current line, P causes the last line displayed to become the new current line. Thus, although the edit buffer is not actually organized into pages, the user can employ repeated P commands to sequentially view successive groups of lines.

The *first* and *last* line-number parameters define the block of lines to be listed; the display starts with the line specified by *first.* (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the line 23 lines after the current line. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

Examples

If the current line is 20, to view the next page of lines in the edit buffer, type

```
*P  <Enter>
```

EDLIN displays 23 lines, beginning with line 21, and changes the current line to line 43.

To view successive pages of 23 lines, repeatedly type

```
*P  <Enter>
```

Message

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

EDLIN: Q

1.0 and later

Quit

Purpose

Terminates the editing session without saving the revised file.

Syntax

Q

Description

The Quit (Q) command causes EDLIN to exit without saving any of the changes made to the edited file during the session. The original file's name and contents are left unchanged and no new file is created.

To reduce the danger of accidentally losing the contents of the edit buffer, EDLIN prompts the user for confirmation before carrying out the Q command.

Example

To quit an editing session, type

```
*Q  <Enter>
```

EDLIN issues a prompt for confirmation and, if the response from the user is *Y*, exits to MS-DOS without saving any changes made to the file during the session.

Message

Abort edit (Y/N)?
This prompt is displayed in response to the Q command. Respond with *Y* to exit to MS-DOS without saving changes made to the file; respond with *N* to continue the editing session.

EDLIN: R

1.0 and later

Replace Text

Purpose

Replaces one string in the edit buffer with another.

Syntax

[*first*][,*last*][?]R[*string1*][^Z*string2*]

where:

first	is the number of the first line to be searched.
last	is the number of the last line to be searched.
?	causes the user to be prompted for confirmation before each replacement is made.
string1	is the sequence of characters to be searched for.
^Z	is a Control-Z character.
string2	is the sequence of characters to be substituted for *string1*.

Note: The character limit for the Replace Text command is 127 characters, including both strings and all other parameters.

Description

The Replace Text (R) command substitutes one character string for another within a specified range of lines. The last line in which a replacement occurs becomes the new current line.

The *first* and *last* line-number parameters define the range of lines to be searched for strings to replace. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the last line in the buffer. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

If *string1* is omitted, EDLIN uses the *string1* from the preceding R command; if there was no preceding R command, EDLIN displays an error message. If *string2* is omitted, EDLIN deletes all occurrences of *string1*. *string1* must be separated from *string2* by a Control-Z (^Z) character. If *string1* is omitted, a Control-Z character must still be included to mark the beginning of *string2*, but if *string2* is omitted when *string1* is present, the Control-Z character has no effect and is therefore optional. (The Control-Z character is entered by pressing Ctrl-Z or the F6 key.)

If the ? option is not included in the command line, EDLIN displays each line that contains a match *after* the replacement is carried out. If the ? option is used, EDLIN displays each line containing a match as it is found and prompts the user for confirmation *before* the string is replaced.

The matching operation is case sensitive; EDLIN carries out the substitution only on sequences of characters that match *string1* exactly. Wildcards are not permitted.

Examples

If the current line is line 10, to replace all occurrences of the string *logical* with the string *bitwise* in lines 11 through 20, type

```
*11,20Rlogical^Zbitwise  <Enter>
```

or

```
*,20Rlogical^Zbitwise  <Enter>
```

To cause EDLIN to prompt for confirmation before replacing each string, type

```
*11,20?Rlogical^Zbitwise  <Enter>
```

or

```
*,20?Rlogical^Zbitwise  <Enter>
```

To delete all occurrences of the string *00H* in line 20, type

```
*20,20R00H^Z  <Enter>
```

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

Insufficient memory
The edit buffer has insufficient room for EDLIN to carry out the specified Replace Text command.

Line too long
The replacement would cause the line being edited to expand beyond 253 characters.

Not found
No occurrence or further occurrences of the string to be replaced were found in the specified range of lines.

O.K.?
If the ? option is used in the command line, this prompt is displayed each time a matching string is found. Respond with *Y* or press the Enter key to replace the string and continue searching; press any other key to leave the string unchanged and continue searching.

EDLIN: S

1.0 and later

Search for Text

Purpose

Searches the edit buffer for a character string.

Syntax

[*first*][,*last*][?]S[*string*]

where:

first	is the number of the first line to be searched.
last	is the number of the last line to be searched.
?	causes the user to be prompted for confirmation before the search is terminated.
string	is the sequence of characters to be searched for (maximum 126 characters).

Description

The Search for Text (S) command searches for a character string within a specified range of lines. When a match is found, EDLIN displays the line containing the match and that line becomes the new current line. If no lines containing the specified string are found, EDLIN displays the message *Not found* and the current line number remains unchanged.

The *first* and *last* line-number parameters define the block of lines to be searched for strings. (Note that the first line number must be less than or equal to the last line number.) Either or both of these numbers can be omitted, but a leading comma is required as a placeholder if *first* is omitted when *last* is present. If omitted, *first* defaults to the line after the current line and *last* defaults to the last line in the buffer. One of the special symbols **.** (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or –*n*) can be used instead of absolute line numbers.

If *string* is omitted, EDLIN uses the *string* from the last S command or *string1* from the last Replace Text (R) command instead.

If the ? option is not included in the command line, EDLIN displays the first line that contains a match for *string,* makes this the new current line, and terminates the search. If the ? option is used, EDLIN displays each line containing a match for *string* as it is found, followed by an *O.K.?* prompt. If the user responds with *Y* or presses the Enter key, EDLIN terminates the search; if the user presses any other key, the search continues.

The matching operation is case sensitive; EDLIN reports only sequences of characters that match *string* exactly. Wildcards are not permitted.

Examples

If the current line is line 10, to find the first occurrence of the string *xyz* in lines 11 through 20, type

```
*11,20Sxyz  <Enter>
```

or

```
*,20Sxyz  <Enter>
```

To find a particular occurrence of *proc* in the edit buffer, type

```
*1,#?Sproc  <Enter>
```

EDLIN displays the first line containing *proc* and prompts with

```
O.K.?
```

Type *Y* or press Enter to stop the search; press any other key to continue the search.

Messages

Entry error
The command line contained an error such as a first line number that was greater than the last line number.

Not found
No match or no further matches for *string* were found in the specified range of lines.

O.K.?
If the ? option is used in the command line, this prompt is displayed each time a matching string is found. Respond with *Y* or press the Enter key to stop searching; press any other key to continue searching.

EDLIN: T

2.0 and later

Transfer Another File

Purpose

Merges the contents of another file with the file in the edit buffer.

Syntax

[*destination*]T[*drive*:][*path*]*filename*

where:

destination	is the number of the line before which the text from *filename* is to be inserted.
path	is the location of the file to be merged (versions 3.0 and later).
filename	is the name of the disk file from which text is to be merged.

Description

The Transfer Another File (T) command merges the contents of a text file with the current contents of the edit buffer and then renumbers the contents of the edit buffer. The first line of the merged text becomes the current line.

The *destination* parameter specifies the line before which the transferred lines are to be inserted. If omitted, *destination* defaults to the current line. One of the special symbols . (current line) or # (end of buffer) or an expression relative to the current line number (+*n* or −*n*) can be used instead of an absolute line number.

The *filename* parameter specifies the file from which text is to be merged and can include a drive and, in versions 3.0 and later, a path. If a drive or path is not specified, the file to be merged into the edit buffer with the T command *must* be in the current directory of the current drive.

Example

If the current line is line 10, to merge the contents of the file named KEYDEFS.C before line 10 of the edit buffer, type

```
*10Tkeydefs.c  <Enter>
```

or

```
*Tkeydefs.c  <Enter>
```

Messages

File not found
The specified filename does not exist in the current or specified location.

Not enough room to merge the entire file
The space available in the edit buffer is not sufficient to hold the entire file named in the T command. Use the Write Lines to Disk (W) command to partially empty the edit buffer.

EDLIN: W

1.0 and later

Write Lines to Disk

Purpose

Writes lines from the edit buffer to the disk.

Syntax

[*n*]W

where:

n is the number of lines to be written to the file.

Description

If the file being edited is too large to fit into the edit buffer, EDLIN ordinarily reads only enough text to fill 75 percent of the buffer when it opens the file, reserving 25 percent of the buffer for changes and additions to the text. The user must then employ the Write Lines to Disk (W) command and the Append Lines from Disk (A) command to transfer successive blocks of text from the disk until the entire file has passed through the edit buffer. The W command causes EDLIN to write lines to the disk file and delete them from the buffer; then the A command can read new lines from the input file, placing them after the end of the text remaining in the buffer.

The *n* parameter specifies the number of lines to be written to the output file; if *n* is omitted or is larger than the number of lines in the edit buffer, EDLIN writes only enough lines to leave the edit buffer about 25 percent full. EDLIN then renumbers the lines remaining in the edit buffer so that the first remaining line becomes line number one.

Examples

To write 200 lines from the edit buffer to disk (effectively deleting those lines from the buffer), type

```
*200W  <Enter>
```

To write lines from the edit buffer to the disk until the edit buffer is only 25 percent full, type

```
*W  <Enter>
```

EXIT

2.0 and later

Terminate Command Processor

Internal

Purpose

Terminates a secondary copy of the command processor.

Syntax

EXIT

Description

Many communications programs, word processors, database managers, and other application programs load and execute a secondary copy of the system's command processor (COMMAND.COM) to let the user carry out MS-DOS commands without losing the context of the work in progress. Secondary copies of the command processor are also commonly used to execute one batch file under the control of another. (For more information about secondary copies of the command processor, *see* USER COMMANDS: COMMAND.)

The EXIT command cancels a secondary command processor. The terminating processor displays no message and control returns directly to the parent program or command processor.

EXIT has no effect on the currently executing command processor if it was loaded with the /P (permanent) switch or if it is the original command processor (the one loaded during system initialization, when the computer was turned on or restarted).

The EXIT command also allows the user to choose Close from the system menu if a COMMAND window is open under Microsoft Windows.

Example

To terminate the currently executing command processor, type

```
C>EXIT  <Enter>
```

Message

Bad command or filename
The EXIT command did not exist in versions earlier than 2.0, so MS-DOS attempted to execute a nonexistent program named EXIT instead.

FC

2.0 and later

Compare Files

External

Purpose

Compares two files and lists the differences on standard output.

Syntax

FC [/A] [/C] [/L] [/LB*n*] [/N] [/*nnnn*] [/T] [/W] [*drive*:]*pathname1* [*drive*:]*pathname2*

or

FC [/B] [*drive*:]*pathname1* [*drive*:]*pathname2*

where:

pathname1	is the name and location of the first file to be compared, optionally preceded by a drive; wildcard characters are not permitted.
pathname2	is the name and location of the second file to be compared, optionally preceded by a drive; wildcard characters are not permitted.
/A	causes FC to abbreviate the output when comparing ASCII text files (version 3.2).
/B	causes a byte-by-byte (binary) comparison; may not be used with any other switch (default when file extension is .EXE, .COM, .SYS, .OBJ, .LIB, or .BIN).
/C	causes FC to ignore case when comparing alphabetic characters.
/L	causes a line-by-line comparison of two ASCII text files (default when file extension is not .EXE, .COM, .SYS, .OBJ, .LIB, or .BIN) (version 3.2).
/LB*n*	sets the size of the internal line buffer to *n* lines (default = 100) (version 3.2).
/N	includes line numbers on the output of an ASCII file comparison (version 3.2).
/*nnnn*	is the number of lines that must match to resynchronize during an ASCII file comparison (default = 2; in versions 2.0 through 3.1, range = 1–9, default = 3).
/T	causes FC to compare tabs in text files literally (default = tabs expanded to spaces, with stops at each eighth character position) (version 3.2).
/W	causes FC to ignore spaces, tabs, and blank lines in text files.

Description

The FC utility compares two text files containing lines of ASCII text delimited by new-line characters or two binary files containing data of any type (such as executable programs).

The differences between the two files are listed on standard output, which defaults to the video display but can be redirected to another character device or a file or can be piped to another program.

The FC program first examines the extensions of the two files being compared and, in most cases, selects the appropriate type of comparison automatically. However, the /B switch can be used to force a binary, or byte-by-byte, comparison of the two files named; the /L switch can be used to force a line-by-line comparison. When the /B switch is present, use of the /L, /N, and /*nnnn* switches causes an error message to be displayed; any other switches in the command line are ignored.

When comparing ASCII text files, FC loads a buffer with sequential sets of lines from each file and compares the two sets. The size of this buffer defaults to 100 lines but can be modified by including the /LB*n* switch in the command line. If differences are found, the name of the first file, the last matched line, and any mismatched lines from that file are displayed, followed by the first rematched line; then the name of the second file, the last matched line, and any mismatched lines are displayed, followed by the first rematched line from that file. The number of consecutive matching lines that must be detected in order for FC to consider the files resynchronized is controlled with the /*nnnn* switch; the default is 2.

If no lines match, if no lines match after the first mismatch, or if the number of mismatched lines exceeds the size of the line buffer, FC displays the message *Resynch failed. Files are too different* (or ****Files are different**** in versions 2.x and 3.0) and terminates.

The /C, /T, and /W switches modify the way in which two text files are compared. The /C switch causes FC to ignore case when comparing alphabetic characters. The /T switch causes FC to compare tab characters (ASCII code 09H) literally, rather than expand them to spaces before comparing corresponding lines. Finally, the /W, or whitespace, switch causes FC to ignore spaces, tabs, and blank lines during the comparison.

The /A and /N switches control the format of the listing of differences between the two text files. The /A switch causes FC to compress the listing of each mismatched set of lines to the first and last lines of each set, separated by ellipsis points. The /N switch causes FC to include the line numbers of the mismatched lines in the display.

During a binary comparison of two files, FC's buffer is reloaded as many times as is necessary to compare the complete files. Unlike the procedure with text-file comparisons, no attempt is made to resynchronize the data if a mismatch is detected and, regardless of the number of mismatches, the comparison process is not terminated. Any differences are displayed with the offset from the start of the file and the actual data from each file. If one file is shorter than the other, FC also displays a warning message at the end of the comparison.

The FC command is present only in MS-DOS. PC-DOS versions 1.0 and later provide a similar function in the COMP command.

Examples

Assume that FILE1.TXT and FILE2.TXT are in the current directory on the disk in the current drive and that they contain the following lines:

FILE1.TXT	FILE2.TXT
`First line.`	`First line.`
`Second line.`	`Second line.`
`Third line.`	`Third line.`
`Fourth line.`	`Fourth line.`
`Fifth line.`	`Sixth line.`
`Sixth line.`	`Fifth line.`
`Seventh line.`	`Seventh line.`
`Eighth line.`	`Eighth line.`
`Ninth line.`	`Ninth line.`
`Tenth line.`	`Tenth line.`

To compare these files line by line, type

```
C>FC FILE1.TXT FILE2.TXT  <Enter>
```

This will result in the following display:

```
***** file1.txt
Fourth line.
Fifth line.
Sixth line.
Seventh line.
***** file2.txt
Fourth line.
Sixth line.
Fifth line.
Seventh line.
*****
```

To compare the same two files and produce an abbreviated listing of differences that includes line numbers, type

```
C>FC /A /N FILE1.TXT FILE2.TXT  <Enter>
```

This will result in the following display:

```
***** file1.txt
    4:  Fourth line.
...
    7:  Seventh line.
***** file2.txt
    4:  Fourth line.
...
    7:  Seventh line.
*****
```

Assume that two binary files, FILE1.BIN and FILE2.BIN, are the same length and contain only the following three differences:

Offset	FILE1.BIN	FILE2.BIN
19H	04H	03H
33H	4AH	4BH
42H	52H	51H

To compare these two binary files, type

```
C>FC /B FILE1.BIN FILE2.BIN  <Enter>
```

This will result in the following display:

```
00000019: 04 03
00000033: 4A 4B
00000042: 52 51
```

Note: The use of the /B switch in this example is optional; binary comparison is the default when .BIN files are compared.

Messages

filename* longer than *filename
After all the corresponding data in the two files was compared, data remained in one of the files.

cannot open *filename* - No such file or directory
The specified file cannot be found or does not exist.

DOS 2.0 or later required
FC does not work with versions of MS-DOS earlier than 2.0.

Incompatible switches
The /B switch was used in combination with one or more of the other switches.

Incorrect DOS version
The version of FC is not compatible with the version of MS-DOS that is running.

no differences encountered
The two files being compared are identical.

out of memory
The available memory in the transient program area is insufficient to compare the two files.

Resynch failed. Files are too different
The number of mismatched lines in an ASCII file comparison exceeded the number of lines that can be loaded into FC's comparison buffer (which by default is 100 lines). Rerun the comparison using the /LB*n* switch to allocate a larger buffer.

usage: fc [/a] [/b] [/c] [/l] [/lbNN] [/w] [/t] [/n] [/NNNN] file1 file2
The command line included an invalid switch or FC was entered without any switches or other parameters.

FDISK

3.2

Configure Fixed Disk

External No Net

Purpose

Configures an MS-DOS partition on a fixed disk. This command is included with PC-DOS beginning with version 2.0.

Syntax

FDISK

Description

A fixed disk can be divided into areas of contiguous tracks, or partitions, that are used by different operating systems. A master control record (partition table) on the disk specifies the ID number and the starting and ending disk tracks for each partition. Each fixed disk can have as many as four partitions, but only one partition can be active (bootable) at any given time.

The FDISK utility is a menu-driven program that adds or deletes an MS-DOS partition on a fixed disk, selects one partition as active, and displays the size and status of all partitions. With most implementations of MS-DOS, each fixed disk can contain only *one* MS-DOS partition.

After an MS-DOS partition is created, the FORMAT command must be used to initialize the partition's directory structure. To make it possible to start the computer from the MS-DOS partition on the fixed-disk drive, the /S switch must be used with FORMAT to transfer the operating-system files and the MS-DOS partition must be the active partition.

Warning: If the MS-DOS partition is deleted, any files stored in the partition are irretrievably lost.

Examples

To display the current partitioning of the fixed disk, type

```
C>FDISK  <Enter>
```

The FDISK utility then displays the following menu:

```
Fixed Disk Setup Program Version 0.02
(C) Copyright Microsoft, 1985.

FDISK Options

Choose one of the following:

    1.  Create DOS Partition
    2.  Change Active Partition

    3.  Delete DOS Partition
    4.  Display Partition Data

Enter choice:[1]

Press ESC to return to DOS
```

Note: A fifth option, *Select Next Fixed Drive*, will appear if more than one fixed disk is installed in the system.

Choose option 4 (*Display Partition Data*). FDISK then displays the partition data for the disk in the following form:

```
Display Partition Information

Partition Status  Type   Start  End  Size
    1       A      DOS       0  613   614
Total disk space is 614 cylinders.

Press ESC to return to FDISK Options
```

Assume that the low-level (hardware) formatting for fixed-disk drive C has just been completed by using the drive manufacturer's setup utility. To establish a bootable MS-DOS partition on the disk, type

```
A>FDISK  <Enter>
```

When the menu is displayed, press Enter to choose option 1 (*Create DOS Partition*). FDISK responds with the following message:

```
Create DOS Partition

Do you wish to use the entire fixed
disk for DOS (Y/N).................?[Y]

Press ESC to return to FDISK Options
```

To partition the entire fixed disk for MS-DOS, press Enter to select *Y* (the default). When the FDISK main menu is again displayed, choose option 4 (*Display Partition Data*) to verify that the MS-DOS partition has in fact been established on the fixed disk.

Messages

***n* is not a choice. Please enter Y or N.**
The response to an FDISK prompt requiring a yes or no answer was not *Y* or *N*.

***n* is not a choice. Please enter a choice**
The response to an FDISK prompt requiring a number was not in the proper range or was not a number.

DOS partition created
A new MS-DOS partition has been established on the fixed disk. Use the FORMAT utility to create a directory structure in that partition.

DOS partition deleted
The previously existing MS-DOS partition on the fixed disk has been deleted. Any files contained in the partition are irretrievably lost.

DOS 2.0 or later required
FDISK does not work with versions of MS-DOS earlier than 2.0.

Do you wish to use the entire fixed
disk for DOS (Y/N).............?[Y]
Option 1, *Create DOS Partition*, has been chosen from the main menu. Respond with *Y* or press Enter to use all available cylinders for a single DOS partition; respond with *N* to specify that only part of the fixed disk should be used.

Enter starting cylinder number..:[*n*]
Option 1, *Create DOS Partition*, has been chosen from the main menu and the user has responded *N* to the *Do you wish to use the entire fixed disk for DOS?* prompt. This message then prompts for the starting cylinder number of the DOS partition being created.

Enter the number of the partition you
want to make active...............:[*n*]
Option 2, *Change Active Partition*, has been chosen from the main menu and this message prompts the user to enter the number of the partition that will become the active partition.

Error loading operating system
An error occurred while attempting to start the system from the fixed disk. Attempt to restart the system. If that fails, start the system from a floppy disk and use the SYS command to copy a new set of the operating-system files to the fixed disk.

Error reading fixed disk
An unrecoverable hardware error was encountered while FDISK was reading data from the fixed disk. The disk may require a low-level (hardware) formatting operation before FDISK can be used; this is usually performed with a special utility program provided by the drive manufacturer.

Error writing fixed disk
An unrecoverable hardware error was encountered while FDISK was writing the new partition control record to the fixed disk. Test the fixed disk with hardware diagnostics before further use.

Fixed disk already has a DOS partition.
The specified fixed disk already contains an MS-DOS partition. Be sure that the correct fixed disk has been selected before proceeding.

Incorrect DOS version
The version of FDISK is not compatible with the version of MS-DOS that is running.

Invalid partition table
The fixed disk's partition table is invalid and the operating system could not be loaded from the fixed disk during system initialization. Restart the computer using a floppy disk and rerun FDISK to determine and correct the problem.

Missing operating system
The DOS partition is the active partition, but it does not contain the operating system. (This message occurs only during system startup.) Use the SYS command to install the operating system.

No DOS partition to delete.
The fixed disk does not contain an MS-DOS partition.

No fixed disks present
FDISK cannot detect a fixed disk in the system. This may reflect a hardware problem with the fixed disk or its controller.

No partitions defined.
This informational message is displayed after the user has chosen option 4, *Display Partition Data*, to indicate that no partitions are currently defined.

No partitions to make active
The fixed disk has not been previously partitioned using FDISK; therefore, an active partition cannot be selected.

No space for a *nnn* cylinder partition.
The fixed disk does not have enough free cylinders to create the desired partition.

No space to create a DOS partition.
The fixed disk does not have enough free cylinders to create an MS-DOS partition.

Partition *n* is already active
The selected partition is already active (bootable); therefore, no action was taken.

Partition *n* made active
This informational message indicates that the selected partition has been made the active partition.

System will now restart
Insert DOS diskette in drive A:
Press any key when ready...
The DOS partition has successfully been created. Strike any key and the system will restart from the disk in drive A.

The current active partition is *n*.
This informational message indicates which partition is currently bootable.

The table partition can't be made active.
The master partition record cannot be made bootable.

Total disk space is *nnn* cylinders.
This informational message indicates the total number of cylinders on the fixed disk.

Total disk space is *nnn* cylinders.
Maximum available space is *nnn*
cylinders at *n*.
The user has responded *N* to the *Do you wish to use the entire fixed disk for DOS?* prompt and this informational message indicates how much space is available for the DOS partition.

Warning: Data in the DOS partition
will be lost. Do you wish to
continue.......................?[N]
If the MS-DOS partition is deleted, all files within the partition are lost. Be sure that the files are backed up to another disk before proceeding. Respond with *N* to return to the FDISK main menu; respond with *Y* to delete the DOS partition and lose any files within it.

FIND

2.0 and later

Find Character String

External

Purpose

Searches the character stream from a file or from standard input for a string and displays any lines that contain the string on standard output.

Syntax

FIND [/C] [/N] [/V] *"string"* [[*drive*:][*path*]*filename*] [[*drive*:][*path*]*filename* ...]

where:

string is the character string to be searched for, always enclosed in quotation marks; case is significant.
filename is the name of the file to be searched, optionally preceded by a drive and/or path; wildcard characters are not permitted.
/C displays only the count of the lines containing *string*.
/N includes the relative line number with each line.
/V displays only those lines that do *not* contain *string*.

Description

The FIND command searches for all occurrences of a specified string in one or more files (or from standard input). Normally, FIND copies each line in which the string is found to standard output, which defaults to the video display but can be redirected to a file or another character device or can be piped to another program.

The string to be searched for must be enclosed in quotation marks. If the search string itself contains sets of quotation marks, each of those sets of quotation marks must be surrounded by an additional set of quotation marks. FIND's string search is case sensitive.

The search string can be followed by the names of one or more source files; these filenames cannot include wildcards. If no filename is supplied, FIND reads lines from standard input; unless input has been redirected from a file or from the output of another program, this means that FIND reads input from the keyboard. (Keyboard input is terminated by pressing Ctrl-Z or F6 followed by Enter.)

The /C switch counts the total number of lines in which the string appears and sends the count, rather than the lines themselves, to standard output. If the /C switch is used with /V, only the total count of lines that do *not* contain the specified search string is displayed. If both /C and /N are included in the same FIND command, the /N is ignored.

The /N switch includes a relative line number with each line sent to standard output. This is especially helpful when the output of FIND is to be used as a guide to editing the files.

The /V switch reverses the action of FIND so that it copies to standard output all lines that do *not* include the specified string.

Examples

To find and display all lines in the files BREAK.ASM, TALK.ASM, and SHELL.ASM that contain the string *es:*, type

```
C>FIND "es:" BREAK.ASM TALK.ASM SHELL.ASM  <Enter>
```

To find and display all lines in the file STORY.TXT that contain the string *he said "no"*, type

```
C>FIND "he said ""no""" STORY.TXT  <Enter>
```

To search the file \SOURCE\MENUMGR.ASM on the current drive and display all lines that do not contain the string *Error*, type

```
C>FIND /V "Error" \SOURCE\MENUMGR.ASM  <Enter>
```

To obtain a listing on the printer of the lines in the file SHELL.ASM in the current directory of the current drive that contain the string *proc*, including line numbers, type

```
C>FIND /N "proc" SHELL.ASM > PRN  <Enter>
```

To search for all lines that contain two strings, pipe the output of one FIND command to be the input of another. For example, to find only those lines in the file MENUMGR.ASM in the current directory of the current drive that contain both the strings *MOV* and *AX*, type

```
C>FIND "MOV" MENUMGR.ASM ¦ FIND "AX"  <Enter>
```

Messages

----------*filename*
This informational message gives the name of the file that is currently being searched.

FIND: Access denied
The specified file is locked or being accessed by another application.

FIND: File not found *filename*
The specified file does not exist or the path or drive is not correct.

FIND: Invalid number of parameters
The command line did not include a search string.

FIND: Invalid Parameter *option*
The command line included an invalid switch.

FIND: Read error in *filename*
A disk error occurred during processing of the specified file.

FIND: Syntax error
The command line included an invalid search string. The string must be enclosed in quotation marks.

Incorrect DOS version
The version of FIND is not compatible with the version of MS-DOS that is running.

FORMAT

1.0 and later

Initialize Disk

External No Net

Purpose

Prepares a disk for use by initializing the directory and file allocation table (FAT).

Syntax

FORMAT [*drive*:] [/S] (versions 1.x)

or

FORMAT [*drive*:] [/O] [/V] [/S] (versions 2.0–3.1)

or

FORMAT *drive*: [/1] [/4] [/8] [/N:*n*] [/T:*n*] [/V] [/S] (version 3.2)

or

FORMAT *drive*: [/1] [/B] [/N:*n*] [/T:*n*] (version 3.2)

where:

drive is the location of the disk to be formatted.
/1 formats a single-sided disk in a double-sided disk drive.
/4 formats a standard double-sided, double-density disk (360 KB) on a quad-density disk drive.
/8 formats a disk with 8 sectors per track.
/B formats a disk with 8 sectors per track and preallocates space for the hidden operating-system files.
/N:*n* formats a disk with *n* sectors per track.
/O formats a disk that is compatible with PC-DOS versions 1.x.
/S creates a system (bootable) disk; for most implementations of FORMAT, this must be the last switch in the command line.
/T:*n* formats a disk with *n* tracks.
/V allows a volume label to be assigned to the disk after formatting.

Note: Each OEM determines which switches will be supported by the FORMAT utility included with the versions of MS-DOS sold with its computers.

Description

The FORMAT command effectively erases any existing data on a disk and creates a new root directory and file allocation table. Each sector of the disk is checked for defects and unusable sectors are marked so that they will not be assigned to files.

If the *drive* parameter is not supplied, the current or default drive is formatted. (A drive letter *must* be specified with version 3.2.) With versions 3.0 and later, the FORMAT program displays a warning if the drive to be formatted is a fixed disk and asks for confirmation before continuing.

When the formatting operation is complete, FORMAT displays the total amount of disk space, the number of bytes lost to defective sectors, the space reserved for or occupied by the hidden operating-system files (if the /B or /S switch was used), and the remaining free disk space. If a floppy disk was formatted, FORMAT then prompts the user to select between formatting another disk and returning to MS-DOS.

Normally, the type of disk drive determines the format that is given to a disk. For example, if a disk is formatted in a standard double-sided, double-density drive, the format defaults to double-sided, 40 tracks per side, 9 sectors per track. The version-specific default formats are 9 or 15 sectors per track with versions 3.0 and later, depending on the drive type; 9 sectors per track with versions 2.x; and 8 sectors per track with versions 1.x. The /1, /4, /8, /N:*n*, and /T:*n* switches can be used to override the default format in some cases. (Not all combinations of /N:*n* and /T:*n* are supported on all hardware.)

Note: A disk formatted with the /4 switch might not be reliably read on a single- or double-sided double-density drive.

The /S switch creates a system (bootable) disk that contains a copy of the operating system. After the format operation is complete, the two hidden files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS) and the nonhidden file COMMAND.COM are copied to the newly formatted disk. Most implementations of FORMAT require that the /S switch, if used, be the last switch in the command line.

The /V switch allows a volume label to be assigned to the new disk. After formatting is complete, FORMAT prompts the user for a volume name, which can be as many as 11 characters. (The characters * ? / ¦ . , ; : + = < > [ ] and tab are not permitted in a volume label.) Volume labels are displayed by the DIR, CHKDSK, TREE, and VOL commands and, with MS-DOS versions 3.1 and later and PC-DOS versions 3.0 and later, can be modified with the LABEL command after the disk has been formatted.

The /O switch causes FORMAT to write an 0E5H byte at the start of each directory entry so that the resulting disk is compatible with MS-DOS and PC-DOS versions 1.x.

The /B switch formats a disk for 8 sectors per track and reserves room on the disk for the operating-system files. The operating system can then be transferred to the disk with the SYS command to make the disk bootable. The /B switch cannot be used in the same FORMAT command line as the /V or /S switch.

Warning: Disks in drives affected by an ASSIGN, JOIN, or SUBST command should not be formatted. Disks cannot be formatted over a network.

Return Codes

0 The FORMAT operation was successful.
3 The program was terminated by entry of a Ctrl-C or Ctrl-Break.
4 The program was terminated because of a fatal system error (any error other than 0, 3, or 5).
5 The program was terminated by an *N* response to the fixed-disk prompt *Proceed with FORMAT (Y/N)?*

Note: Return codes are available with MS-DOS version 3.2.

Examples

To format the disk in drive B, type

```
C>FORMAT B:  <Enter>
```

In response, FORMAT displays the following message:

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

With versions earlier than 3.2, FORMAT then displays the message

```
Formatting ...
```

after the Enter key is pressed, to show that the formatting operation is in progress. With version 3.2, FORMAT displays the message

```
Head: n Cylinder: nn
```

instead, to show the progress of the formatting operation. With all versions, FORMAT displays the following messages if the formatting operation is successful:

```
Format complete
    362496 bytes total disk space
    362496 bytes available on disk

Format another (Y/N)?
```

The byte values may vary depending on the drive type or the switches used in the command line. If bad sectors were encountered during the format operation, FORMAT also displays the number of bytes in bad sectors.

Note: The *Format complete* message overwrites the head/cylinder status line but is appended to the *Formatting ...* status line.

To format and assign a volume label to the disk in drive B, type

```
C>FORMAT B: /V  <Enter>
```

After the usual formatting messages, FORMAT prompts as follows:

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

The user can then enter a volume name of as many as 11 characters (except * ? / ¦ . , ; : + = < > [ ] or tab), followed by a press of the Enter key.

To format the disk in drive B and make it a system (bootable) disk, type

```
C>FORMAT B: /S  <Enter>
```

FORMAT initializes the disk in the usual manner and then copies the two files containing the operating system (IO.SYS and MSDOS.SYS or IBMBIO.COM and IBMDOS.COM) and the file COMMAND.COM onto the disk. When the formatting operation is completed on a 360 KB floppy disk, the following messages appear:

```
Format complete
System transferred

    362496 bytes total disk space
     62464 bytes used by system
    300032 bytes available on disk

Format another (Y/N)?
```

The number of bytes used by the system will vary with the version of MS-DOS in use.

Messages

***n* bytes total disk space**
***n* bytes used by system**
***n* bytes in bad sectors**
***n* bytes available on disk**
When formatting is complete, FORMAT displays this message with information about space available on the disk. The *bytes used by system* line will not appear if the /S switch was not specified; the *bytes in bad sectors* line will not appear if no bad sectors were found.

Attempted write-protect violation
The disk to be formatted is write protected. Remove the write-protect tab and respond with a *Y* to the *Format another (Y/N)?* prompt.

Cannot find System Files
The /S switch was used and FORMAT was unable to find the necessary system files in the default drive or in drive A.

Cannot FORMAT a Network drive
An attempt was made to format a disk in a drive that has been assigned to a network.

Cannot format an ASSIGNed or SUBSTed drive.
An attempt was made to format a disk in a drive affected by an ASSIGN or SUBST command.

Disk unsuitable for system disk
Defective sectors were detected on the tracks where the operating-system files would normally reside on a bootable disk. Such a disk should be used only for data files, if at all.

Drive letter must be specified
A drive letter must be specified when using version 3.2.

Drive not ready
The floppy-disk drive is empty or the drive door is not closed.

Enter current Volume Label for drive *X*:
The specified drive is a fixed disk, so FORMAT prompts the user to enter the current volume label for verification.

Error in IOCTL call
An internal system error occurred when a pre-version-3.2 block-device driver was used with version 3.2 of FORMAT.

Error reading partition table
FORMAT was unable to read the fixed disk's partition table. Use FDISK on the fixed disk and then try the FORMAT command again.

Error writing directory
FORMAT was unable to create a directory on the disk it is attempting to format. The disk is defective.

Error writing FAT
FORMAT was unable to create the FAT on the disk it is attempting to format. The disk is defective.

Error writing partition table
FORMAT was unable to write the fixed disk's partition table. Use FDISK on the fixed disk and then try the FORMAT command again.

Format another (Y/N)?
At the end of a successful formatting operation or after a nonfatal error, this prompt offers the user the opportunity to format another disk using the same switches specified in the original FORMAT command. Respond with *Y* to format another disk; respond with *N* to return to MS-DOS.

Format complete
The formatting operation has ended. This message contains a number of space characters after it and is printed over the top of the head/cylinder status message, effectively erasing it.

Format failure
The formatting operation was not successful. (This message is usually preceded by another message telling the user why the format failed.) This message contains a number of space characters after it and is printed over the top of the head/cylinder status message, effectively erasing it.

Format not supported on drive *X*:
Device parameters that the computer cannot support were specified in the FORMAT command line.

Formatting . . .
This informational message indicates that the FORMAT operation is in progress (versions 1.0 through 3.1).

Head: *n* Cylinder: *nn*
This informational message indicates the progress of the FORMAT command during the formatting operation (version 3.2).

Incorrect DOS version
The version of FORMAT is not compatible with the version of MS-DOS that is running.

Insert DOS disk in drive *X*:
and strike ENTER when ready
The /S switch was specified in the FORMAT command line and the disk containing the FORMAT command does not also contain the hidden system files.

Insert new diskette for drive *X*:
and strike ENTER when ready
This prompt allows the user to change disks before the FORMAT operation continues.

Insufficient memory for system transfer
The command line included the /S switch, but available RAM is insufficient to hold the system files during the FORMAT operation.

Invalid characters in volume label
Certain characters (* ? / ¦ . , ; : + = < > [ ] and tab) are not allowed in a volume name.

Invalid device parameters from device driver
The DEVICE or DRIVPARM device-driver parameters in the CONFIG.SYS file were incorrectly set or the fixed disk specified in the command line was formatted using MS-DOS versions 2.x without first running FDISK. FORMAT displays this message when the number of hidden sectors is not evenly divisible by the number of sectors per track (meaning that the partition does not start on a track boundary).

Invalid drive specification
The drive specified after the FORMAT command is not a valid drive.

Invalid media or Track 0 bad - disk unusable
One of the switches supplied in the command line is not valid for the drive containing the disk to be formatted (for example, the /8 switch for a quad-density floppy disk) or track 0 of the disk being formatted is unusable to the point that FORMAT is unable to create a directory or file allocation table (FAT).

Invalid parameter
One of the switches supplied in the command line is not valid or is not supported by the version of FORMAT being used.

Invalid volume ID
The volume label entered in response to the *Enter current Volume Label for drive X:* prompt was not the same as the current volume label. Use the VOL command to determine the current volume label.

Non-System disk or disk error
Replace and strike any key when ready
The command line contained a /S or /B switch, but the source disk does not contain the operating-system files.

Not a block device
The drive containing the disk to be formatted is not recognized by MS-DOS as a valid block device.

Parameters not compatible
Switches that cannot be used together were specified in the command line.

Parameters not compatible with fixed disk
One of the switches specified in the command line is not compatible with the specified drive.

Parameters not supported
One of the parameters specified in the command line is not supported by the version of FORMAT being used.

Parameters not Supported by Drive
The device driver for the specified drive does not support generic IOCTL function requests.

Re-insert diskette for drive *X*:
This message prompts the user to reinsert the disk being formatted into the specified drive.

System transferred
The system files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS) and the file COMMAND.COM have been successfully transferred to the newly formatted disk.

Too many open files
FORMAT was unable to write the volume label because insufficient system file handles were available. Increase the value of FILES in the CONFIG.SYS file.

Volume label (11 characters, ENTER for none)?
After formatting a disk with the /V option, FORMAT offers the user the opportunity to enter a volume label for the disk.

Unable to write BOOT
The first track of the disk or MS-DOS partition is bad and cannot be made bootable.

WARNING, ALL DATA ON NON-REMOVABLE DISK
DRIVE *X*: WILL BE LOST!
Proceed with Format (Y/N)?
If a fixed disk is specified as the disk to be formatted, FORMAT warns the user and gives the opportunity to cancel the FORMAT command (versions 3.0 and later).

GRAFTABL

3.0 and later

Load Graphics Character Set

External

Purpose

Installs a RAM-resident table of bitmaps that defines the screen appearance of character codes 128 through 255 in graphics mode.

Syntax

GRAFTABL

Description

On IBM PCs and compatibles in graphics display modes, the video-display BIOS routines (Interrupt 10H) display characters by writing bitmapped matrices of dots to the display. The dot pattern of each screen character's matrix is defined by an entry in a table of bitmaps. The table of bitmaps for the regular ASCII characters, coded 0 through 7FH (0–127), is permanently located in ROM and is always available for use by the system's video driver. The GRAFTABL utility contains a similar table of bitmaps for the upper (extended) characters, coded 80H through 0FFH (128–255). The GRAFTABL command loads this table into RAM and places the address of the table in the vector for Interrupt 1FH.

The GRAFTABL command is not needed for the IBM PCjr or for an enhanced graphics adapter; their ROM BIOS already contains tables of bitmaps for the extended character set.

GRAFTABL is a terminate-and-stay-resident (TSR) program; therefore, its installation reduces the amount of RAM available for use by application programs.

The GRAFTABL command can be executed only once after the computer has been turned on or restarted. An attempt to execute it again will result in an informational message stating that the graphics characters are already loaded.

Example

To load the table of bitmaps for characters 80H through 0FFH (128–255) for use in graphics mode, type

```
C>GRAFTABL  <Enter>
```

Messages

DOS 2.0 or later required
GRAFTABL does not work with versions of MS-DOS earlier than 2.0.

Graphics characters already loaded
The GRAFTABL command has already been executed since the system was turned on or restarted.

Graphics characters loaded
The table of bitmaps has been successfully loaded into RAM and the interrupt vector that points to the table has been initialized.

Incorrect DOS version
The version of GRAFTABL is not compatible with the version of MS-DOS that is running.

GRAPHICS

3.2

Load Graphics Screen-Dump Program

External

Purpose

Installs a resident program that can dump screen contents to the printer in graphics mode. This command is also available with PC-DOS versions 2.0 and later.

Syntax

GRAPHICS (PC-DOS 2.x)

or

GRAPHICS [*printer*] [/B] [/R] (PC-DOS 3.0 and above)

or

GRAPHICS [*printer*] [/B] [/C] [/F] [/P *port*] [/R] (MS-DOS 3.2)

where:

printer is the type of printer to be supported, from the following list:

COLOR1	IBM Personal Computer Color Printer with black ribbon
COLOR4	IBM Personal Computer Color Printer with red-green-blue-black (RGB) ribbon
COLOR8	IBM Personal Computer Color Printer with cyan-magenta-yellow-black (CMY) ribbon
COMPACT	IBM Personal Computer Compact Printer
GRAPHICS	IBM Personal Computer Graphics Printer or compatible (the default)

/B prints the background in color; valid only with the COLOR4 and COLOR8 printers.

/C centers the printout on the page.

/F flips (rotates) the printout 90 degrees.

/P *port* specifies which port the printer is attached to (1–3, where 1 = LPT1, 2 = LPT2, and 3 = LPT3).

/R prints the image as it appears on the screen (white characters on a black background) rather than reversed (the default, black characters on a white background).

Description

The default system routine for dumping the screen to the printer (invoked by Shift-PrtSc) cannot interpret the display in graphics modes. The GRAPHICS command loads a more sophisticated routine that can dump CGA-compatible graphics displays to several models of IBM graphics printers or compatibles.The GRAPHICS command is not compatible with the Hercules monochrome graphics card or with an enhanced graphics adapter in its enhanced display modes.

If the display is in 640 x 200 graphics mode, the screen dump is printed sideways (rotated 90 degrees). A 320 x 200 graphic can be rotated manually by specifying the /F switch in the command line; however, the image will be elongated horizontally. A rotated image is printed along the left side of the page, which is actually the top of the page in terms of image orientation. The /C option can be used to center a rotated 320 x 200 image on the page.

When used with a printer with a black ribbon, GRAPHICS produces screen dumps with as many as four shades of gray to represent the colors. When used with a printer with a color ribbon (type COLOR4 or COLOR8), GRAPHICS prints all the colors except the background color. With printer types COLOR4 and COLOR8, the /B switch can be used to print the background color also.

Ordinarily, the screen image being dumped is reversed from its appearance on the screen; that is, the light areas on the screen are dark on the printed output and vice versa. The /R switch produces a screen dump that is not reversed in this manner.

If the *printer* parameter is not included in the command line, the GRAPHICS program assumes an IBM Personal Computer Graphics Printer or compatible.

If two or more printers are attached to the system, the /P switch can be used to specify which printer GRAPHICS should use.

The GRAPHICS command is a terminate-and-stay-resident (TSR) program; therefore, its installation reduces the amount of RAM available for use by application programs.

Examples

To load the graphics printing program for use with an IBM Personal Computer Graphics Printer or compatible connected to LPT2, type

```
C>GRAPHICS /P 2  <Enter>
```

Note: A tab, a semicolon character (;), or an equal sign (=) can be used between the /P and the port number instead of a space.

To load the graphics printing program for use with the IBM Personal Computer Color Printer with an RGB ribbon and specify that the background color be printed, type

```
C>GRAPHICS COLOR4 /B  <Enter>
```

To load the graphics printing program for use with the IBM Personal Computer Compact Printer and specify that the images be printed sideways and centered on the page, type

```
C>GRAPHICS COMPACT /F /C  <Enter>
```

Messages

DOS 2.0 or later required
GRAPHICS does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of GRAPHICS is not compatible with the version of MS-DOS that is running.

Unrecognized printer
The printer type specified in the command line is invalid or the printer is not supported.

Unrecognized printer port
The port specified with the /P switch is not a number in the range 1 through 3 or an invalid separator character was used.

JOIN

Join Disk to Directory

3.0 and later

External No Net

Purpose

Joins the directory structure of a disk drive to a subdirectory on another drive.

Syntax

JOIN [*drive1*: *drive2*:*path*]

or

JOIN *drive1*: /D

where:

drive1	is the drive whose directory structure will be joined to a subdirectory of another drive.
drive2:*path*	is the drive and directory that will be used to reference files on *drive1*.
/D	cancels the effect of a previous JOIN command on *drive1*.

Description

The JOIN command allows the directory structure of a disk in one drive to be joined, or spliced, into an empty subdirectory of a disk in another drive. After a JOIN, the entire directory structure of the disk in *drive1*, starting at the root, together with all the files that it contains, appears to be the directory structure of the specified subdirectory on the disk in *drive2*; the drive letter for *drive1* is no longer available. If the directory at the end of the path on *drive2* already exists, it must not contain any files; if it does not exist, JOIN will attempt to create it.

The current directory status of *drive1* has no effect on the JOIN operation. Regardless of which directory or subdirectory is active when the JOIN command is entered, the entire directory structure, including the root directory, is joined to the subdirectory on the disk in *drive2*.

The /D switch cancels any previous JOIN command for a specific drive.

If the JOIN command is entered without parameters, it displays a list of all joins currently in effect.

Warning: The JOIN command should not be used on drives affected by a SUBST or ASSIGN command. Similarly, the BACKUP, RESTORE, FORMAT, DISKCOPY, and DISKCOMP commands should not be used on drives affected by the JOIN command. Drives that have been redirected over a network cannot be joined.

Examples

To join drive B to the subdirectory \DRIVEB on drive C, type

```
C>JOIN B: C:\DRIVEB  <Enter>
```

A subsequent JOIN command without parameters displays

```
B: => C:\DRIVEB
```

To then list the files in the root directory of the disk in drive B, type

```
C>DIR C:\DRIVEB  <Enter>
```

To cancel a previous JOIN command affecting drive B, type

```
C>JOIN B: /D  <Enter>
```

Messages

Cannot JOIN a network drive
A drive assigned to a network cannot be joined to another drive.

Directory not empty
A drive cannot be joined to a directory that already contains files.

DOS 2.0 or later required
JOIN does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of JOIN is not compatible with the version of MS-DOS that is running.

Incorrect number of parameters
There were missing, extra, or incorrect parameters in the command line.

Invalid parameter
A drive cannot be joined to the root directory of any drive.

Not enough memory
The available system memory is insufficient for MS-DOS to run the JOIN command.

KEYB*xx*

3.2

Define Keyboard

External

Purpose

Installs a table that defines the translation of keys to the extended character codes, replacing the default table in the ROM BIOS. This command is included with PC-DOS beginning with version 3.0.

Syntax

KEYB*xx*

where:

xx is a code that selects a keyboard configuration:

DV	Dvorak keyboard (MS-DOS only)
FR	French
GR	German
IT	Italian
SP	European Spanish
UK	United Kingdom English

Note: KEYB*xx* is hardware dependent; therefore, implementation of this command may vary for different OEM versions of MS-DOS.

Description

The KEYB*xx* utility configures the keyboard for use with a language other than United States English, making available special characters that are appropriate for the specified country's language and currency. These special characters are represented by the extended character codes (128–255) that correspond to the characters implemented on the OEM's display adapter. (Both the KEYB*xx* and the GRAFTABL commands must be used to make these characters available in graphics modes on a color/graphics adapter.)

After KEYB*xx* is loaded, special accented characters not part of the language in use are also available through the use of dead keys — keys that are pressed and released before the letter key is pressed. The following dead keys are available on a United States English keyboard for an IBM PC, PC/XT, PC/AT, or strict compatible:

Keyboard Program	**Dead Key**	**Resulting Accent**
KEYBGR (Germany)	+	`
	=	´
KEYBFR (France)	[	^
	{	¨
KEYBSP (Spain)	[	´
	]	`
	{	¨
	}	^
KEYBUK (United Kingdom)	Not supported	
KEYBIT (Italy)	Not supported	

The dead-key combinations supported are

Keyboard Program	**Combinations Supported**
Germany	á é É í ó ú à è ì ò ù
France	ä Ä ë ï ö Ö ü Ü ÿ â ê î ô û
Spain	ä Ä ë ï ö Ö ü Ü ÿ á é É í ó ú à è ì ò ù â ê î ô û
United Kingdom	Dead key not supported
Italy	Dead key not supported

On an IBM PC, PC/XT, PC/AT, or strict compatible, the key sequence Ctrl-Alt-F1 can be used at any time to return the keyboard to the default (United States English) configuration; the sequence Ctrl-Alt-F2 then returns the keyboard to the selected configuration.

KEYB*xx* should be loaded only once during an MS-DOS session; the computer should be restarted if KEYB*xx* is loaded for use with a different language.

KEYB*xx* is a terminate-and-stay-resident (TSR) utility and therefore reduces the amount of memory available to transient application programs (by approximately 2 KB). The only way to reclaim this memory is to restart the system.

Example

To configure the keyboard for Germany, type

```
C>KEYBGR  <Enter>
```

Messages

Bad command or filename
The selected keyboard does not exist or the program that configures the keyboard is not present on the disk.

Incorrect DOS version
The version of KEYB*xx* is not compatible with the version of MS-DOS that is running.

LABEL

3.1 and later

Modify Volume Label

External No Net

Purpose

Adds, alters, or deletes a volume label on a disk. This command is included with PC-DOS beginning with version 3.0.

Syntax

LABEL [*drive*:][*label*]

where:

drive is any valid disk drive.
label is a name up to 11 characters long.

Description

With MS-DOS versions 2.0 and later, each disk can have a name called a volume label, which is implemented as a special type of entry in the disk's root directory. With MS-DOS versions 2.x, this volume label can be assigned to a disk only at the time the disk is formatted, using the FORMAT command's /V switch. However, with PC-DOS versions 3.0 and later and MS-DOS versions 3.1 and later, the volume label can be added, modified, or deleted at any time using the LABEL command. (A disk's volume label can be displayed with the VOL command; the label is also included as part of the output from the CHKDSK, DIR, and TREE commands.)

If a new volume name is included in the LABEL command line, the disk's label is changed immediately. If LABEL is entered alone or with only a drive letter, a message is displayed giving the current volume label of the disk in the specified drive (or the default drive, if no drive letter is given) and prompting the user for a new label. (A volume label can be from 1 to 11 characters; it cannot contain any of the characters * ? / \ ¦ . , ; : + = < > [ ] or tab.) If no new volume name is supplied (the user did not type a volume label before pressing Enter), LABEL prompts the user to indicate whether the previous volume label should be deleted. Existing files on the disk are in no way affected by the LABEL command.

The LABEL command cannot be used on a network drive. With MS-DOS version 3.2, the LABEL command also cannot be used on a disk in a drive that is affected by an ASSIGN or SUBST command.

Examples

To give the volume label PAYROLL to the disk in drive B, type

```
C>LABEL B:PAYROLL  <Enter>
```

Note that LABEL immediately overwrites any existing volume label on drive B with the new name; no warning of an existing volume label is given.

To remove the volume label LEDGER from the disk in drive A, type

```
C>LABEL A:  <Enter>
```

The LABEL command displays

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

Press the Enter key to receive the additional prompt

```
Delete current volume label (Y/N)?
```

Then respond with *Y* and Enter to remove the volume label from the disk in drive A.

Messages

Cannot LABEL a Network drive
The disk drive specified in the command line cannot be a network drive.

Cannot LABEL a SUBSTed or ASSIGNed drive
The disk drive specified in the command line is currently affected by a SUBST or ASSIGN command (MS-DOS version 3.2).

Delete current volume label (Y/N)?
No volume label was entered in response to the volume-label prompt and a volume label already exists on the disk. Respond with *Y* to delete the current label; respond with *N* to terminate the command.

Incorrect DOS version
The version of LABEL is not compatible with the version of MS-DOS that is running.

Invalid characters in volume label
The characters * ? / \ ¦ . , ; : + = < > [ ] and tab cannot be part of a volume label.

Invalid drive specification
The drive specified in the command line is not valid or does not exist in the system.

No room in root directory
The root directory of the disk in the designated drive is full and a volume label cannot be added. Delete a file or subdirectory from the root directory to make room for the label.

Too many files open
LABEL was unable to write the volume label because no system file handles were available. Increase the value of FILES in the CONFIG.SYS file.

Volume in drive *X* has no label
Volume label (11 characters, ENTER for none)?

or

Volume in drive *X* is *xxxxxxxxxxx*
Volume label (11 characters, ENTER for none)?
This informational message informs the user of the current volume label and prompts the user to add, change, or delete it.

MKDIR or MD

2.0 and later

Make Directory

Internal

Purpose

Creates a new directory.

Syntax

MKDIR [*drive*:][*path*] *new_directory*

or

MD [*drive*:][*path*] *new_directory*

where:

new_directory	is a valid directory name, optionally preceded by an existing path and/or a disk drive.

Description

The MKDIR command creates a directory, adding a branch to the hierarchical directory structure of the disk. If the name of the new directory is preceded by a path, indicating that the new directory is to be a subdirectory of that path, the specified path must already exist.

If *new_directory* is not preceded by an existing path or a backslash character (\), it is presumed to be relative to the current directory. If *new_directory* is preceded by a backslash alone, the directory created will be a subdirectory of the root directory, regardless of the current directory. The length of the full path (including *new_ directory*) must not exceed 63 characters.

Warning: The MKDIR command should not be used to create new directories on drives affected by an ASSIGN, JOIN, or SUBST command.

Examples

To create a directory named SOURCE in the current directory of the disk in the current drive, type

```
C>MKDIR SOURCE  <Enter>
```

or

```
C>MD SOURCE  <Enter>
```

To create a directory named LETTERS in the existing directory named WORD (which is a subdirectory of the root directory) on the disk in drive D, type

```
C>MKDIR D:\WORD\LETTERS  <Enter>
```

or

```
C>MD D:\WORD\LETTERS  <Enter>
```

Messages

Invalid drive specification
The drive specified in the command line is not valid or does not exist in the system.

Invalid number of parameters
The name of the new directory was not included in the MKDIR command line.

Unable to create directory
The specified directory cannot be created. This may be caused by a full disk (if the new directory would cause the current directory to be extended), a full root directory (if the new directory's parent is the root directory), the existence of a file or directory with the same name, or an invalid *new_directory* name.

MODE

3.2

Configure Device

External

Purpose

The MODE command has four distinct uses:

- To reconfigure a printer attached to a parallel port (LPT1, LPT2, or LPT3) for printing at 80 or 132 characters per line, 6 or 8 lines per inch, or both (if the printer supports these features). In this form, MODE can also be used to select a parallel printer other than the one attached to LPT1 for use as the default printer.
- To select another display or reconfigure the current display. Reconfiguration includes changing between 40-column and 80-column display, changing between monochrome and color display, centering the display on the screen, or any combination of these.
- To configure the baud rate, parity, and number of databits and stop bits of a serial communications port (COM1 or COM2) for use with a specific printer, modem, or other serial device.
- To redirect printer output from a parallel port to one of the serial ports, so that the serial port becomes the system's default printer port.

Because the syntax for each of these uses of MODE is different, they are discussed separately on the following pages.

Although each form of the MODE command can be issued at the system prompt, MODE commands are commonly used within the AUTOEXEC.BAT file to automatically perform any necessary reconfiguration each time the system is turned on or restarted.

The MODE command is included with PC-DOS beginning with version 1.0.

Message

Incorrect Version of MODE
The version of MODE is not compatible with the version of MS-DOS that is running.

MODE

3.2

Configure Printer

External

Purpose

Sets characteristics for IBM-compatible printers connected to a parallel printer port (LPT1, LPT2, or LPT3). This form of the MODE command is included with PC-DOS beginning with version 1.0.

Syntax

MODE LPT*n*[:][*cpl*] [,[*lpi*][,P]]

where:

LPT*n*	is the parallel printer port (*n* = 1, 2, or 3).
cpl	is the number of characters per line (80 or 132, default = 80).
lpi	is the number of lines per inch (6 or 8, default = 6).
P	causes continuous retries when the printer is not ready.

Description

This form of the MODE command configures an IBM or compatible printer connected to parallel port *n*. Its effect on other printer types may vary. The command has the side effect of canceling any redirection that was previously applied to the specified port with a Redirect Printing MODE command.

The first parameter, LPT*n*, designates the parallel printer port to be configured (LPT1, LPT2, or LPT3). All the other parameters are optional.

The *cpl* parameter selects between printing 80 characters on a line (the default) and 132 characters on a line. The *lpi* parameter selects between 6 lines per inch (the default) and 8 lines per inch. (Note that the attached printer must be capable of printing 132 characters per line or 8 lines per inch and of understanding IBM-compatible printer-control codes; otherwise, specifying these values will have no effect.)

The last parameter in the command line, P, configures the system to retry output continuously (or until Ctrl-Break is pressed) if the printer is not ready or not on line (interpreted by the computer as a time-out error), rather than display an error message. (Note that if P is used and *lpi* is omitted, the comma preceding *lpi* must be specified.) Use of the P option causes part of the MODE program to become permanently resident in memory. (This option is not available in PC-DOS version 1.0.)

Examples

To configure the printer on the first parallel port to print 132 characters per line, with 8 lines per inch, type

```
C>MODE LPT1:132,8  <Enter>
```

To configure the system to continually send output to the printer on the second parallel port if a time-out error occurs but to leave the other values at their defaults, type

```
C>MODE LPT2:,,P  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

Infinite retry of parallel printer timeout
The P option was included in the command line and the system will continuously retry to send output to the printer attached to the specified port if it is not ready or not on line.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid parameters
The command line included an incorrect parallel-port specification or one of the configuration parameters was not correct.

LPT*n*: set for 80
The specified printer has been configured for 80 characters per line.

LPT*n*: set for 132
The specified printer has been configured for 132 characters per line.

Printer error
The configuration command could not be carried out because the printer is turned off, not ready, or not on line.

Printer lines per inch set
The printer has successfully been configured for the specified 6 or 8 lines per inch.

Resident portion of MODE loaded
The P option was specified in the command line and part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MODE

3.2

Set Display Mode

External

Purpose

Selects the active video adapter and its display mode or reconfigures the current display. This form of the MODE command is included with PC-DOS beginning with version 2.0.

Syntax

MODE *display*

or

MODE [*display*],*shift*[,T]

where:

display is a video adapter and display mode from the following list:

40	Color/graphics adapter, 40 characters per line
80	Color/graphics adapter, 80 characters per line
BW40	Color/graphics adapter, 40 characters per line, color disabled from composite output
BW80	Color/graphics adapter, 80 characters per line, color disabled from composite output
CO40	Color/graphics adapter, 40 characters per line, color enabled
CO80	Color/graphics adapter, 80 characters per line, color enabled
MONO	Monochrome adapter

shift is R or L, to shift the display left or right one (40-column display) or two (80-column display) character positions.

T causes a test pattern to be displayed for screen alignment.

Description

This form of the MODE command has two uses. The first is to select the active video adapter and its display mode (if more than one adapter is present in the system) or to reconfigure the current adapter. The second is to shift the screen display to the left or right to center it. In both cases, the screen is cleared as a side effect of the command.

The *display* parameter selects the active video adapter and mode or reconfigures the current adapter. If a display adapter that is not available is specified, MODE displays an error message.

The *shift* parameter is simply the single character R or L preceded by a comma. Each shift command causes the screen image to be shifted by two characters if the display adapter is in 80-column mode or by one character if it is in 40-column mode. When the T option is also included in the command line, the screen image is shifted, a test pattern is displayed, and the user is prompted to indicate whether the screen should be shifted again. Note that use of *shift* causes part of the MODE program to become permanently resident in memory.

Examples

In a system with both a color/graphics adapter and a monochrome display adapter, to select the monochrome display as the active display, type

```
C>MODE MONO  <Enter>
```

To select a color 80-column text mode on the color/graphics adapter, shift the screen image two characters to the left, and display a test pattern, type

```
C>MODE CO80,L,T  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Do you see the leftmost 0? (Y/N)

or

Do you see the rightmost 9? (Y/N)
When the *shift* and T options are used together, this message allows the user to shift the test-pattern display successive positions until it is properly centered.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid parameter
The specified display adapter or mode is not available.

Requested Screen Shift out of range
The display cannot be shifted any further.

Unable to shift Screen left
The screen has already been shifted as far left as possible or the active display adapter cannot be shifted (monochrome or enhanced graphics adapter).

Unable to shift Screen right
The screen has already been shifted as far right as possible or the active display adapter cannot be shifted (monochrome or enhanced graphics adapter).

MODE

3.2

Configure Serial Port

External

Purpose

Controls the configuration of the serial communications adapter. This form of the MODE command is included with PC-DOS beginning with version 1.1.

Syntax

MODE COM*n*[:]*baud*[,*parity*[,*databits*[,*stopbits*[,P]]]]

where:

COM*n*	is the serial port (*n* = 1 or 2).
baud	is the baud rate (110, 150, 300, 1200, 2400, 4800, or 9600).
parity	is the type of parity checking (N = none, O = odd, E = even, default = E).
databits	is the number of bits per character (7 or 8, default = 7).
stopbits	is the number of stop bits (1 or 2, default = 1, except with 110 baud where default = 2).
P	causes continuous retries when the output device is not ready.

Description

This form of the MODE command configures the specified serial port for communication with an external device such as a printer, a terminal, or a modem.

The first parameter, COM*n*, designates the serial port to be configured (COM1 or COM2). Except for the port number and the baud rate, which are required, a parameter can be left unchanged by entering a comma without a value in its position in the command line. (If *all* optional parameters are to be left unchanged and P is not used in the command line, no commas are required.)

The baud rate must be one of the values 110, 150, 300, 600, 1200, 2400, 4800, or 9600. The first two digits can be used as an abbreviation for the full value.

The *parity* parameter specifies the type of parity checking to be done on each character and must be one of the characters N, O, or E (for none, odd, or even, respectively); the default is even parity. The *databits* parameter specifies the length of a character and must be either 7 or 8; the default is 7. The *stopbits* parameter is either 1 or 2. If *baud* is set for 110, the default number of *stopbits* is 2; otherwise, the default is 1.

The last parameter in the command line, P, configures the system to retry output continuously (or until Ctrl-Break is pressed) if the device interfaced to the serial port is not ready or not on line, rather than display an error message. Use of the P option causes part of the MODE program to become permanently resident in memory.

Consult the user's manual for the specific printer, modem, terminal, or other device to determine the proper settings for the MODE parameters.

If a serial printer is to be used instead of LPT1 as the system's default printer, the Redirect Printing MODE command must be specified *after* the Configure Serial Port MODE command.

Example

To configure the first serial port for 9600 baud, no parity, 8 databits, and 1 stop bit, type

```
C>MODE COM1:9600,N,8,1  <Enter>
```

Messages

COM*n: baud,parity,databits,stopbits,timeout*
After the serial port is configured successfully, MODE displays an advisory message confirming the settings. If the P option was not used in the command line, a hyphen character (-) is displayed for *timeout*, to indicate no continuous retries if the printer is not ready or is not on line.

COM port does not exist
The serial port specified in the command line does not exist in the system.

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

Invalid baud rate specified
The baud rate included in the command line was not one of the allowed values or was abbreviated incorrectly.

Invalid parameters
The command line specified a COM port that does not exist in the system or one of the configuration parameters for the COM port was not valid.

No COM: ports
The computer does not have any serial ports installed.

Resident portion of MODE loaded
The P option was specified in the command line and part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MODE

3.2

Redirect Printing

External

Purpose

Redirects output from a parallel port to a serial communications port. This form of the MODE command is included with PC-DOS beginning with version 1.1.

Syntax

MODE LPT*n*[:][=COM*n*[:]]

where:

LPT*n*	is the parallel port to be redirected (*n* = 1, 2, or 3).
COM*n*	is the serial port (*n* = 1 or 2) to be used for output instead of LPT*n*.

Description

This form of the MODE command redirects any output for the specified parallel port, sending it to the specified serial communications port instead. The parallel port can be LPT1, LPT2, or LPT3; the serial port can be either COM1 or COM2. A Configure Serial Port MODE command is required *before* the Redirect Printing MODE command, to configure the serial port for the proper baud rate, parity, word length, and stop bits.

Redirection can be canceled by entering *MODE LPT*n alone.

Use of MODE to redirect printer output causes part of the MODE program to become permanently resident in memory. Canceling the redirection will not remove this resident portion from memory.

Example

To cause all output to the first parallel port (LPT1) to be redirected to the first serial port (COM1), type

```
C>MODE LPT1:=COM1:  <Enter>
```

Messages

DOS 2.0 or later required
MODE does not work with versions of MS-DOS earlier than 2.0.

Illegal device name
Either the parallel port or the serial port specified in the command line does not exist in the system.

Incorrect DOS version
The version of MODE is not compatible with the version of MS-DOS that is running.

INTERNAL ERROR in MODE application
An internal error occurred in the MODE utility and the requested reconfiguration was not carried out.

LPT*n*: not redirected
No serial port was specified and any previous redirection from the specified parallel port was canceled.

LPT*n*: redirected to COM*n*:
The MODE command has successfully redirected the output for the specified parallel port to the specified serial port.

Resident portion of MODE loaded
Part of the MODE command has become permanently resident in memory, decreasing slightly the amount of memory available to other programs.

MORE

2.0 and later

Display by Screenful

External

Purpose

Displays output one screenful at a time on standard output.

Syntax

MORE

Description

The MORE filter reads lines of text from standard input and sends them to standard output one screenful (23 lines) at a time. At the end of each screenful, MORE displays the message -- *More* -- and then waits for any key to be pressed before it continues. (Pressing Crtl-C or Ctrl-Break terminates the MORE filter.)

The default input device is the keyboard; the default output device is the video display. Because standard input can be redirected, the MORE filter can also accept input from another character device or a file or from the piped output of another program or filter. Similarly, the output of MORE can be redirected to any character device or file or can be piped to another program (however, the message -- *More* -- will be included with the redirected or piped output).

Examples

To display the file SHELL.C one screenful at a time, type

```
C>MORE < SHELL.C  <Enter>
```

To display the directory of \MASM\SOURCE in the current drive one screenful at a time, pipe the output of the DIR command to the MORE filter by typing

```
C>DIR \MASM\SOURCE ¦ MORE  <Enter>
```

Messages

-- More --
This informational message is displayed at the end of each screenful of text. Press any key to resume output.

MORE: Incorrect DOS version
The version of MORE is not compatible with the version of MS-DOS that is running.

PATH

2.0 and later

Define Command Search Path

Internal

Purpose

Specifies one or more additional drives and/or directories to be searched for a program or batch file if the file cannot be found in the current or specified drive and directory.

Syntax

PATH [*drive*:][*path*][;[*drive*:][*path*]...]

or

PATH ;

where:

drive	is the drive containing the disk to be searched for the executable file.
path	is the name of the directory to be searched for the executable file.

Description

When a command line is entered at the MS-DOS system prompt, the command processor first checks to see if the specified command is one of its internal commands. If it is not, the command processor searches the current directory of the current drive for a file with the same name and the extension .COM, .EXE, or .BAT, in that order. If found, the file is loaded into memory and executed (if the extension is .COM or .EXE) or interpreted by the resident batch-file processor (if the extension is .BAT); otherwise, MS-DOS displays the message *Bad command or file name,* followed by the system prompt. In versions 3.0 and later, a path can precede the command name, causing MS-DOS to make the initial search for a program or batch file under the specified path.

The PATH command designates one or more disk drives and/or directory paths to be searched sequentially for a program or batch file if the file cannot be found in the current or specified drive and directory. The drives and/or directory paths are searched in the order they appear in the PATH command. Multiple *drive:path* pairs can be specified, separated by semicolons. A copy of the PATH string is passed to each executing process as a part of the process's environment.

If the *drive* parameter is specified without an associated path, MS-DOS assumes the root directory of *drive*. If the PATH command is followed only by a semicolon, MS-DOS deletes the existing path. If the PATH command is entered with no parameters, MS-DOS displays the existing path.

Invalid or nonexistent drives and/or paths in the PATH command do not result in an error message but are ignored when the PATH string is inspected later during a search for a program or batch file.

The PATH command is generally placed in the AUTOEXEC.BAT file on the system disk so that the search order will be defined each time the system is turned on or restarted.

Examples

To define the directory \BIN on the disk in drive A as the directory to be searched for a program or batch file if the file is not found in the current or specified directory, type

```
C>PATH A:\BIN  <Enter>
```

Subsequent entry of the command

```
C>PATH  <Enter>
```

results in the display

```
PATH=A:\BIN
```

To define the root, \BIN, \DOS, and \DATA directories on drive C and the \UTIL directory on the disk in drive B as the locations to be searched for a program or batch file if the file is not found in the current or specified directory, type

```
C>PATH C:\;C:\BIN;C:\DOS;C:\DATA;B:\UTIL  <Enter>
```

To delete the current search path, type

```
C>PATH ;  <Enter>
```

Message

No Path
The PATH command was entered without parameters and no search path is currently in effect.

PRINT

Print Spooler

2.0 and later

External

Purpose

Loads and configures the background print spooler or adds or deletes files from the print spooler's queue.

Syntax

PRINT [/D:*device*] [/B:*n*] [/M:*n*] [/Q:*n*] [/S:*n*] [/U:*n*] [[*drive*:][*path*]*filename*] [/C][/P] [[[*drive*:][*path*]*filename*] [/C][/P]...]

or

PRINT /T

where:

filename is the name of the file to be added to or deleted from the print queue, optionally preceded by a drive (and a path with versions 3.0 and later); wildcard characters are permitted.

/B:*n* sets the print-buffer size in bytes (1–32767, default = 512) (versions 3.0 and later).

/C deletes the immediately preceding file and all subsequent files from the print queue (until a /P switch is encountered).

/D:*device* is the character device to be used for printing (default = PRN); must be the first switch, if used (versions 3.0 and later).

/M:*n* is the length of time in timer ticks that PRINT keeps control during each of its time slices (1–255, default = 2) (versions 3.0 and later).

/P adds the immediately preceding file and all subsequent files to the print queue (until a /C switch is encountered).

/Q:*n* is the maximum number of files allowed in the print queue (1–32, default = 10) (versions 3.0 and later).

/S:*n* is the number of time slices per second that PRINT gives control to the foreground process (1–255, default = 8) (versions 3.0 and later).

/T terminates printing and empties the print queue.

/U:*n* is the number of timer ticks that PRINT waits for a busy or unavailable printer or for a disk access or MS-DOS function call to terminate before giving up the time slice (1–255, default = 1) (versions 3.0 and later).

Description

The PRINT utility is a terminate-and-stay-resident (TSR) program that can print files from disk while other programs are running. PRINT maintains a first-in, first-out (FIFO) queue that can hold the names of as many as 32 files. PRINT does not attempt to interpret the contents of a file, except to expand tab characters (ASCII code 09H) with spaces to the next eight-column boundary and to interpret 1AH characters as end-of-file marks. (A program such as PRINT that can transfer files to a printer without any special knowledge of their contents or origin is called a print spooler.)

Note: The PRINT utility continues printing a file until it encounters an end-of-file character (1AH). Therefore, if PRINT is used with nontext files, it may encounter a 1AH character before reaching the end of the file and terminate printing before the entire file has been processed. In such cases, files should be printed using the COPY command, with PRN as the destination.

The PRINT program employs a technique called time-slicing, which is based on its use of the timer-tick interrupt and its detailed knowledge of MS-DOS. PRINT uses this interrupt, which occurs 18.2 times per second on IBM PC-compatible machines, to divide the processor's time between an application or utility program (such as a word processor or a spreadsheet) and the print spooler. Because the application program typically controls the display screen and the keyboard and receives most of the CPU time, it is called the foreground program. The print spooler, which receives a lesser part of the CPU time and usually operates without indicating its status or progress to the operator, is called the background program.

The /B:*n*, /D:*device*, /Q:*n*, /M:*n*, /S:*n*, and /U:*n* switches configure the PRINT utility. These switches are used only the first time the PRINT command is entered after the system has been turned on or restarted.

The /D:*device* switch, which must be the first switch in the command line if used, specifies the peripheral device the print spooler is to use for output. This can be any legal character-output device that is present in the system. If /D:*device* is not included in the first PRINT command, PRINT prompts the user to select an output device (default = PRN). Once an output device has been assigned, a new device cannot be selected without restarting the system.

The /B:*n* switch sets the size of PRINT's file buffer, which controls the amount of data that is read from a file at one time for printing. The value of *n* must be between 1 and 32767 bytes (default value = 512). Large file buffers reduce the amount of extra disk activity caused by the print spooler, but they also reduce the amount of memory available for use by other programs. The /Q:*n* switch controls the size of PRINT's queue — that is, the number of files that can be held in the buffer pending printing. The queue can be configured to hold 1 to 32 files (default = 10).

The /S:*n*, /M:*n*, and /U:*n* switches, available only with versions 3.0 and later, control the time-slicing behavior of PRINT. The /S:*n* switch sets the number of time slices per second — that is, how many times per second — PRINT will be given control; *n* is in the range 1 through 255 (default = 8). The /M:*n* switch sets the length of time (in timer ticks) that PRINT will keep control during each of its time slices; *n* is in the range 1 through 255 (default = 2). The /U:*n* switch specifies how long (in timer ticks) PRINT should wait for a busy or unavailable printer or for a disk access or MS-DOS function call to terminate before giving up its time slice; again, *n* is in the range 1 through 255 (default = 1). Unless there are special circumstances, the default values for these switches will give acceptable performance.

Files are added to the print queue by entering PRINT followed by one or more pathnames. Files are printed in the order they are placed in the queue. At the end of each file, the print spooler advances the paper to the top of the next page. If a filename containing wildcards is used, all matching files are added to the queue in the order in which they appear in the directory. After a file is queued for printing, it should not be renamed or erased, nor should the disk containing the file be removed, until the printing is complete.

Note: Each print queue entry can be a maximum of 63 characters, including the drive and path.

The /P and /C switches allow files to be added to and deleted from the print queue in the same command line. The /P switch (the default) adds to the print queue the immediately preceding file in the command line and all subsequent files until a /C switch is encountered. Conversely, the /C switch cancels printing for the immediately preceding file in the command line and for all subsequent files until a /P switch is encountered. If a canceled file is currently being printed, PRINT prints the message *File* filename *canceled by operator* on the listing, sounds the printer's alarm (if it has one), and advances the paper to the top of the next page.

The /T switch terminates printing by deleting all files from the print queue. If a file is currently being printed, PRINT prints the message *All files canceled by operator* on the listing, sounds the printer's alarm (if it has one), and advances the paper to the top of the next page.

If PRINT encounters a disk error while attempting to print a particular file, it cancels that file, prints an error message on the printer, sounds the printer's alarm (if it has one), advances the paper to the top of the next page, and goes to the next file in the print queue.

If the PRINT command is entered with no parameters, the contents of the print queue are displayed.

Because PRINT is a TSR utility, it reduces the amount of memory available for use by other programs. The only way to recover the memory occupied by PRINT, even after printing is complete, is to restart the system.

Examples

To install and configure the PRINT program and specify the auxiliary device (AUX) as the printing device, with a print queue that can hold as many as 32 filenames and with a buffer size of 2048 bytes, type

```
C>PRINT /D:AUX /Q:32 /B:2048  <Enter>
```

To add the file DOC.TXT in the current directory of the current drive to the print spooler's queue, type

```
C>PRINT DOC.TXT  <Enter>
```

To delete the file READY.TXT from the print queue and simultaneously add the files FINAL.TXT and REPORT.TXT to the queue, type

```
C>PRINT READY.TXT /C FINAL.TXT /P REPORT.TXT  <Enter>
```

To cancel the file being printed and remove all pending files from the print queue, type

```
C>PRINT /T  <Enter>
```

Messages

***filename* File not found**
A disk was changed or the file was renamed or erased after the PRINT command was entered but before the file was actually printed.

***filename* File not in print queue**
A command line with a /C switch specified a file that is not in the print queue.

***filename* is currently being printed**
This informational message shows which file PRINT is currently printing.

***filename* is in queue**
This informational message shows which file is in the queue waiting to be printed.

***filename* Pathname too long**
The pathname of a file to be printed exceeded 63 characters.

Access denied
An attempt was made to print a locked file.

All files canceled by operator
The /T switch was included in the command line. PRINT terminates printing of the current file, empties the print queue, sounds the printer alarm (if it has one), and advances the paper to the top of the next page.

Cannot use PRINT - Use NET PRINT
If network support has been installed, the NET PRINT command must be used to print files.

**Errors on list device indicate that it
may be off-line. Please check it.**
The printer has been turned off or placed off line while files are still in the print queue.

File *filename* canceled by operator
A PRINT command was entered with the /C switch to cancel a specific file. If the specified file is currently being printed, PRINT terminates printing of the file, sounds the printer alarm (if it has one), advances the paper to the top of the next page, and resumes printing with the next file in the queue.

Incorrect DOS version
The version of PRINT is not compatible with the version of MS-DOS that is running.

Invalid drive specification
A drive letter specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line included an invalid switch or configuration switches were used after the first time the PRINT command was used.

List output is not assigned to a device
An invalid destination device was previously entered. Restart the system and specify a valid device in the PRINT command.

Name of list device [PRN]:
This message is displayed in response to the first PRINT command line if the /D:*device* switch was not included. Specify any valid character-output device (default = PRN).

No paper error writing device *device*
An out-of-paper device error was detected while printing on the specified device.

PRINT queue is empty
No files are waiting to be printed.

PRINT queue is full
No additional files can be added to the print queue until the current file is printed. To increase the size of the print queue, restart the system and use the /Q:*n* switch in the PRINT command.

Resident part of PRINT installed
This informational message is displayed on the first entry of a PRINT command to indicate that the PRINT utility is now resident in memory. The amount of memory available to application programs is reduced accordingly.

PROMPT

2.0 and later

Define System Prompt

Internal

Purpose

Defines the form of the command processor's prompt. This command is included in PC-DOS beginning with version 2.1.

Syntax

PROMPT [*string*]

where:

string is a combination of ordinary printable characters and the following special display codes:

Code	Meaning
$b	¦ character
$d	Current date (in the form *Day mm-dd-yyyy*)
$e	Escape character (1BH)
$g	> character
$h	Backspace character (erases the previous character)
$l	< character
$n	Current drive
$p	Current drive and path
$q	= character
$t	Current time (in the form *hh:mm:ss.hh*)
$v	MS-DOS version number
$_	Carriage return/linefeed pair (starts a new line)
$$	$ character

Description

The system's default command processor, COMMAND.COM, displays a prompt on the screen whenever it is ready to accept a command from the user. The command processor determines the format of the prompt from the PROMPT environment variable, if it exists. Otherwise, it uses the default format, which in most OEM implementations of MS-DOS is the letter of the current drive followed by a greater-than sign (for example, C>).

The PROMPT command allows the user to customize the system prompt. This command is usually included in the AUTOEXEC.BAT file so that MS-DOS displays the custom prompt when the system is turned on or restarted.

The *string* parameter can be any combination of printable characters and the special $ control codes listed in the preceding table. The special $ codes allow certain variable information, such as the date and time, to be obtained from the operating system and displayed as part of the prompt. Such system information can be edited in the prompt with the backspace function, which is invoked with the code $h.

Note: When the time is displayed as part of a prompt, it is updated only when the command processor redisplays the prompt.

The escape character, invoked with the code $e, can be used to include standard ANSI escape sequences in *string* to control the appearance of text or its position on the screen. *See* USER COMMANDS: ANSI.SYS for further information on the ANSI escape sequences and the ANSI device driver.

If PROMPT is entered with no parameters, the system prompt is reset to the default format.

The PROMPT command works by modifying the PROMPT environment variable. The same result can be obtained using the SET command with *PROMPT=*string as its argument. *See* USER COMMANDS: SET for further discussion of the environment block and environment variables.

Examples

To define the system prompt as the word *Command* followed by a colon, type

```
C>PROMPT Command:  <Enter>
```

On fixed-disk-based systems it is desirable to display the current drive and path as part of the prompt. To define such a prompt followed by a > character, type

```
C>PROMPT $p$g  <Enter>
```

To define the system prompt to display the time, date, and current drive and path followed by a > character, each on a separate line, type

```
C>PROMPT $t$_$d$_$p$g  <Enter>
```

The system will respond with a display in the following form:

```
16:07:31.56
Thu  6-18-1987
C:\BIN\DOS>
```

To create a prompt that displays the time without the seconds and hundredths of a second, followed by a space and the date without the year, followed by a space and the current drive and a > character, type

```
C>PROMPT $t$h$h$h$h$h$h $d$h$h$h$h$h $n$g  <Enter>
```

The system will respond with

```
16:07 Thu  6-18 C>
```

To define a prompt that always displays the current time and date in the upper right corner of the screen before displaying the current drive and the > character on the current line, type

```
C>PROMPT $e[s$e[0;60H$t$h$h$h$h$h$h $d$e[u$n$g  <Enter>
```

The escape sequence *$e[s* saves the current cursor position; the sequence *$e[0;60H* positions the cursor at row 0, column 60; the next several codes format the date and time; the sequence *$e[u* restores the original cursor position. (This example requires that the ANSI driver be loaded to interpret the escape sequences.)

RAMDRIVE.SYS

3.2

Virtual Disk

External

Purpose

Creates a virtual disk in memory.

Syntax

DEVICE=[*drive*:][*path*]RAMDRIVE.SYS [*size*] [*sector*] [*directory*] [/A ¦ /E]

where:

size	is the size of the virtual disk in kilobytes (minimum = 16, default = 64).
sector	is the sector size in bytes (128, 256, 512, or 1024; default = 128).
directory	is the maximum number of entries in the virtual disk's root directory (3–1024, default = 64).
/A	causes RAMDRIVE to use Lotus/Intel/Microsoft Expanded Memory for storage (cannot be used with /E).
/E	causes RAMDRIVE to use extended memory for storage (cannot be used with /A).

Note: Unless a /A or /E switch is used, the virtual disk is created in conventional memory.

Description

The RAMDRIVE.SYS installable device driver allows the configuration of one or more virtual disks (sometimes referred to as electronic disks or RAMdisks). A virtual disk is implemented by mapping a disk's structure—directory, file allocation table, and files area—onto an area of random-access memory, rather than onto actual sectors located on a magnetic recording medium. Access to files stored on a virtual disk is very fast, because no moving parts are involved and the "disk" operates at the speed of the system's memory.

Warning: Because a RAMdisk resides entirely in RAM and is therefore volatile, any information stored there is irretrievably lost when the computer loses power or is restarted.

RAMDRIVE.SYS can create a virtual disk in conventional memory, extended memory, or Lotus/Intel/Microsoft Expanded Memory. Conventional memory is the term for the up-to-640 KB of RAM that contain MS-DOS and any application programs. Extended memory is the term for the memory at addresses above 1 MB (100000H) that is available on 80286-based personal computers such as the IBM PC/AT. Expanded memory is the term for a subsystem of bank-switched memory boards (and a driver to manage them) that is compatible with the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS).

A virtual disk can be installed in conventional memory by simply inserting the line *DEVICE=RAMDRIVE.SYS* into the system's CONFIG.SYS file and restarting the system. A new "drive" then becomes available in the system, with a default size of 64 KB, 128-byte sectors, and 64 available directory entries (assuming memory is sufficient). The virtual disk is assigned the next available drive letter (which is displayed in RAMDRIVE's sign-on message). The drive letter assigned depends on the number of other physical and virtual disks in the system and also on the position of the *DEVICE=RAMDRIVE.SYS* line in the CONFIG.SYS file relative to other installed block devices. Available memory permitting, multiple virtual disks can be created by using multiple *DEVICE=RAMDRIVE.SYS* lines. Several optional parameters allow the user to customize the size and configuration of the virtual disk and to use extended memory or expanded memory if it is available.

The *size* parameter specifies the amount of RAM, in kilobytes, to be allocated to the virtual disk. The default is 64 KB, but any size from 16 KB to the total amount of available memory can be specified.

The *sector* parameter sets the virtual sector size used within the virtual disk. The *sector* value can be 128, 256, 512, or 1024 bytes (default = 128 bytes). Selection of the smallest sector size results in a minimum of wasted virtual disk space per file but also results in a somewhat slower transfer of data. Physical disk devices on IBM PC-compatible systems always use 512-byte sectors.

Warning: The 1024-byte sector size is not supported in most implementations of MS-DOS and will terminate the installation of RAMDRIVE.SYS if it is used. Check the documentation included with the computer to see if this value is supported.

The *directory* parameter sets the number of available entries in the virtual disk's root directory. The allowed range is 3 to 1024 (default = 64). Each directory entry requires 32 bytes. RAMDRIVE rounds the number of available directory entries up, if necessary, so that an integral number of sectors are assigned to the root directory.

The /A switch causes Lotus/Intel/Microsoft Expanded Memory to be used for the virtual disk, rather than conventional memory; the /E switch causes extended memory to be used. Either option allows very large virtual disks to be configured while still leaving the maximum amount of conventional memory available for use by application programs. The /A and /E switches cannot be used together.

Note: If RAMDRIVE uses conventional memory for virtual disk storage, the memory cannot be reclaimed except by modifying the CONFIG.SYS file and restarting the system.

Examples

To create a virtual disk drive with the default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries, include the following command

```
DEVICE=RAMDRIVE.SYS
```

in the CONFIG.SYS file and restart the system.

To create a 4 MB virtual disk drive in Lotus/Intel/Microsoft Expanded Memory, with 512-byte sectors and 224 available directory entries, when RAMDRIVE.SYS is located in a directory named \DRIVERS on drive C, include the command

```
DEVICE=C:\DRIVERS\RAMDRIVE.SYS 4096 512 224 /A
```

in the CONFIG.SYS file and restart the system.

Messages

Microsoft RAMDrive version *n.nn* virtual disk *X*:
Disk size: *nn*k
Sector size: *nnn* bytes
Allocation unit: *n* sectors
Directory entries: *nnn*
RAMDRIVE.SYS was successfully installed and this message informs the user of the version of RAMDRIVE.SYS that created the virtual disk, the drive letter assigned to the disk, and the characteristics of the disk.

RAMDrive: Above Board Memory Manager not present
The /A switch was used in the command line and the Lotus/Intel/Microsoft Expanded Memory Manager is not present in the system. Place the DEVICE command that loads the memory manager *before* the *DEVICE=RAMDRIVE.SYS* command in the CONFIG.SYS file.

RAMDrive: Above Board Memory Status shows errors
The Above Board device driver is bad or damaged or the board itself is defective. Consult the Above Board manual or the manufacturer.

RAMDrive: Computer must be PC-AT, or PC-AT compatible.
The /E switch was used in the command line and the computer is not an 80286-based IBM PC/AT or compatible.

RAMDrive: Incorrect DOS version
The version of RAMDRIVE.SYS is not compatible with the version of MS-DOS that is running.

RAMDrive: Insufficient memory
Available memory is insufficient for RAMDRIVE.SYS to create a virtual drive.

RAMDrive: Invalid parameter
One of the parameters supplied in the command line is incorrect or is not supported by the computer.

RAMDrive: I/O error accessing drive memory
The Expanded Memory Manager device driver is bad or damaged or the board itself is defective. Consult the board's manual or contact the manufacturer.

RAMDrive: No extended memory available
The /E switch was specified but the system does not contain extended memory.

RECOVER

Recover Files

2.0 and later

External No Net

Purpose

Reconstructs files from a disk that has developed unreadable sectors or has a damaged directory.

Syntax

RECOVER *drive*:

or

RECOVER [*drive*:][*path*]*filename*

where:

drive is the letter of the drive holding the disk with a damaged directory.

filename is the name of the file that will be reconstructed, optionally preceded by a drive and/or path; wildcard characters are not permitted.

Description

The RECOVER command partially rescues a file on a disk that has developed bad sectors by deleting the bad sectors from the file. RECOVER can also reconstruct files (including files stored in subdirectories) from a disk that has a damaged directory.

When RECOVER is used with a filename, the file is read allocation unit by allocation unit; unreadable allocation units are marked as bad and are no longer allocated to the file. The resulting file is usable, although the data contained in the bad allocation units is lost. (The recovered file may or may not be reusable by the specific application that created it.) The directory entry for *filename* is also adjusted to reflect the sectors that were lost and the bad sectors are marked in the disk's file allocation table so that they are not reused for another file.

If a disk's directory is damaged, it still may be possible to recover all the files on the disk and build a new directory by using RECOVER with *drive* as the only command-line parameter. RECOVER completely erases the previous contents of the damaged directory and constructs new directory entries for each of the original files by inspecting the disk's file allocation table. The recovered files receive names of the form FILE*nnnn*.REC, starting with FILE0001.REC. Each recovered file's size is always a multiple of the disk cluster size, so recovered files may require editing to eliminate spurious data at the ends of the files.

RECOVER restores each subdirectory as an individual file that contains the names of the files originally stored in it. The actual files contained within those subdirectories are also reconstructed, although they are no longer associated with the subdirectory in which they originally resided. Restored files and subdirectories, regardless of their location on the damaged disk, are placed in the new root directory. If there are more files on the damaged disk than can be contained in the new root directory (for example, more than 112 for a 5.25-inch, 360 KB floppy disk), the user must repeat the RECOVER command after copying the already-recovered files to another disk and deleting them from the damaged disk.

Examples

To recover the file MENUMGR.C in the current directory of the current drive, type

```
C>RECOVER MENUMGR.C  <Enter>
```

To recover all files on the disk in drive B, which has a damaged directory, type

```
C>RECOVER B:  <Enter>
```

Messages

***n* file(s) recovered**
When RECOVER is used on a disk with a damaged directory, this informational message is displayed at the conclusion of processing to indicate how many files of the form FILE*nnnn*.REC were constructed.

***n* of *n* bytes recovered**
When RECOVER is used on a damaged file, this informational message is displayed at the conclusion of processing to advise how many bytes of the file were recovered.

Cannot RECOVER a Network drive
Files on a drive assigned to a network cannot be recovered.

File not found
The file specified in the command line cannot be found or does not exist.

Incorrect DOS version
The version of RECOVER is not compatible with the version of MS-DOS that is running.

Invalid drive or file name
An invalid drive letter was specified or the filename contains a wildcard.

Invalid number of parameters
More than one drive letter or filename was specified in the command line.

Press any key to begin recovery of the
file(s) on drive *X*
This prompt message gives the user the opportunity to change disks after the RECOVER program is loaded but before processing begins.

Warning - directory full
New directory entries for the reconstructed files cannot be created because the root directory is full. Copy the recovered files to another disk, delete them from the damaged disk, and then repeat the RECOVER command on the damaged disk.

RENAME or REN

1.0 and later

Change Filename

Internal

Purpose

Changes the name of a file or set of files.

Syntax

RENAME [*drive*:][*path*]*oldname newname*

or

REN [*drive*:][*path*]*oldname newname*

where:

oldname is the name of an existing file or set of files, optionally preceded by a drive and/or path; wildcard characters are permitted.

newname is the new name to be assigned to *oldname*; wildcard characters are permitted, but a drive and/or path cannot be specified.

Description

The RENAME command changes the name of an existing file or set of files. It does not make copies of files or move files from one location in the disk's directory structure to another or from one drive to another.

The *oldname* parameter can refer to a single file or can include wildcards to specify a set of files; a drive and path can be included as part of *oldname*.

The *newname* parameter specifies the new name to be given to the file or files; it cannot include a drive or path. A wildcard in *newname* causes that portion of the original filename to be left unchanged. If the new name for a file is the same as the name of an existing file, RENAME terminates with an error message.

Examples

To rename the file REVS.DOC, located in the current directory of the current drive, to CHANGES.TXT, type

```
C>RENAME REVS.DOC CHANGES.TXT  <Enter>
```

or

```
C>REN REVS.DOC CHANGES.TXT  <Enter>
```

To rename all files with a .DOC extension in the \SOURCE directory on the disk in drive D to have a .TXT extension, type

```
C>REN D:\SOURCE\*.DOC *.TXT  <Enter>
```

Messages

Duplicate file name or File not found
The new name specified for a file already exists or a file with the old name cannot be found or does not exist.

Invalid directory
The command line included a reference to a directory that is invalid or does not exist.

Invalid drive specification
The command line included a reference to a disk drive that is invalid or does not exist in the system.

Invalid number of parameters
The command line included too few or too many filenames.

Invalid parameter
The *newname* parameter in the command line included a drive and/or path.

REPLACE

3.2

Update Files

External

Purpose

Selectively adds or replaces files on a disk.

Syntax

REPLACE [*drive*:]*pathname* [*drive*:][*path*] [/A][/D][/P][/R][/S][/W]

where:

pathname	is the name and location of the source files to be transferred, optionally preceded by a drive; wildcard characters are permitted in the filename.
drive:path	is the destination for the file being transferred; filenames are not permitted in the destination parameter.
/A	transfers only those source files that do not exist at the destination (cannot be used with /S or /D).
/D	transfers only those source files with a more recent date than their destination counterparts (cannot be used with /A).
/P	prompts the user for confirmation before each file is transferred.
/R	allows REPLACE to overwrite destination read-only files.
/S	searches all subdirectories of the destination directory for a match with the source files (cannot be used with /A).
/W	causes REPLACE to wait for the disk to be changed before transferring files.

Description

The REPLACE utility allows files to be updated easily to more recent versions. REPLACE examines the source and destination directories and, depending on the switches used in the command line, selectively updates matching files or copies only those files that exist on the source disk but not the destination disk.

The *pathname* parameter (the source) specifies the name and location of the files to be transferred (optionally preceded by a drive); wildcards are permitted in the filename. The *drive:path* parameter (the destination) specifies the location of the files to be replaced and can consist of a drive, a path, or both. If only a drive is specified as the destination, REPLACE assumes the current directory of the disk in that drive. If the destination is omitted completely, REPLACE assumes the current drive and directory. The /S switch causes REPLACE to also search all subdirectories of the destination directory for files to be replaced.

The /A, /D, and /P switches allow selective replacement of files on the destination disk. When the /A switch is used, REPLACE transfers only those files on the source disk that do not exist in the destination directory. When the /D switch is used, REPLACE transfers only those source files that match the destination filenames but have a more recent date than their destination counterparts. (The /D switch is not available with the PC-DOS version of REPLACE.) The /P switch causes REPLACE to prompt the user for confirmation before each file is transferred.

The /R switch allows the replacement of read-only as well as normal files. If the /R switch is not used and one of the destination files that would otherwise be replaced is marked read-only, the REPLACE program terminates with an error message. (REPLACE cannot be used to update hidden or system files.)

The /W switch causes REPLACE to pause and wait for the user to press any key before beginning the transfer of files. This allows the user to change disks in floppy-disk systems with no fixed disk and in those cases where the REPLACE program itself is present on neither the source nor the destination disk.

Return Codes

0	The REPLACE operation was successful.
1	An error was found in the REPLACE command line.
2	No matching files were found to replace.
3	The source or destination path was invalid or does not exist.
5	One of the files to be replaced was marked read-only and the /R switch was not included in the command line.
8	Memory was insufficient to run the REPLACE command.
15	An invalid drive was specified in the command line.
Other	Standard MS-DOS error codes (returned on a failed Interrupt 21H file-function request).

Examples

To replace the files in the directory \SOURCE on the current drive with all matching files on the disk in drive A that have a more recent date, type

```
C>REPLACE A:*.* \SOURCE /D  <Enter>
```

To transfer from the disk in drive A only those files that are not already present in the current directory, type

```
C>REPLACE A:*.* /A  <Enter>
```

Messages

***n* File(s) added**
After the replacement operation is completed, if the /A switch was used in the command line, REPLACE displays the total number of files added.

***n* File(s) replaced**
After the replacement operation is completed, REPLACE displays the total number of files processed.

Access denied *'pathname'*
One of the files to be replaced on the destination disk is marked read-only and the /R switch was not included in the command line.

Add *pathname*? (Y/N)
The /A and /P switches were specified in the command line and REPLACE prompts the user for confirmation before adding each file.

Adding *pathname*
The /A switch was specified in the command line and REPLACE displays the name of each file it adds.

File cannot be copied onto itself *'pathname'*
The source and destination command-line parameters specified the same file in the same location.

Incorrect DOS Version
The version of REPLACE is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The destination disk does not have enough available space to hold the files being added or replaced.

Insufficient memory
The system does not have enough RAM available to process the REPLACE command.

Invalid drive specification 'X:'
The command line specified a disk drive that is invalid or does not exist in the system.

Invalid parameter *'switch'*
The command line included a switch that is not supported by the REPLACE command.

No files added
The /A switch was used and the specified file(s) already exist on the destination disk.

No files found *'pathname'*
The files to be added or replaced on the destination disk were not found on the source disk.

No files replaced
The files at the destination are identical with the files on the source disk or do not meet the criteria specified by the switches.

Parameters not compatible
The command line included two or more switches that cannot be used together.

Path not Found *'pathname'*
The source or destination parameter included a nonexistent path or directory.

Path too long
The source or destination parameter included a path element that is too large (probably because of a missing backslash character [\]).

Press any key to begin adding file(s)
The /W and /A switches were specified in the command line and REPLACE waits for the user to press a key before proceeding, allowing disks to be changed.

Press any key to begin replacing file(s)
The /W switch was specified in the command line and REPLACE waits for the user to press a key before proceeding, allowing disks to be changed.

Replace *pathname*? (Y/N)
The /P switch was specified in the command line and REPLACE prompts the user for confirmation before replacing the file.

Replacing *pathname*
This informational message indicates the progress of the REPLACE command by displaying the name of each file as it is being replaced.

Source path required
Although the destination parameter can usually be omitted and defaults to the current drive and directory, the source location for the files to be replaced must always be specified.

Unexpected DOS Error *n*
This message usually indicates a bad or damaged disk. Use the CHKDSK command to determine the problem.

RESTORE

2.0 and later

Restore Backup Files

External

Purpose

Restores files from a disk created with the BACKUP command.

Syntax

RESTORE *drive1*: [*drive2*:][*pathname*] [/A:*date*] [/B:*date*] [/E:*time*] [/L:*time*][/M][/N] [/S][/P]

where:

drive1 is the drive that contains the backup files created by the BACKUP command.

drive2 is the drive to which the backup files will be restored.

pathname is the name of the file(s) to be restored from *drive1*; wildcard characters are permitted in the filename. If a path is used, a filename must be specified.

/A:*date* restores files that were modified on or after *date.*

/B:*date* restores files that were modified on or before *date.*

/E:*time* restores files modified at or before *time.*

/L:*time* restores files modified at or after *time.*

/M restores only files modified since the last backup.

/N restores only files that no longer exist on the destination disk.

/P prompts the user for confirmation before restoring hidden or read-only files or before overwriting files that have changed since they were last backed up .

/S restores all files in the subdirectories of the specified directory, in addition to the files in the specified directory.

Note: The PC-DOS version of RESTORE supports only the /P and /S switches.

Description

The RESTORE command restores files from a backup disk or directory created with the BACKUP command to their original location in a directory structure. Before version 3.1, the RESTORE command could restore files only from one floppy disk to another or from a floppy disk to a fixed disk. With later versions, RESTORE can also restore files from one fixed disk to another or from a fixed disk to a floppy disk.

The *drive1* parameter specifies the source for the backed-up files. If the source disk is a fixed disk, the backup files are always obtained from the directory \BACKUP. If multiple floppy disks were used to hold the backed-up files, RESTORE prompts the user for each disk as it is required.

The destination can be any combination of a drive, a path, and a filename; the filename can include wildcards. If the destination drive is omitted, MS-DOS assumes the current drive. If a path is not specified, the files are restored to the current directory. (Note that files must be restored to the same directory they were backed up from.) If a path is specified, a filename must be specified as well. If neither a path nor a filename is included in the command line, all directories, subdirectories, and files on the backup disk(s) are restored to the destination disk. The /S switch can be used to force restoration of the files in all the subdirectories of a named directory.

Files are restored in the order they were backed up, regardless of their current order on the destination disk. If files with the same name and location already exist on the destination disk, they are replaced by the backup copies.

The RESTORE program supports a number of switches that allow selective restoration of files from the backup disk. The /A:*date*, /B:*date*, /E:*time*, and /L:*time* switches allow files to be restored based on the time and/or date they were backed up. The /M switch restores only those files that have been changed on the destination disk since the backup disk was created. The /P switch prompts the user before restoring a hidden or read-only file or a file that has been changed since it was last backed up.

The MS-DOS and PC-DOS RESTORE programs are compatible except when a /A:*date*, /B:*date*, /E:*time*, /L:*time*, /M, or /N switch is used. These switches are not supported in the PC-DOS version.

Warning: The RESTORE command should not be used on a disk drive affected by an ASSIGN, SUBST, or JOIN command.

Return Codes

0 The restore operation was successful.
1 No files were found to restore.
2 Some files were not restored because of a file-sharing conflict (versions 3.0 and later).
3 The restore operation was terminated by the user.
4 The program was terminated by an unrecoverable (critical) hardware error.

Examples

To restore the file named MENUMGR.C from the backup disk in drive A to the directory named \SOURCE on the disk in drive B, type

```
C>RESTORE A: B:\SOURCE\MENUMGR.C  <Enter>
```

To restore all the files on the backup disk in drive A to their original locations in the directory structure of drive C, type

```
C>RESTORE A: C:\*.* /S  <Enter>
```

To restore all the files with the extension .C from the backup disk in drive A to the directory named \SOURCE on drive C, requesting confirmation for those files that are read-only or hidden, type

```
C>RESTORE A: C:\SOURCE\*.C /P  <Enter>
```

Messages

***** Files were backed up at *time* on *date* *****
This informational message shows when the BACKUP command was used on the backed-up files.

***** Not able to restore file *****
The backup file or the destination disk contains an error. Use the CHKDSK command to determine the problem.

***** Restoring files from drive *X*: *****
Diskette: *n*
This informational message indicates the progress of the RESTORE command.

DOS 2.0 or later required
RESTORE does not work with versions of MS-DOS earlier than 2.0.

File creation error
The destination directory is full. This usually occurs only if the destination is the root directory but can also happen if a file is being restored to a subdirectory and the disk itself is full.

Incorrect DOS version
The version of RESTORE is not compatible with the version of MS-DOS that is running.

Insert backup diskette *n* in drive *X*:
Strike any key when ready
This message prompts the user to insert the next backup disk in sequence. Disks used in multidisk backups should always be labeled and numbered during a BACKUP operation.

Insert restore target diskette in drive *X*:
Strike any key when ready
This prompt is displayed when files are being restored to a floppy disk.

Insufficient memory
Available memory is not sufficient for the RESTORE program to execute.

Invalid drive specification
The command line included a drive that is invalid or does not exist in the system.

Invalid number of parameters
The command line included too many or too few parameters.

Invalid parameter
The command line included an invalid switch or other parameter.

Invalid path
The destination parameter included a path that is invalid or does not exist.

Restore file sequence error
Files are being restored from a multidisk set of backup disks and a floppy disk was used out of order.

Source and target drives are the same
Files cannot be restored from a drive to the same drive.

Source does not contain backup files
The files on the backup disk are not in the special format used by the BACKUP and RESTORE programs.

System files restored
Target disk may not be bootable
The backup disk included copies of the hidden operating-system files MSDOS.SYS and IO.SYS (or IBMDOS.COM and IBMBIO.COM in PC-DOS) and these files were restored to the destination disk. The destination disk is bootable only if these two files are the first files on the disk and IO.SYS (or IBMBIO.COM) is written into contiguous clusters.

Target is full
The destination disk is full and no further files can be restored.

Target is Non-Removable
The disk to which files are being restored is not removable.

The last file was not restored
The destination disk is full or the last file on the backup disk was bad.

Warning! Diskette is out of sequence
Replace diskette or continue if okay
Files are being restored from a multidisk set of backup disks and a floppy disk was used out of order.

Warning! File *filename*
is a hidden file
Replace the file (Y/N)?
The backed-up file has the same filename as a hidden file on the destination disk, which may be overwritten. (This message appears only if the /P switch was used.) Respond with *Y* to overwrite the file on the destination disk; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! File *filename*
is a read-only file
Replace the file (Y/N)?
The backed-up file has the same name as a read-only file on the destination disk, which may be overwritten. (This message appears only if the /P switch was used.) Respond with

Y to overwrite the file on the destination disk; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! File *filename*
was changed after it was backed up
Replace the file (Y/N)?
Data has been changed or added to the destination file since the backup disk was created and this data will be lost if the file is restored. (This message appears only if the /P switch was used.) Respond with *Y* to restore the backed-up file; respond with *N* to leave the destination file unchanged and continue the RESTORE operation.

Warning! No files were found to restore
No files were found on the backup disk that matched the destination file specification.

RMDIR or RD

2.0 and later

Remove Directory

Internal

Purpose

Removes an empty directory from the hierarchical file structure.

Syntax

RMDIR [*drive*:][*path*]*directory_name*

or

RD [*drive*:][*path*]*directory_name*

where:

directory_name	is the name of the directory to be removed, optionally preceded by a drive and/or path.

Description

The RMDIR command removes an empty directory from a disk's hierarchical file structure. The directory being deleted cannot contain any files or subdirectories (except for the special **.** and **..** entries). The root directory or current directory of a disk cannot be deleted.

If the *path* parameter is used, it must specify a valid existing path. If no path is specified and *directory_name* is not preceded by a backslash (\), MS-DOS assumes that the directory to be removed is a subdirectory of the current directory. If no path is specified and *directory_name* is preceded by a backslash, MS-DOS assumes that the directory is a subdirectory of the root directory. The length of the full path (including the drive designator and directory name) must not exceed 63 characters.

The RMDIR command should not be used to remove subdirectories from drives affected by an ASSIGN or JOIN command. A directory affected by the SUBST command cannot be removed.

Note: If a directory contains files marked as hidden or system, that directory cannot be removed even though no files appear to exist when the directory contents are viewed using the DIR command.

Example

To remove the empty directory \LIB, which is a subdirectory of the \MSC directory on the disk in drive A, type

```
C>RMDIR A:\MSC\LIB  <Enter>
```

or

```
C>RD A:\MSC\LIB  <Enter>
```

Message

Invalid path, not directory, or directory not empty
The named directory cannot be deleted because it does not exist, some element of the path to the directory does not exist, or the directory contains files or subdirectories.

SELECT

IBM

Configure System Disk for a Specific Country

External

Purpose

Creates a system disk with time, date, and keyboard configured for a selected country. This command is available only with PC-DOS.

Syntax

SELECT [[*drive1*:] *drive2*:[*path*]] *country keyboard*

where:

drive1 is a floppy-disk drive (A or B) containing the distribution disk or, at a minimum, the PC-DOS system files, COMMAND.COM, and the FORMAT and XCOPY utilities (default = drive A) (version 3.2).

drive2 is the drive containing the disk to receive the PC-DOS system files and country information and can include a path (default = drive B) (version 3.2).

country is a code from the table below that controls the time, date, and currency formats.

keyboard is a code from the table below that controls the keyboard configuration.

Country	Country Code	Keyboard Code
Australia	061	*
Belgium	032	*
Canadian French	002	*
Denmark	045	*
Finland	358	*
France	033	FR
West Germany	049	GR
Israel	972	*
Italy	039	IT
Middle East	785	*
Netherlands	031	*
Norway	047	*
Portugal	351	*
Spain	034	SP
Sweden	046	*
Switzerland	041	*

(more)

Country	Country Code	Keyboard Code
United Kingdom	044	UK
United States	001	US

*Available only in version 3.2 and may be supplied on a separate floppy disk.

Description

The SELECT utility allows the user to create a bootable system disk configured for a particular country's keyboard layout and date, time, and currency formats without performing these steps separately.

Version 3.2 of SELECT uses the FORMAT command to format the disk in *drive2*, then uses the XCOPY command to copy all files on the disk in *drive1* (including the hidden system files) to *drive2*. If a country configuration other than one of the six KEYB*xx* utilities supplied on the distribution disk is specified, SELECT prompts the user to insert the disk containing the appropriate file.

Versions 3.0 and 3.1 of SELECT use the DISKCOPY program to copy all files on the disk in drive A (including the hidden system files) to the disk in drive B, formatting the disk if necessary.

All versions then add the appropriate CONFIG.SYS and AUTOEXEC.BAT files to the new disk to configure PC-DOS for use with the specified keyboard and country configuration. The specified configuration does not take effect until the computer is turned on or restarted using the new disk.

Examples

To create a PC-DOS system disk configured for West Germany using version 3.0 or 3.1, place a copy of the original PC-DOS distribution disk in drive A and a blank disk in drive B; then type

```
A>SELECT 049 GR  <Enter>
```

During the copy operation, the usual DISKCOPY prompts and messages are displayed. When the copy operation is complete, the two disks are compared using DISKCOMP, producing the usual DISKCOMP prompts and messages. The resulting disk includes all the files from the distribution disk (including the hidden system files), a CONFIG.SYS file that contains the line

```
COUNTRY=049
```

and an AUTOEXEC.BAT file that contains the following lines:

```
KEYBGR
ECHO OFF
CLS
DATE
TIME
VER
```

To create a PC-DOS system disk configured for West Germany using version 3.2, place a copy of the original PC-DOS distribution disk in drive A and a blank disk in drive B; then type

```
A>SELECT 049 GR  <Enter>
```

SELECT first uses the FORMAT command to format the disk in drive B, then uses XCOPY to copy all files on the distribution disk (including the system files), and finally creates a CONFIG.SYS file that contains the line

```
COUNTRY=049
```

and an AUTOEXEC.BAT file that contains the following lines:

```
PATH \;
KEYBGR
ECHO OFF
CLS
DATE
TIME
VER
```

Messages

Cannot execute *X*: *filename*
One of the files needed by SELECT (FORMAT, DISKCOPY, DISKCOMP, or XCOPY) is not on the source disk or is a version that is not compatible with the version of PC-DOS that is running.

File creation error
The root directory of the destination disk is full or unable to contain any more files or one of the files being created has the same name as a directory already on the destination disk.

Incorrect DOS version
The version of SELECT is not compatible with the version of PC-DOS that is running (version 3.2).

Incorrect number of parameters
Too many or too few parameters were specified in the command line or a separator character was omitted between two parameters (version 3.2).

Insert DOS diskette in drive A:
Strike any key when ready
This message prompts the user to insert the distribution disk containing the system files and COMMAND.COM into drive A (version 3.2).

Insert KEYB*xx*.COM diskette in drive *X*:
Strike any key when ready
The user responded *Y* to a previous prompt asking if KEYB*xx* is on another disk. This message prompts the user to insert that disk into the specified drive (version 3.2).

Insert target diskette in drive A:
Strike any key when ready
This message prompts the user to insert the disk that will become the country-specific system disk into drive A (versions 3.0 and 3.1).

Insert target diskette in drive B:
Strike any key when ready
This message prompts the user to insert the disk that will become the country-specific system disk into drive B (version 3.2).

Invalid country code
The country code given in the command line is not supported by this version of PC-DOS or is not a valid country code.

Invalid drive specification
One of the drives specified in the command line is invalid or does not exist in the system (version 3.2).

Invalid keyboard code
The keyboard code given in the command line is not supported by this version of PC-DOS or is not a valid keyboard code.

Invalid parameter
One of the parameters specified in the command line is invalid or is not supported by the version of SELECT that is running (version 3.2).

Invalid path
The path specified for *drive2* is invalid, contains invalid characters, or is longer than 63 characters (version 3.2).

Is KEYB*xx*.COM on another
diskette (Y/N)?
The keyboard reconfiguration file for the specified country is not on the source disk. Respond with *Y* to cause SELECT to prompt for the disk containing the keyboard file after the FORMAT operation is completed; respond with *N* to terminate the SELECT command (version 3.2).

Keyboard routine not found.
The user responded *N* to a previous prompt asking if KEYB*xx* is on another disk (version 3.2).

SELECT is used to install DOS the first
time. Select erases everything on the
specified target and then installs DOS.
Do you want to continue (Y/N)?
This message warns the user that the specified disk will be formatted and all files on the source disk will be copied over. Respond with *Y* to continue; respond with *N* to terminate the SELECT command (version 3.2).

Unable to copy keyboard routine
An error occurred while the KEYB*xx*.COM program was being copied. Use the CHKDSK command to check the keyboard program on the source disk for damage (version 3.2).

Unable to create directory
The directory specified in the command line was not created because a directory with the same name already exists on the destination disk, the root directory of the destination disk is full, one of the directory names specified in the path does not exist, or a file with the same name already exists (version 3.2).

SET

2.0 and later

Set Environment Variable

Internal

Purpose

Defines an environment variable and a string that is its value.

Syntax

SET [*name*=*value*]

or

SET *name*=

where:

name	is a string of characters that defines an environment variable; lowercase letters are automatically converted to uppercase.
value	is a string of characters, a pathname, or a filename that defines the current value of *name*; no case conversion is made for *value*.

Description

The environment is a series of null-terminated ASCII (ASCIIZ) strings that contains environment variables and their values. (An environment variable associates a string consisting of a filename, a pathname, or other literal data with a symbolic name that can be referenced by programs. The form of the association is *name*=*value*.) The original, or master, environment belongs to the command processor and is established when the system is turned on or restarted. When a program is subsequently executed by the command processor or by another program, the new program inherits a private copy of its parent's environment.

The SET command enables the user to add, change, or delete an environment variable from the command processor's environment. If *value* is not included in the SET command, MS-DOS deletes the environment variable *name* from the environment. If the SET command is issued with no parameters, MS-DOS displays the values of all the variables in the environment.

With MS-DOS versions 2.x and 3.x, two particular variables are always found in an environment: PATH and COMSPEC. These variables are initialized during the system startup process and tell COMMAND.COM which subdirectories to search for executable files and where to find the transient portion of COMMAND.COM for reloading (versions 3.0 and later). (By default, PATH is a null string and therefore searches only the current or specified directory.) These special environment variables are influenced by the PATH and SHELL commands, respectively, but can also be changed with SET commands. Note, however, that changing the value of COMSPEC with SET will serve no useful purpose—changing to a different command processor must be done using an appropriate SHELL command in the CONFIG.SYS file (the system must be restarted for it to take effect). Note also that it is not necessary to use the SET command with the PATH or PROMPT commands—MS-DOS will automatically add their new values to the environment if they are changed.

The environment, which can be as large as 32 KB, can be an effective source of global configuration information to executing programs. For instance, the Microsoft C Compiler and Microsoft Object Linker use environment variables to locate *include* and object library files. Environment variables can also be referenced as replaceable parameters in batch files, using the form %*name*%.

Under normal circumstances, MS-DOS expands the environment as necessary when SET commands are entered. However, when a batch file is being interpreted or when terminate-and-stay-resident (TSR) utilities have been loaded, the size of the command processor's environment becomes fixed. Under these circumstances, a SET command may result in the error message *Out of environment space.*

With version 3.2, the initial size of the environment can be increased either by using the COMMAND command with the /P and /E:*nnnn* switches at the system prompt or by including a SHELL command specifying COMMAND.COM followed by the /E:*nnnn* switch in the CONFIG.SYS file. *See* USER COMMANDS: COMMAND; CONFIG.SYS: SHELL.

Examples

To define the environment variable *USER* and set its value to *FRED*, type

```
C>SET USER=FRED  <Enter>
```

To change the value of the environment variable *USER* to *SALLY*, type

```
C>SET USER=SALLY  <Enter>
```

To delete the environment variable *USER* and its value from the environment, type

```
C>SET USER=  <Enter>
```

To display all the environment variables, type

```
C>SET  <Enter>
```

The output of this command will be in the following form:

```
COMSPEC=C:\DOS3\COMMAND.COM
PROMPT=$p$_$n$g
PATH=D:\BIN;C:\DOS3;C:\WP\WORD;C:\ASM;C:\MSC\BIN
INCLUDE=c:\msc\include;c:\windows\lib
LIB=c:\msc\lib;c:\windows\lib
TMP=c:\temp
PCF32=c:\forth\pc32
PROCOMM=c:\procomm\
```

Message

Out of environment space
The command processor's environment is full and cannot be expanded (usually because the SET command was issued from a batch file or the system has terminate-and-stay-resident [TSR] utilities installed).

SHARE

3.0 and later

Install File-Sharing Support

External

Purpose

Loads the resident file-sharing support module required by Microsoft Networks.

Syntax

SHARE [/F:*n*] [/L:*n*]

where:

/F:*n*	allocates *n* bytes of memory to hold file-sharing information (default = 2048).
/L:*n*	configures support for *n* simultaneous file-region locks (default = 20).

Description

The code that supports file sharing and locking in a networking environment is isolated in the user-installable SHARE module. After SHARE is loaded, MS-DOS checks all read and write requests against the file-sharing module. On personal computers that do not utilize network services, the SHARE module need not be loaded, leaving more memory for application programs.

The /F:*n* switch controls the amount of buffer space allocated for file-sharing information. Each open file requires the length of its full name, including the path, plus some overhead; the average pathname is approximately 20 bytes long. If the /F:*n* switch is not included in the command line, the buffer size defaults to 2048 bytes (sufficient for approximately 100 files with pathnames of average length).

The /L:*n* switch controls the number of entries to be allocated for an internal table containing file-locking information. Each active lock on a region of a file occupies one entry in the table. If the /L:*n* switch is absent, the default is support for 20 simultaneously active locks.

Example

To install the file-sharing support module, allocating 4096 bytes of space for file-sharing information and 40 file-region locks, type

```
C>SHARE /F:4096 /L:40  <Enter>
```

Messages

Incorrect DOS version
The version of SHARE is not compatible with the version of MS-DOS that is running.

Incorrect parameter
The command line included an invalid switch.

Not enough memory
System memory is insufficient to load the SHARE module or to reserve the designated file-sharing information space or file-region locks.

SHARE already installed
The SHARE command has already been executed since the system was turned on or restarted; additional executions have no effect.

SORT

2.0 and later

Alphabetic Sort Filter

External

Purpose

Reads records from standard input, sorts them alphabetically, and writes the sorted records to standard output.

Syntax

SORT [/R][/+*column*]

where:

/R	specifies a reverse, or descending, alphabetic sort.
/+*column*	specifies the first column to be used for sorting each line (default = 1).

Description

The SORT program is a filter that reads lines from standard input until an end-of-file marker is reached, sorts the lines into alphabetic order, and writes the sorted lines to standard output.

Standard input defaults to the keyboard; standard output defaults to the video display. Because standard input can be redirected, the SORT filter can also accept input from another character device, a file, or the piped output of another program or filter. (The most common use of SORT is to sort the redirected input from an ASCII text file.) Similarly, the output of SORT can be redirected to any character device or file or can be piped to another program.

SORT normally orders the lines of the input text stream alphabetically using the entire line, starting with column 1 as the sort key. Tab characters are not expanded to spaces. If the character in the sort-key column of one line is identical with the character in the sort-key column of the next line, SORT checks the next column to the right to determine which line will go before the other. If the second columns are also identical, the search continues to the right until a differing column is found. The maximum amount of data that can be sorted is 63 KB.

The /R switch causes SORT to arrange the set of lines in reverse alphabetic order. The /+*column* switch lets the user specify a column other than column 1 as the first sort key.

With versions 2.x, SORT arranges the input lines based on the ASCII value of the character in each line's sort-key column; the sort operation is therefore case sensitive. With versions 3.0 and later, SORT assigns lowercase letters the same ASCII value as uppercase letters; hence, case is effectively ignored. Depending on the COUNTRY command in effect (*see* USER COMMANDS: CONFIG.SYS: COUNTRY), versions 3.0 and later map accented characters with ASCII codes in the range 80H through 0E1H (128–225) to their unaccented equivalents for sorting.

Warning: If the output of the SORT command is redirected to a file with the same name as the input file, the contents of the input file may be destroyed.

Examples

The examples in this entry operate on an ASCII text file named RECORDS.TXT that contains the following lines:

```
Smith    Seattle
Adams    New York
Zoole    Bellevue
Jones    Boston
```

Each line of the file contains a person's surname, starting in column 1, and a city name, starting in column 10.

To sort the file RECORDS.TXT by surname and display the sorted lines on standard output, type

```
C>SORT < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Adams    New York
Jones    Boston
Smith    Seattle
Zoole    Bellevue
```

To sort the file RECORDS.TXT by surname and write the sorted lines into the file READY.DOC, type

```
C>SORT < RECORDS.TXT > READY.DOC  <Enter>
```

To sort the file RECORDS.TXT by surname in reverse alphabetic order and display the sorted lines on standard output, type

```
C>SORT /R < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Zoole    Bellevue
Smith    Seattle
Jones    Boston
Adams    New York
```

To sort the file RECORDS.TXT by city name and display the sorted lines on standard output, type

```
C>SORT /+10 < RECORDS.TXT  <Enter>
```

This will result in the following display:

```
Zoole    Bellevue
Jones    Boston
Adams    New York
Smith    Seattle
```

To use SORT as a filter to arrange a directory listing alphabetically, type

```
C>DIR ¦ SORT  <Enter>
```

To use SORT as a filter to arrange a directory listing alphabetically based on the first character of each file's extension, type

```
C>DIR ¦ SORT /+10  <Enter>
```

Messages

Invalid parameter
One of the parameters specified in the command line is invalid or the syntax is incorrect.

SORT: Incorrect DOS version
The version of SORT is not compatible with the version of MS-DOS that is running.

SORT: Insufficient disk space
The output of the SORT filter has been redirected to a file and the disk is full.

SORT: Insufficient memory
The available system memory is insufficient to run the SORT program.

SUBST

3.1 and later

Substitute Drive for Subdirectory

External No Net

Purpose

Causes a drive letter to be substituted for a directory name. SUBST is present in MS-DOS to support older application programs that do not accept pathnames.

Syntax

SUBST [*drive1*: [*drive2*:]*path*]

or

SUBST *drive1*: /D

where:

drive1	is the drive letter to be used to reference the files in *path*.
drive2	is a drive letter other than *drive1* that can optionally precede the name of the subdirectory being substituted.
path	is the subdirectory to be accessed when *drive1* is referenced, optionally preceded by *drive2*.
/D	cancels the effect of a previous SUBST command for *drive1*.

Description

The SUBST command allows a drive letter to be substituted for a subdirectory name.

The *drive1* parameter can be any valid drive letter except the current drive or *drive2*. Drive letters A through E are always available; drive letters beyond E require that an appropriate LASTDRIVE command be added to the CONFIG.SYS file and the system be restarted (*see* USER COMMANDS: CONFIG.SYS: LASTDRIVE).

After a SUBST command, the files on the disk normally referenced by *drive1* are no longer accessible. However, the files in the location specified by *path* can still be referenced by the usual methods (using their actual drive and path) as well as by the substituted drive designator.

If the SUBST command is entered without parameters, MS-DOS displays the substitutions currently in effect.

Warning: The SUBST command masks the actual disk-drive characteristics from commands that perform critical disk operations. Therefore, ASSIGN, BACKUP, CHKDSK, DISKCOMP, DISKCOPY, FDISK, FORMAT, JOIN, LABEL, and RESTORE should not be used on a drive affected by a SUBST command. CHDIR, MKDIR, RMDIR, and PATH commands that include the affected drive should be used with caution. A network drive cannot be named in a SUBST command.

Examples

To substitute drive B for the directory C:\ASM\SOURCE, type

```
C>SUBST B: C:\ASM\SOURCE  <Enter>
```

To display the substitutions currently in effect, type

```
C>SUBST  <Enter>
```

In this case, the SUBST command displays

```
B: => C:\ASM\SOURCE
```

To cancel the effect of a previous SUBST command that substituted drive B for a subdirectory, type

```
C>SUBST B: /D  <Enter>
```

Messages

Cannot SUBST a network drive
One or both of the drive parameters in the command line referred to a drive that is assigned to a network.

DOS 2.0 or later required
SUBST does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of SUBST is not compatible with the version of MS-DOS that is running.

Incorrect number of parameters
The command line included too many or too few parameters.

Invalid parameter
The drive named in the command line is invalid, does not exist, is the default drive, or is the same as the drive in the path to be substituted.

Not enough memory
The available system memory is insufficient to run the SUBST command.

Path not found
An element of the path included in the command line is invalid or does not exist.

SYS

1.0 and later

Transfer System Files

External No Net

Purpose

Copies the hidden files that contain the operating system from the disk in the current drive to another formatted disk.

Syntax

SYS *drive*:

where:

drive	is the location of the disk that will receive the system files. This parameter is required.

Description

An MS-DOS system disk must contain three files to be bootable: the two operating-system files and the command processor. The operating system itself is contained in the files IO.SYS and MSDOS.SYS (or IBMBIO.COM and IBMDOS.COM in PC-DOS), which must always be the first two files in the disk's directory. Both have file attributes set for system and hidden (all versions) and read-only (versions 2.0 and later). IO.SYS (or IBMBIO.COM) contains the default set of device drivers for the system; it must occupy contiguous sectors in the disk's files area. MSDOS.SYS (or IBMDOS.COM) contains the kernel of the operating system proper. The third required file is the shell, or command processor, which by default is COMMAND.COM. This is an unrestricted file and can be located anywhere on the disk.

The SYS command transfers the two operating-system files from the default drive to the specified destination disk. The destination disk that receives the files must meet one of the following requirements:

- The disk is formatted but completely empty.
- The disk currently contains hidden MS-DOS system files that are large enough to allow replacement by the new system files.
- The disk has been formatted with the /B switch to reserve room for the system files. (Note that /B produces a disk with only eight sectors per track.)

If the disk already contains the two hidden system files, the SYS command can be used to transfer an equivalent or later version of MS-DOS.

After the two hidden operating-system files are installed with the SYS command, the COMMAND.COM file (or another command processor) must be transferred to the destination disk with the COPY command. The resulting disk is a bootable system disk.

Note: Because the two system files have the hidden attribute, they do not appear on a directory listing produced by the DIR command. The CHKDSK command does report the presence of hidden files on a disk and will list their names if the /V switch is used but will not list such information as the file size or date and time of creation.

Example

To transfer a copy of the system files to the disk in drive B, type

```
C>SYS B:  <Enter>
```

Messages

Cannot SYS to a Network drive
The drive specified in the command line is currently assigned to a network.

Destination disk cannot be booted
The hidden operating-system files were transferred to the destination disk but could not be placed in contiguous sectors.

Incompatible system size
The destination disk already contains operating-system files and they are smaller than those being copied.

Incorrect DOS version
The version of SYS is not compatible with the version of MS-DOS that is running.

Insert destination disk in drive *X*
and strike any key when ready
This message prompts the user to insert the disk onto which the operating-system files will be copied into the specified drive.

Insert system disk in drive *X*
and strike any key when ready
This message prompts the user to insert a disk containing the operating-system files into the specified drive.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line contained an invalid drive letter.

No room for system on destination disk
Contiguous space at the beginning of the destination disk is insufficient for the operating-system files. This can occur when files already exist on the destination disk or when sections of the disk are marked as unusable by the FORMAT command.

No system on default drive
The disk in the default drive does not contain the two hidden system files. Replace the disk with a bootable system disk.

System transferred
The operating-system files have been successfully transferred to the destination disk.

TIME

Set System Time

1.0 and later

Internal

Purpose

Sets or displays the system time. TIME is an external command with PC-DOS version 1.0.

Syntax

TIME [*hh*:*mm*[:*ss*[.*xx*]]]

where:

hh	is hours (0-23).
mm	is minutes (0-59).
ss	is seconds (0-59).
xx	is hundredths of a second (0–99).

Note: No spaces are allowed between any of the time parameters.

Description

All computers that run MS-DOS have as part of their hardware configuration a timer, or clock, that maintains the current system date and time. One use of this clock, among others, is to insert the current date and time into a file's directory entry when the file is created or modified.

The TIME command allows the user to display or modify the current time that is being maintained by the system's real-time clock. TIME is also executed by MS-DOS when the system is turned on or restarted, unless an AUTOEXEC.BAT file is on the system disk, in which case the command is executed only if it is included in the AUTOEXEC.BAT file.

On IBM PC/ATs and compatibles, the TIME command does not permanently change the system time stored in the built-in battery-backed clock/calendar; the newly entered time is lost when the system is turned off or restarted. On these machines, the SETUP program (found on the *Diagnostics for IBM Personal Computer AT* disk or equivalent) must be used to permanently alter the clock/calendar's current time.

On IBM PCs, PC/XTs, and compatibles equipped with add-on cards containing battery-backed clock/calendar circuitry, it is usually necessary to run a time/date installation program (included with the card) to set the system date and time from the clock/calendar on the card. The TIME command generally has no effect on these card-mounted clock/calendars.

The format of times displayed by the system depends on the current country code, which is determined by the optional COUNTRY command in the CONFIG.SYS file (*see* USER COMMANDS: CONFIG.SYS: COUNTRY). The default display format is the 24-hour format (00:00–23:59).

Examples

To display the current time, type

```
C>TIME  <Enter>
```

This results in output of the following form:

```
Current time is 12:49:04.93
Enter new time:
```

To leave the time unchanged, press the Enter key.

To set the system time to 8:30 P.M., type

```
C>TIME 20:30  <Enter>
```

Messages

Current time is *hh:mm:ss.xx*
This informational message is displayed in response to any valid TIME command.

Invalid parameter
The delimiter in the time parameter included in the command line was not a colon (:) or a period (.).

Invalid time
Enter new time:
An invalid time, time format, or delimiter was specified in the command line or in response to the *Enter new time:* prompt. Note that no spaces are allowed around delimiters.

TREE

3.2

Display Directory Structure

External

Purpose

Displays the hierarchical directory structure of a disk and, optionally, the names of the files in each subdirectory. This command is included with PC-DOS beginning with version 2.0.

Syntax

TREE [*drive*:][/F]

where:

drive is the location of the disk whose directory structure is to be displayed.
/F displays the filenames in each directory in addition to the directory names.

Description

The TREE command displays on standard output the pathname of each directory on the disk in the specified drive, beginning with the subdirectories of the root directory. If a disk drive is not designated, TREE assumes the current, or default, drive. The name of each directory is followed by a list of its subdirectories. If the /F switch is included in the command line, the names of the files in each subdirectory are also displayed. (Prior to version 3.1, the PC-DOS TREE command does not list the files in the root directory if /F is used.)

The output of the TREE command can be redirected to another output device or a file or can be piped to another program.

Examples

Assume that the root directory of the disk in drive B contains three subdirectories: \SOURCE, \LIBS, and \DOC. The subdirectory \SOURCE in turn contains two subdirectories: \ASM and \PASCAL. To display the directory structure of this disk, type

```
C>TREE B:  <Enter>
```

The TREE command displays the following list:

```
DIRECTORY PATH LISTING FOR VOLUME MYDISK

Path: B:\SOURCE

Sub-directories:  ASM
                  PASCAL


Path: B:\SOURCE\ASM

Sub-directories:  None


Path: B:\SOURCE\PASCAL

Sub-directories:  None


Path: B:\LIBS

Sub-directories:  None


Path: B:\DOC

Sub-directories:  None
```

To display the directory structure of the disk in drive B and also display all files in each directory, type

```
C>TREE B: /F  <Enter>
```

To print the directory-structure listing of the disk in drive B on an attached printer, type

```
C>TREE B: > PRN  <Enter>
```

To display the directory structure of the disk in drive B one screenful at a time, type

```
C>TREE B: ¦ MORE  <Enter>
```

For a more compressed listing of all subdirectories on the disk in drive B, type

```
C>TREE B: ¦ FIND "Path:"  <Enter>
```

The output appears in the following form:

```
Path: B:\SOURCE
Path: B:\SOURCE\ASM
Path: B:\SOURCE\PASCAL
Path: B:\LIBS
Path: B:\DOC
```

Messages

DOS 2.0 or later required
TREE does not work with versions of MS-DOS earlier than 2.0.

Incorrect DOS version
The version of TREE is not compatible with the version of MS-DOS that is running.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid parameter
The command line contained a path or filename in addition to a disk drive or contained an invalid switch.

No sub-directories exist
The specified drive has no subdirectories.

TYPE

1.0 and later

Display File

Internal

Purpose

Sends the contents of an ASCII text file to standard output.

Syntax

TYPE [*drive*:][*path*]*filename*

where:

filename is the name of the text file to be displayed, optionally preceded by a drive and/or path; wildcard characters are not permitted.

Description

The TYPE command displays the contents of a text file on standard output (usually the video display) until it encounters an end-of-file character (ASCII code 1AH). Tab characters in the file are expanded to spaces with tab stops at each eighth character position. If a file contains characters with ASCII values less than 32 or greater than 127, the resulting display includes graphics characters and other unintelligible information.

The output of the TYPE command can be redirected to another file or character device or can be piped to another program.

Examples

To display the file SHELL.C in the directory \SOURCE on the disk in drive A, type

```
C>TYPE A:\SOURCE\SHELL.C  <Enter>
```

To direct the output of the same file to the printer, type

```
C>TYPE A:\SOURCE\SHELL.C > PRN  <Enter>
```

The TYPE command can be used with the MORE filter to paginate output. For example, to display the contents of the file MENU.ASM one screenful at a time, type

```
C>TYPE MENU.ASM ¦ MORE  <Enter>
```

Messages

File not found
The file specified in the command line cannot be found or does not exist.

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Invalid path or file name
The path specified in the command line is invalid or does not exist.

VDISK.SYS

IBM

Virtual Disk

External

Purpose

Creates a virtual disk in memory. This installable driver is available only with PC-DOS.

Syntax

DEVICE=[*drive*:][*path*]VDISK.SYS [*size*] [*sector*] [*directory*] [/E] (version 3.0)

or

DEVICE=[*drive*:][*path*]VDISK.SYS [*size*] [*sector*] [*directory*] [/E[:*max*]] (version 3.1)

or

DEVICE=[*drive*:][*path*]VDISK.SYS [*comment*] [*size*] [*comment*] [*sector*] [*comment*] [*directory*] [/E[:*max*]] (version 3.2)

where:

comment is a string of ASCII characters in the range 32 through 126, excluding the slash character (/) (version 3.2).
size is the size of the virtual disk in kilobytes (minimum = 1, default = 64).
sector is the sector size in bytes (128, 256, or 512; default = 128).
directory is the maximum number of entries in the virtual disk's root directory (2–512, default = 64).
/E causes VDISK to use extended memory.
/E:*max* causes VDISK to use extended memory and sets the maximum number of sectors (1–8, default = 8) to transfer from extended memory at one time (versions 3.1 and later).

Note: Unless the /E switch is used, the virtual disk is created in conventional memory.

Description

The VDISK.SYS installable device driver allows the configuration of one or more virtual disks (sometimes referred to as electronic disks or RAMdisks). A virtual disk is implemented by mapping a disk's structure — directory, file allocation table, and files area — onto an area of random-access memory, rather than onto actual sectors located on a magnetic recording medium. Access to files stored in a virtual disk is very fast, because no moving parts are involved and the "disk" operates at the speed of the system's memory. (The VDISK driver is available only with PC-DOS; a similar program named RAMDRIVE.SYS is included with MS-DOS.)

Warning: Because a RAMdisk resides entirely in RAM and is therefore volatile, any information stored there is irretrievably lost when the computer loses power or is restarted.

VDISK can create a virtual disk in either conventional memory or extended memory. Conventional memory is the term for the up-to-640 KB of RAM that contain PC-DOS and any application programs. Extended memory is the term for the memory at addresses above 1 MB (100000H) that is available on 80286-based personal computers such as the IBM PC/AT.

A virtual disk can be installed in conventional memory by simply inserting the line *DEVICE=VDISK.SYS* into the system's CONFIG.SYS file and restarting the system. (If the file VDISK.SYS is not in the root directory of the startup disk, it may be preceded by a drive and/or path.) A new "drive" then becomes available in the system, with default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries (assuming there is sufficient memory). The virtual disk is assigned the next available drive letter (which is displayed in VDISK's sign-on message). The drive letter assigned depends on the number of other physical and virtual disks in the system and also on the position of the *DEVICE=VDISK.SYS* line in the CONFIG.SYS file relative to other installed block devices. Available memory permitting, multiple virtual disks can be created by using multiple *DEVICE=VDISK.SYS* lines. Several optional parameters allow the user to customize the size and configuration of the virtual disk and to use extended memory if it is available.

The *size* parameter specifies the amount of RAM, in kilobytes, to be allocated to the virtual disk. The default is 64 KB, but any size from 1 KB to the total amount of available memory can be specified. If the size specified is greater than available memory or less than 1 KB, VDISK ignores it and creates a virtual disk of 64 KB. If necessary, VDISK also adjusts the *size* value to ensure that at least 64 KB of memory remain available in the system.

The *sector* parameter sets the virtual sector size used within the virtual disk. The *sector* value may be 128, 256, or 512 bytes (default = 128 bytes). Selection of the smallest sector size results in a minimum of wasted virtual disk space per file but also results in somewhat slower transfer of data.

Note: Physical disk devices in IBM PC-compatible systems always use 512-byte sectors.

The *directory* parameter sets the number of available entries in the virtual disk's root directory. The allowed range is 2 through 512 (default = 64). Each directory entry requires 32 bytes. VDISK rounds the number of available directory entries up, if necessary, so that an integral number of sectors are assigned to the root directory.

The /E switch causes VDISK to use extended memory for the virtual disk, rather than conventional memory. This allows very large virtual disks to be configured while still leaving the maximum amount of conventional memory available for use by application programs. If the /E switch is used and extended memory is not present in the system, the VDISK driver will not install itself.

When /E is used in the form /E:*max*, the variable *max* controls how many virtual sectors can be transferred at a time from extended memory. The value of *max* must be in the range 1 through 8 (default = 8). If VDISK operation appears to conflict with the communications port or other interrupt-driven peripheral devices, the *max* variable should be set to a smaller number. The *max* option is available only with versions 3.1 and 3.2.

Note: If VDISK uses conventional memory for virtual disk storage, the memory cannot be reclaimed except by modifying the CONFIG.SYS file and restarting the system.

Examples

To create a virtual disk drive with the default values of 64 KB disk size, 128-byte sectors, and 64 available directory entries, include the command

```
DEVICE=VDISK.SYS
```

in the CONFIG.SYS file and restart the system.

To create a 360 KB virtual disk with 512-byte sectors and 112 available directory entries when the file VDISK.SYS is located in a directory named \BIN on drive C, include the command

```
DEVICE=C:\BIN\VDISK.SYS 360 512 112
```

in the CONFIG.SYS file and restart the system. The directory for this virtual disk requires 3584 bytes (112 entries * 32 bytes), or 7 sectors.

With version 3.2, comments can be inserted between the values to identify them. For example, to create a 1 MB virtual disk drive in extended memory with 256-byte sectors and 128 directory entries, placing comments before the values to identify them, include the command

```
DEVICE=VDISK.SYS DISK_SIZE: 1024 SECTOR_SIZE: 256 DIR_ENTRIES: 128 /E
```

in the CONFIG.SYS file and restart the system.

Messages

Buffer size adjusted
No *size* value was specified or the specified value was larger than the amount of available memory.

Directory entries adjusted
No *directory* value was specified, VDISK adjusted the *directory* value up to the nearest sector-size boundary, or the *size* value was too small to hold the file allocation table, the directory, and two additional sectors, in which case VDISK adjusted *directory* downward until these conditions were met.

Invalid switch character
A slash character (/) was included in a comment or the /E switch was entered incorrectly.

Sector size adjusted
The *sector* value was missing from the command line or an incorrect value was entered; therefore, VDISK used the default value of 128 bytes.

Transfer size adjusted
A value outside the range 1 through 8 was specified with the /E:*max* switch; therefore, VDISK used the default value of 8.

VDISK not installed - Extender Card switches do not match the system memory size
The switch settings on the extender card are not correct or the extended memory exists in an expansion unit, which VDISK is not capable of using.

VDISK not installed - insufficient memory
Less than 64 KB of system memory remained after attempted installation, the /E switch was specified and the system does not contain extended memory, or the amount of available extended memory was too small to support the installation of VDISK.

VDISK Version *n.nn* virtual disk *X*:
Buffer size: *nn* KB
Sector size: *nnn*
Directory size: *nnn*
Transfer size: *n*
VDISK was successfully installed and this message informs the user of the drive letter assigned to the virtual disk, the version of VDISK that created the disk, and the characteristics of the disk. The *Transfer size:* message appears only in versions 3.1 and 3.2 and only if the /E switch was used.

VER

2.0 and later

Display Version

Internal

Purpose

Displays the MS-DOS version number.

Syntax

VER

Description

The VER command displays on standard output (usually the video display) the number of the MS-DOS version that is running. The version number is also displayed as part of the copyright notice when the system is turned on or restarted, unless an AUTOEXEC.BAT file is on the system disk. (The VER command can be included in the AUTOEXEC.BAT file to display the version number, but it will not display the copyright information.)

Examples

To display the MS-DOS version number, type

```
C>VER  <Enter>
```

On a system that is running MS-DOS version 3.2, the following message is displayed:

```
MS-DOS Version 3.2
```

To print the MS-DOS version number on an attached printer instead of displaying it on the screen, type

```
C>VER > PRN  <Enter>
```

VERIFY

2.0 and later

Set Verify Flag

Internal

Purpose

Sets the system's internal flag controlling verification of disk writes.

Syntax

VERIFY [ON¦OFF]

Description

The VERIFY command sets or clears an internal MS-DOS flag that controls verification of data written to disks. (The actual verification process is usually carried out by the device driver and the disk-drive controller.) The VERIFY ON command has the same effect on a global basis as the /V switch has on COPY operations. (When VERIFY is on, use of the /V switch with COPY has no additional effect.) VERIFY ON remains in effect until a program turns it off with a Set Verify system call or until the user types *VERIFY OFF* at the command prompt. The VERIFY command does not affect the operation of character devices.

When the VERIFY command is entered without an ON or OFF, MS-DOS displays the current state of the system's internal verify flag. The default setting of the verify flag is off.

Examples

To turn on verification of disk writes, type

```
C>VERIFY ON  <Enter>
```

To display the current status of the verify flag, type

```
C>VERIFY  <Enter>
```

Messages

Must specify ON or OFF
The command line contained an invalid parameter.

VERIFY is off

or

VERIFY is on
No setting was specified in the command line and VERIFY displays this informational message indicating the current status of the verify flag.

VOL

2.0 and later

Display Disk Name

Internal

Purpose

Displays a disk's volume label if one exists.

Syntax

VOL [*drive*:]

where:

drive is the location of the disk whose volume label is to be displayed.

Description

The VOL command displays a disk's name, or volume label. If *drive* is not included in the command line, the volume label of the disk in the current drive is displayed.

A volume label can be assigned to a disk when it is formatted by using the /V switch with the FORMAT command. A volume label can be added, changed, or deleted *after* a disk has already been formatted by using the LABEL command (PC-DOS versions 3.0 and later, MS-DOS versions 3.1 and later). The CHKDSK, DIR, and TREE commands also display a disk's volume label as part of their output.

Example

To display the volume label for the disk in the current drive, type

```
C>VOL  <Enter>
```

If the disk's name is HARDDISK, the VOL command produces the following output:

```
Volume in drive C is HARDDISK
```

Messages

Invalid drive specification
The drive specified in the command line is invalid or does not exist in the system.

Volume in drive *X* has no label
The disk in the current or specified drive was not previously assigned a volume label with the FORMAT or LABEL command.

XCOPY

3.2

Copy Files

External

Purpose

Copies files and directories, optionally also copying subdirectories and the files they contain.

Syntax

XCOPY *source* [*destination*][/A] [/D:*mm-dd-yy*] [/E] [/M] [/P] [/S] [/V] [/W]

where:

source is the name of the file(s) to be copied, optionally preceded by a drive and/or path; wildcard characters are permitted in the filename. If the path is omitted, a drive letter must be specified; this parameter is not optional.

destination is the destination location and, optionally, the name for the copied files, and can be preceded by a drive; wildcard characters are permitted in the filename.

/A copies only those source files with the archive bit set.

/D:*mm-dd-yy* copies only files modified on or after the specified date. (The date format depends on the COUNTRY command in effect, if any.)

/E copies empty subdirectories; if this switch is used, the /S switch must also be specified.

/M copies only those files with the archive bit set; also turns off the archive bit of each source file after it is copied.

/P prompts the user for confirmation before copying each file.

/S copies all nonempty subdirectories of *source* and the files they contain.

/V performs read-after-write verification of destination file(s).

/W waits for the user to press a key before copying any files, allowing disks to be changed.

Description

The XCOPY command copies one or more source files to one or more destination files. Unlike the COPY command, however, a single XCOPY command can copy *all* files contained in the entire hierarchical file structure of the source disk to the destination disk, creating a corresponding set of directories and subdirectories at the destination to hold the copied files.

The *source* parameter identifies the file or files to be copied. It can consist of any combination of a drive, path, and filename (optionally including wildcards) but *must* include either a drive or a pathname. If only a drive is specified, all files in the current directory of that drive are copied. If a path without a drive or filename is specified, all files in the named directory are copied from the current drive.

The *destination* parameter can also consist of any combination of drive, path, and filename. Unless only a single file is being copied and it is also being renamed as part of the XCOPY operation, *destination* is usually simply a drive and/or path specifying where to place the copied file. If *destination* includes a filename, XCOPY displays a message asking if the specified destination is a file or a directory. Depending on the user's response, XCOPY then either copies the source file to a destination file with the specified name or creates a directory with the specified name and copies the source files into it. (Note that if the user responds that the destination is to be a file and multiple source files were specified in the command line, only the last source file is copied to the specified destination.) If no destination is specified, the source file is copied to a file with the same name in the current directory of the current drive.

The /A, /D:*mm-dd-yy*, /M, and /P switches allow selective copying of files. The /A switch is used to copy only source files with the archive bit set; the /M switch also copies only source files with the archive bit set but turns off each source file's archive bit after the file is copied. The /D:*mm-dd-yy* switch is used to copy files that were modified on or after a selected date; the date must be entered in one of the formats discussed in the entry for the system's DATE command or in the format of the COUNTRY command currently in effect (*see* USER COMMANDS: CONFIG.SYS: COUNTRY). The /P switch causes XCOPY to prompt the user for confirmation before transferring each file.

The /E and /S switches allow an entire branch of the source disk's hierarchical directory structure to be copied. If the /S switch is specified, XCOPY copies all nonempty subdirectories of *source*, creating equivalent destination subdirectories, if necessary, to hold the files. If the /E switch is specified, XCOPY also duplicates empty source subdirectories in the equivalent destination locations. If the /E switch is used, the /S switch must also be specified.

The /V switch causes a Verify call to be issued on the destination file(s) to ensure that the data was written correctly. Its effect is equivalent to that of the VERIFY ON command.

Finally, the /W switch causes XCOPY to wait for the user to press a key before copying any files, thus allowing an exchange of disks before the files are transferred. This is useful in systems without a fixed disk, because it allows XCOPY to be used when the program itself is not on either the source or the destination disk.

Note: With MS-DOS versions of XCOPY, the related program MCOPY can be created by simply copying the file XCOPY.EXE to a file named MCOPY.EXE using the following command:

```
C>COPY /B XCOPY.EXE MCOPY.EXE  <Enter>
```

What distinguishes MCOPY from XCOPY is the program name; when either program is loaded, it looks at the name under which it was invoked and reconfigures itself accordingly. MCOPY's behavior is similar to XCOPY's, except that MCOPY automatically determines whether the name specified as the destination is a file or a directory according to the following rules:

- If the source is a directory, the specified destination is a directory.
- If the source includes multiple files, the specified destination is a directory.
- If the destination name ends with a backslash character (\), the specified destination is a directory.

MCOPY supports all the XCOPY switches.

Not all implementations of XCOPY can be renamed to MCOPY and function accordingly. The PC-DOS version of XCOPY, for example, does not support this feature.

Return Codes

0 No errors were detected during the copy operation.
1 No files were found to copy.
2 The copy operation was terminated by a Ctrl-C or Ctrl-Break.
4 Initialization error occurred: not enough memory, file not found, or command-line syntax error.
5 The copy operation was terminated by an *A* response to an *Abort, Retry, Ignore?* prompt.

Examples

To copy all files in the directory C:\SOURCE to the directory C:\SOURCE\BACKUP, type

```
C>XCOPY C:\SOURCE\*.* C:\SOURCE\BACKUP  <Enter>
```

To copy all files and directories on drive C to the disk in drive D, type

```
C>XCOPY C:\*.* D: /S /E  <Enter>
```

Messages

***nn* File(s) copied**
This informational message is displayed at the completion of an XCOPY command and indicates the total number of source files processed.

***filename* File not found**
The source file specified in the command line is invalid or does not exist.

***X:pathname* (Y/N)?**
The /P switch was specified in the command line. XCOPY displays the name of each file, preceded by a drive (and path, if one was specified), and asks for confirmation before copying the file.

Access denied
A destination file could not be overwritten because it was marked read-only.

Cannot COPY from a reserved device
A character device such as AUX or COM1 cannot be the source of an XCOPY operation.

Cannot COPY to a reserved device
A character device such as PRN cannot be the destination of an XCOPY operation.

Cannot perform a cyclic copy
The command line included a /S switch and the destination directory is a subdirectory of the source directory. A subdirectory cannot be copied onto itself.

Does *name* specify a file name
or directory name on the target
(F = file, D = directory)?
The specified destination directory does not already exist; the user is prompted to determine whether it should be created. Respond with *F* to copy the source file to a file named *name*; respond with *D* to create a subdirectory named *name* and copy the source file into it.

File cannot be copied onto itself
The name and location of the source file are the same as the name and location of the destination file.

File creation error
A destination file or directory could not be created. The destination disk may be full.

Incorrect DOS version
The version of XCOPY is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The disk does not contain enough available space to perform the specified XCOPY operation.

Insufficient memory
The available system memory is insufficient to perform the XCOPY operation.

Invalid date
The command included a /D switch and the date was not formatted properly.

Invalid drive specification
The source or destination drive specified in the command line is not valid or does not exist in the system.

Invalid number of parameters
The command line contained too many or too few filenames or other parameters.

Invalid parameter
A switch supplied in the command line is not valid.

Invalid path
A directory specified in the command line is invalid or does not exist.

Lock Violation
XCOPY attempted to access a file in use by another program. Respond with *A* to the error-message prompt and try XCOPY later or wait for a few minutes and respond with *R*.

Path not found
One of the pathnames specified in the command line is invalid or does not exist.

Path too long
The path element of the source or destination parameter was longer than 63 characters.

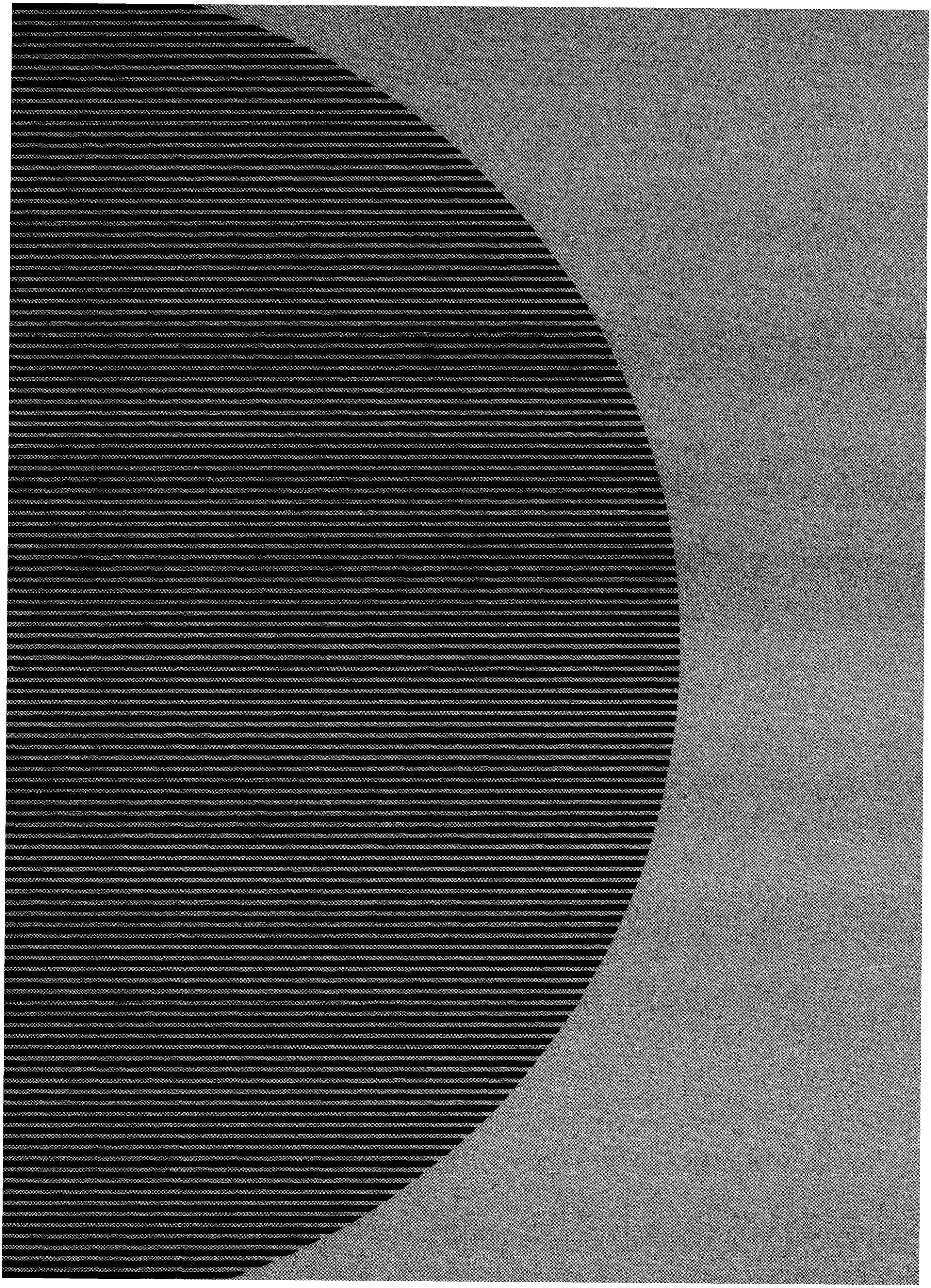
Press any key to begin copying file(s)
The /W switch was specified in the command line and XCOPY waits for the user to press a key before beginning the copy process.

Reading source file(s)...
This informational message is displayed during the XCOPY operation.

Sharing violation
XCOPY attempted to access a file in use by another program. Respond with *A* to the error-message prompt and try XCOPY later or wait a few minutes and respond with *R*.

Too many open files
XCOPY failed due to a lack of available system file handles. Increase the size of the FILES command in the CONFIG.SYS file, restart the system, and attempt the XCOPY command again.

Unable to create directory
A destination directory cannot have the same name as an existing file in the prospective parent directory.



Section IV
Programming Utilities

Introduction

This section of *The MS-DOS Encyclopedia* describes the Microsoft utilities, documentation aids, and debuggers that can be used with the Microsoft C, FORTRAN, Pascal, and BASIC compilers and with the Microsoft Macro Assembler (MASM). Included are operating instructions for MASM, the Macro Assembler; LIB, the Library Manager; LINK, the Microsôft Object Linker; the DEBUG, SYMDEB, and CodeView program debuggers; MAKE, which automates maintenance of programs; CREF, which produces a cross-reference listing of symbols; and EXE2BIN, EXEMOD, and EXEPACK, which modify executable files.

Entries (except for the program debuggers) are arranged alphabetically by the name of the programming utility. The three Microsoft debuggers are listed at the end of the section in the following order: DEBUG, SYMDEB, CodeView. Individual DEBUG and SYMDEB commands appear alphabetically under the headings DEBUG and SYMDEB.

Each utility entry includes

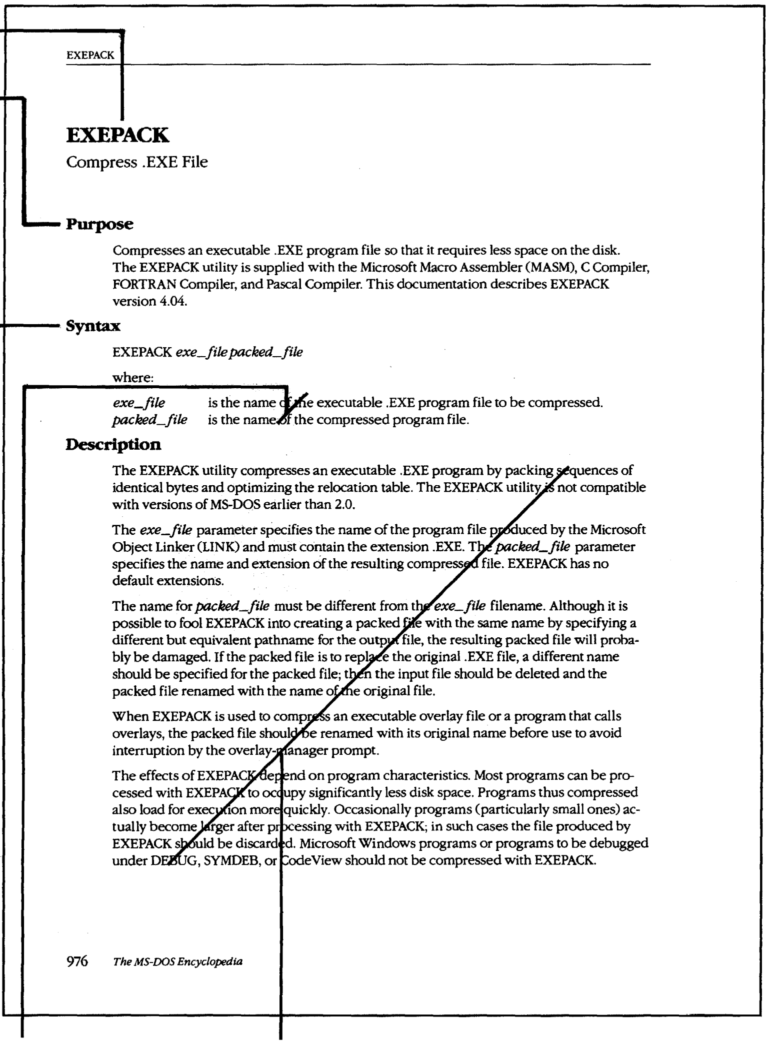
- Utility name
- Utility purpose
- Prototype command line and summary of options
- Detailed description of utility
- One or more examples of utility use
- Return codes (where applicable)
- Error messages and warnings (where applicable)

The experienced user can find information with a quick glance at the first part of a utility entry; a less experienced user can refer to the detailed explanation and examples in a more leisurely fashion. The next two pages contain an example of a typical entry from the Programming Utilities section, with explanations of each component.

HEADING
The utility name.

PURPOSE
An abstract of utility purpose and usage plus a statement of which Microsoft products the utility is supplied with and the utility version described in the entry.

SYNTAX
A prototype command line, with variable names in italic and optional parameters in square brackets. The various elements of the command line should be entered in the order shown. Any punctuation must be used exactly as shown; in commands that use commas as separators, the comma usually must be included as a placeholder even if the parameter is omitted. Except where noted, commands, parameters, and switches can be entered in either uppercase or lowercase. Utility names can be preceded by a drive and/or path.



EXEPACK

EXEPACK

Compress .EXE File

Purpose

Compresses an executable .EXE program file so that it requires less space on the disk. The EXEPACK utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEPACK version 4.04.

Syntax

EXEPACK *exe_filepacked_file*

where:

exe_file is the name of the executable .EXE program file to be compressed.
packed_file is the name of the compressed program file.

Description

The EXEPACK utility compresses an executable .EXE program by packing sequences of identical bytes and optimizing the relocation table. The EXEPACK utility is not compatible with versions of MS-DOS earlier than 2.0.

The *exe_file* parameter specifies the name of the program file produced by the Microsoft Object Linker (LINK) and must contain the extension .EXE. The *packed_file* parameter specifies the name and extension of the resulting compressed file. EXEPACK has no default extensions.

The name for *packed_file* must be different from the *exe_file* filename. Although it is possible to fool EXEPACK into creating a packed file with the same name by specifying a different but equivalent pathname for the output file, the resulting packed file will probably be damaged. If the packed file is to replace the original .EXE file, a different name should be specified for the packed file; then the input file should be deleted and the packed file renamed with the name of the original file.

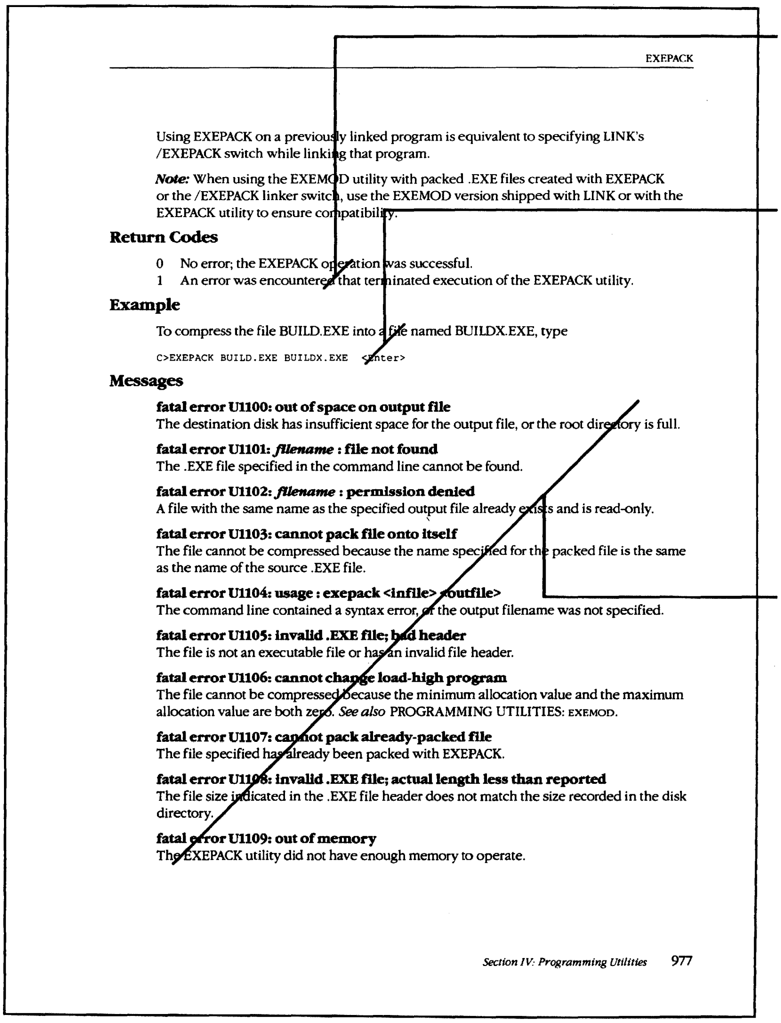
When EXEPACK is used to compress an executable overlay file or a program that calls overlays, the packed file should be renamed with its original name before use to avoid interruption by the overlay-manager prompt.

The effects of EXEPACK depend on program characteristics. Most programs can be processed with EXEPACK to occupy significantly less disk space. Programs thus compressed also load for execution more quickly. Occasionally programs (particularly small ones) actually become larger after processing with EXEPACK; in such cases the file produced by EXEPACK should be discarded. Microsoft Windows programs or programs to be debugged under DEBUG, SYMDEB, or CodeView should not be compressed with EXEPACK.

976 *The MS-DOS Encyclopedia*

BELOW WHERE
A brief explanation of each command parameter and switch. Filenames are always listed first, followed by the switches in alphabetic order. Any special position required for a filename or switch is shown in the syntax line and noted in the explanation.

DESCRIPTION
A detailed description of the utility, including a full explanation of default values, possible interactions of command parameters and options, useful background information, and any applicable warnings.



EXEPACK

Using EXEPACK on a previously linked program is equivalent to specifying LINK's /EXEPACK switch while linking that program.

Note: When using the EXEMOD utility with packed .EXE files created with EXEPACK or the /EXEPACK linker switch, use the EXEMOD version shipped with LINK or with the EXEPACK utility to ensure compatibility.

Return Codes

0 No error; the EXEPACK operation was successful.
1 An error was encountered that terminated execution of the EXEPACK utility.

Example

To compress the file BUILD.EXE into a file named BUILDX.EXE, type

```
C>EXEPACK BUILD.EXE BUILDX.EXE  <Enter>
```

Messages

fatal error U1100: out of space on output file
The destination disk has insufficient space for the output file, or the root directory is full.

fatal error U1101: *filename* : file not found
The .EXE file specified in the command line cannot be found.

fatal error U1102: *filename* : permission denied
A file with the same name as the specified output file already exists and is read-only.

fatal error U1103: cannot pack file onto itself
The file cannot be compressed because the name specified for the packed file is the same as the name of the source .EXE file.

fatal error U1104: usage : exepack <infile> <outfile>
The command line contained a syntax error, or the output filename was not specified.

fatal error U1105: invalid .EXE file; bad header
The file is not an executable file or has an invalid file header.

fatal error U1106: cannot change load-high program
The file cannot be compressed because the minimum allocation value and the maximum allocation value are both zero. *See also* PROGRAMMING UTILITIES: EXEMOD.

fatal error U1107: cannot pack already-packed file
The file specified has already been packed with EXEPACK.

fatal error U1108: invalid .EXE file; actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

fatal error U1109: out of memory
The EXEPACK utility did not have enough memory to operate.

Section IV: Programming Utilities 977

RETURN CODES
Exit codes returned by the utility (if any) that can be tested in a batch file or by another program.

EXAMPLES
One or more examples of the utility at work, including examples of the resulting output where appropriate. User entry appears in color; do not type the prompt, which appears in black. Press the Enter key (labeled Return on some keyboards) as directed at the end of each command line.

MESSAGES
An alphabetic list of messages that may be displayed when the utility is used. Following each message is a brief explanation of the condition that produces the message and, where appropriate, any action that should be taken.

CREF

Generate Cross-Reference Listing

Purpose

Produces a cross-reference listing of all symbols in an assembly-language program. The CREF utility is supplied with the Microsoft Macro Assembler (MASM). This documentation describes CREF version 4.0.

Syntax

CREF

or

CREF *crf_file*[;]

or

CREF *crf_file,ref_file*

where:

crf_file	is the input file previously produced by MASM (default extension = .CRF).
ref_file	is the output ASCII text file to be created (default extension = .REF).

Description

The CREF utility processes a file produced by MASM and generates an ASCII cross-reference listing in a file on disk or directly on a character device (such as a printer). The output file contains an alphabetic list of the symbols in the assembled program, including the line number of each reference to the symbol and the total number of symbols in the program. A pound sign (#) follows the line number of the reference that defines the symbol.

The *crf_file* has the default extension .CRF. It is produced by providing MASM with a filename other than NUL in the cross-reference position in the command line, by responding to the *Cross-reference:* prompt, or by including the /C switch in the MASM command line or at any MASM prompt. An assembly source listing file (.LST) must also be requested in the MASM command line or in response to the MASM prompts in order to generate a valid .CRF file.

If a semicolon follows the *crf_file* parameter in the CREF command, the resulting *ref_file* containing the cross-reference listing is given the same drive and pathname as *crf_file*, with a .REF extension. If the optional *ref_file* parameter is present, it can consist of any pathname with an optional extension (default is .REF). The cross-reference listing can be sent directly to a character device, rather than to a file, by specifying a valid character device name (such as PRN) in the *ref_file* position.

If the CREF utility is run without any parameters or with some parameters missing, the CREF utility prompts the operator for the necessary information.

Return Codes

0 No error; the CREF operation was successful.
1 An error was encountered that terminated execution of the CREF utility.

Examples

To process the file MENUMGR.CRF (created during assembly of MENUMGR.ASM) into the cross-reference file MENUMGR.REF, type

```
C>CREF MENUMGR;  <Enter>
```

To process the file MENUMGR.CRF and assign the name MENU.REF to the resulting cross-reference file, type

```
C>CREF MENUMGR,MENU  <Enter>
```

To process the file MENUMGR.CRF and send the cross-reference listing directly to the printer, type

```
C>CREF MENUMGR,PRN  <Enter>
```

To run the CREF program in interactive mode, type

```
C>CREF  <Enter>
```

The following is an example of an interactive CREF session:

```
C>CREF  <Enter>
Microsoft (R) Cross Reference Utility  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights reserved.

Cross-reference [.CRF]: MENUMGR  <Enter>
Listing [MENUMGR.REF]:  <Enter>

9 Symbols

C>
```

The following sequence of commands produces the cross-reference listing HELLO.REF from the assembly-language source file HELLO.ASM:

```
C>MASM HELLO,HELLO,HELLO,HELLO  <Enter>
C>CREF HELLO;  <Enter>
```

Contents of the file HELLO.ASM:

```
        name    hello
        page    55,132
        title   HELLO.ASM - print Hello on terminal
;
; HELLO.COM utility to demonstrate CREF listing
;
cr      equ     0dh             ;ASCII carriage return
lf      equ     0ah             ;ASCII linefeed

cseg    segment para public "CODE"

        org     100h

        assume  cs:cseg,ds:cseg,es:cseg,ss:cseg

print   proc    near
        mov     dx,offset message
        mov     ah,9            ;print the string "Hello"
        int     21h
        mov     ax,4c00h        ;exit to MS-DOS
        int     21h             ;with "return code" of zero
print   endp

message db      cr,lf,'Hello!',cr,lf,'$'

cseg    ends

        end     print
```

Contents of the file HELLO.REF:

```
Microsoft Cross-Reference  Version 4.00            Mon Sep 07 23:31:21 1987
HELLO.ASM - print Hello on terminal

  Symbol Cross-Reference          (# is definition)                  Cref-1

CODE . . . . . . . . . . . . . .  10
CR . . . . . . . . . . . . . . .   7      7#     24     25
CSEG . . . . . . . . . . . . . .  10     10#     14     14     14     14     27

LF . . . . . . . . . . . . . . .   8      8#     24     25

MESSAGE. . . . . . . . . . . . .  17     24      24#

PRINT. . . . . . . . . . . . . .  16     16#     29


 6 Symbols
```

Messages

can't open cross-reference file for reading
The pathname or drive specified for the input .CRF file is invalid or does not exist.

can't open listing file for writing
A write error has halted the creation of the .REF listing file. This indicates that the disk is full or write-protected, that the specified output file is read-only, or that the specified device is not available.

cref has no switches
A switch was specified in the command line; CREF has no optional switches.

DOS 2.0 or later required
CREF does not work with versions of MS-DOS earlier than 2.0.

extra file name ignored
More than two filenames were specified in the command line. The CREF utility generates the cross-reference listing using the first two filenames specified.

line invalid, start again
No .CRF file was specified in the command line or at the prompt. Specify a valid .CRF file at the prompt following this message.

out of heap space
Memory is insufficient to process the .CRF file. Remove memory-resident programs and shells or add more memory.

premature eof
The input file specified is damaged or is not a valid .CRF file.

read error on stdin
A Control-Z was received from the keyboard or a redirected file and has halted CREF.

EXE2BIN

Convert .EXE File to Binary-Image File

Purpose

Converts an executable file in the .EXE format to a memory-image file in binary format. The EXE2BIN utility is supplied with the MS-DOS distribution disks.

Syntax

EXE2BIN *exe_file* [*bin_file*]

where:

exe_file	is the .EXE-format file to be converted (default extension = .EXE).
bin_file	is the name to be given to the converted file (default extension = .BIN).

Description

The .EXE executable program files produced by the Microsoft Object Linker (LINK) contain a special header and a relocation table as well as the program code and data. The EXE2BIN utility can be used to convert a .EXE file to a .COM executable file, which is an absolute memory image of the program to be executed and does not contain a special header or relocation table. The EXE2BIN utility can also be used to convert .EXE files with an origin of zero (such as installable MS-DOS device drivers) to pure memory-image files. Files in memory-image format (a common format for device drivers and for programs to be placed in ROM for execution) usually have a .BIN or .SYS extension.

To convert a .EXE program to a binary-image file, the following are required:

- The program must be a valid .EXE file produced by LINK.
- The program can contain only one segment and cannot contain a declared stack segment.
- The program code and data portion of the .EXE file must be less than 64 KB.

To convert a .EXE program to an executable .COM file, the following are required:

- The origin of the program must be 0100H, which must also be specified as the entry point.
- The program code and data portion of the .EXE file must be less than 65227 bytes (64 KB minus 256 bytes used by the program segment prefix minus 2 bytes initially placed on the stack).
- The program must not include any FAR references.

Note: Many compilers cannot create programs that can be converted to .COM files. Check the compiler documentation for specific information concerning executable .COM files.

The *exe_file* parameter in the command line can have any filename and can include a drive and path; the default extension is .EXE. The optional *bin_file* parameter can also contain any filename and a drive and path; the default extension is .BIN. If no path is specified with the *bin_file* parameter, the output file is given the same drive and path as the *exe_file.* If no *bin_file* parameter is supplied, the output file is given the same name as the *exe_file,* with the extension .BIN.

If the program in the .EXE file requires segment fixups (that is, if the program contains instructions requiring segment relocation, which would ordinarily be done by the MS-DOS loader using the .EXE file's relocation table), EXE2BIN prompts for a base segment address. When segment fixups are necessary, the resulting program is not relocatable and must be loaded at the given location to be executed; the MS-DOS loader cannot load the program.

Examples

To convert the file HELLO.EXE to the file HELLO.BIN, type

```
C>EXE2BIN HELLO  <Enter>
```

To convert the file CLEAN.EXE, which has an origin of 0100H and meets the requirements for an executable .COM file, to the file CLEAN.COM, type

```
C>EXE2BIN CLEAN.EXE CLEAN.COM  <Enter>
```

To convert the file ASYNCH.EXE, produced by assembling and linking the device-driver source file ASYNCH.ASM, to the installable device-driver file ASYNCH.SYS, type

```
C>EXE2BIN ASYNCH.EXE ASYNCH.SYS  <Enter>
```

Messages

File cannot be converted
The program to be converted has one of the following problems: The program has an origin of 0100H but a different entry point; the program requires segment fixups; the program code and data are larger than 64 KB; the program has more than one declared segment; or the file is not a valid .EXE-format file.

File creation error
EXE2BIN cannot create the output file because a read-only file with the same name already exists, because the specified directory is full, or because the specified disk is full, write-protected, or unreadable.

File not found
The file does not exist or the incorrect path was given.

Fixups needed - base segment (hex):
The .EXE-format file contains segment references that would ordinarily be relocated by the .EXE file loader. Specify the absolute segment address at which the converted module will be executed.

Incorrect DOS version
The version of EXE2BIN is not compatible with the version of MS-DOS that is running.

Insufficient disk space
The destination disk has insufficent space to create the memory-image output file.

Insufficient memory
Not enough memory is available to run EXE2BIN.

WARNING - Read error in EXE file.
Amount read less than size in header.
The file size given in the .EXE header is inconsistent with the actual size of the file.

EXEMOD

Modify .EXE File Header

Purpose

Allows inspection or modification of the fields in a .EXE file header. The EXEMOD utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEMOD version 4.02.

Syntax

EXEMOD *exe_file*[/H]

or

EXEMOD *exe_file*[/STACK *n*][/MAX *n*][/MIN *n*]

where:

exe_file	is the name of an executable program in .EXE format (the extension .EXE is assumed).
/H	displays the values in the file's header.
/STACK *n*	modifies the size of the program's stack segment to *n* (hexadecimal) bytes.
/MAX *n*	sets the maximum memory allocation for the program to *n* (hexadecimal) paragraphs.
/MIN *n*	sets the minimum memory allocation for the program to *n* (hexadecimal) paragraphs.

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

Programs that are executable under MS-DOS can be in one of two file formats: .COM, which is an absolute image of the file to be executed and limits the program size to 65227 bytes (64 KB minus 256 bytes used by the program segment prefix minus 2 bytes initially placed on the stack); or .EXE, which allows a program of any size to be loaded and has a special header containing information about the program's entry point, stack size, and memory requirements, plus a relocation table.

The EXEMOD utility can be used to display or modify those fields of a .EXE program header that control the size of the stack segment and the amount of memory allocated to the program when MS-DOS loads the program into the transient program area for execution.

The /STACK*n* switch controls the number of bytes in the program's STACK segment by setting the initial SP to the hexadecimal value specified. The minimum paragraph allocation value is adjusted if necessary. The EXEMOD /STACK*n* switch should be used only with programs compiled by Microsoft C version 3.0 or later, Microsoft Pascal version 3.3

or later, or Microsoft FORTRAN version 3.0 or later. Use of the /STACK*n* switch with a program developed with another compiler can cause the program to fail or cause EXEMOD to return an error message.

The /MAX*n* switch specifies the maximum number of additional paragraphs of memory to allocate for use by the program. The /MIN*n* switch specifies the minimum number of paragraphs of memory, in addition to the size of the program itself and its stack and data segments, that are required for the program to execute. If enough memory exists to satisfy the minimum additional paragraphs requested but not enough exists to satisfy the maximum, MS-DOS allocates all available memory to the program.

To display the current memory allocation and stack size values from a .EXE file's header, the /H switch can be used or the file's name can be entered as the only parameter in the command line.

When EXEMOD is used on a previously packed .EXE file (a file that was processed by EXEPACK or linked with the /EXEPACK switch), the values set or displayed in the file's header are the values that will apply after the file is expanded at load time. EXEMOD displays a message advising the user that the file being modified was previously packed.

The EXEMOD switches /MAX*n* and /STACK*n* correspond to the Microsoft Object Linker's /CPARMAXALLOC:*n* and /STACK:*n* switches, respectively. *See* PROGRAMMING UTILITIES: LINK.

Return Codes

0 No error; EXEMOD operation was successful.
1 An error was encountered that terminated execution of the EXEMOD program.

Examples

To display the values in the file header of the DUMP.EXE program, type

```
C>EXEMOD DUMP.EXE  <Enter>
```

or

```
C>EXEMOD DUMP.EXE /H  <Enter>
```

The EXEMOD utility displays the following:

```
Microsoft (R) EXE File Header Utility  Version 4.02
Copyright (C) Microsoft Corp 1985.  All rights reserved.
DUMP.EXE                           (hex)           (dec)

.EXE size (bytes)                    580            1408
Minimum load size (bytes)            383             899
Overlay number                         0               0
Initial CS:IP                  0000:0000
Initial SS:SP                  0034:0040              64
Minimum allocation (para)              5               5
Maximum allocation (para)           FFFF           65535
Header size (para)                    20              32
Relocation table offset               20              32
Relocation entries                     1               1
```

To change the size of the STACK segment for the DUMP.EXE program to 400H (1024) bytes, type

```
C>EXEMOD DUMP.EXE /STACK 400  <Enter>
```

EXEMOD displays the message

```
EXEMOD : warning U4051: minimum allocation less than stack; correcting minimum
```

Messages

error U1050: usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]
An error was detected in the EXEMOD command line.

error U1051: invalid .EXE file : bad header
The file is not an executable file or has an invalid file header.

error U1052: invalid .EXE file : actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

error U1053: cannot change load-high program
The header of the file cannot be modified because the minimum allocation value and the maximum allocation value are both zero.

error U1054: file not .EXE
The file specified does not have a .EXE extension.

error U1055: *filename* : cannot find file
The .EXE file specified in the command line cannot be found.

error U1056: *filename* : permission denied
The .EXE file specified in the command line is read-only.

warning U4050: packed file
The specified file is a packed file; that is, it was previously processed with the EXEPACK utility or was linked with the /EXEPACK switch. This is an informational message only; EXEMOD still modifies the file. The header values displayed are the values that will apply after the packed value is expanded at load time.

warning U4051: minimum allocation less than stack; correcting minimum
The minimum allocation value is not large enough to accommodate the stack; the minimum allocation value is adjusted. This is an informational message only.

warning U4052: minimum allocation greater than maximum; correcting maximum
If the minimum allocation value is greater than the maximum allocation value, the maximum value is adjusted. This is an informational message only.

EXEPACK

Compress .EXE File

Purpose

Compresses an executable .EXE program file so that it requires less space on the disk. The EXEPACK utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes EXEPACK version 4.04.

Syntax

EXEPACK *exe_filepacked_file*

where:

exe_file	is the name of the executable .EXE program file to be compressed.
packed_file	is the name of the compressed program file.

Description

The EXEPACK utility compresses an executable .EXE program by packing sequences of identical bytes and optimizing the relocation table. The EXEPACK utility is not compatible with versions of MS-DOS earlier than 2.0.

The *exe_file* parameter specifies the name of the program file produced by the Microsoft Object Linker (LINK) and must contain the extension .EXE. The *packed_file* parameter specifies the name and extension of the resulting compressed file. EXEPACK has no default extensions.

The name for *packed_file* must be different from the *exe_file* filename. Although it is possible to fool EXEPACK into creating a packed file with the same name by specifying a different but equivalent pathname for the output file, the resulting packed file will probably be damaged. If the packed file is to replace the original .EXE file, a different name should be specified for the packed file; then the input file should be deleted and the packed file renamed with the name of the original file.

When EXEPACK is used to compress an executable overlay file or a program that calls overlays, the packed file should be renamed with its original name before use to avoid interruption by the overlay-manager prompt.

The effects of EXEPACK depend on program characteristics. Most programs can be processed with EXEPACK to occupy significantly less disk space. Programs thus compressed also load for execution more quickly. Occasionally programs (particularly small ones) actually become larger after processing with EXEPACK; in such cases the file produced by EXEPACK should be discarded. Microsoft Windows programs or programs to be debugged under DEBUG, SYMDEB, or CodeView should not be compressed with EXEPACK.

Using EXEPACK on a previously linked program is equivalent to specifying LINK's /EXEPACK switch while linking that program.

Note: When using the EXEMOD utility with packed .EXE files created with EXEPACK or the /EXEPACK linker switch, use the EXEMOD version shipped with LINK or with the EXEPACK utility to ensure compatibility.

Return Codes

0 No error; the EXEPACK operation was successful.
1 An error was encountered that terminated execution of the EXEPACK utility.

Example

To compress the file BUILD.EXE into a file named BUILDX.EXE, type

```
C>EXEPACK BUILD.EXE BUILDX.EXE  <Enter>
```

Messages

fatal error U1100: out of space on output file
The destination disk has insufficient space for the output file, or the root directory is full.

fatal error U1101: *filename* : file not found
The .EXE file specified in the command line cannot be found.

fatal error U1102: *filename* : permission denied
A file with the same name as the specified output file already exists and is read-only.

fatal error U1103: cannot pack file onto itself
The file cannot be compressed because the name specified for the packed file is the same as the name of the source .EXE file.

fatal error U1104: usage : exepack <infile> <outfile>
The command line contained a syntax error, or the output filename was not specified.

fatal error U1105: invalid .EXE file; bad header
The file is not an executable file or has an invalid file header.

fatal error U1106: cannot change load-high program
The file cannot be compressed because the minimum allocation value and the maximum allocation value are both zero. *See also* PROGRAMMING UTILITIES: EXEMOD.

fatal error U1107: cannot pack already-packed file
The file specified has already been packed with EXEPACK.

fatal error U1108: invalid .EXE file; actual length less than reported
The file size indicated in the .EXE file header does not match the size recorded in the disk directory.

fatal error U1109: out of memory
The EXEPACK utility did not have enough memory to operate.

fatal error U1110: error reading relocation table
The file cannot be compressed because the relocation table cannot be found or is invalid.

fatal error U1111: file not suitable for packing
The file could not be packed because the packed load image of the specified file was larger than the unpacked load image.

fatal error U1112: *filename* : unknown error
An unknown system error occurred while the specified file was being processed.

warning U4100: omitting debug data from output file
EXEPACK has stripped all symbolic debug information from the output file.

LIB

Library Manager

Purpose

Creates or modifies an object module library file. The LIB utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, FORTRAN Compiler, and Pascal Compiler. This documentation describes LIB version 3.06.

Syntax

LIB

or

LIB *library_file* [/PAGESIZE:*n*] [*operation*][,[*list_file*][,[*new_library_file*]]] [;]

or

LIB @*response_file*

where:

library_file is the name of the object module library file to be created or modified (default extension = .LIB).

/PAGESIZE:*n* is the page size of the library file and must immediately follow *library_file* if used; *n* is a power of 2 between 16 and 32768, inclusive (default = 16). Can be abbreviated /P:*n*.

operation is one or more library manipulations to be performed. Each *operation* is specified as a code followed by an object module name (case is not significant):

+*name* Add object module or another library to library.
–*name* Delete object module from library.
–+*name* Replace object module in library.
*name Copy object module from library to object file.
–*name Copy object module to object file and then delete object module from library.

list_file is the name of the file or character device to receive the cross-reference listing for the library file (default = NUL device).

new_library_file is the name to be assigned to the modified object module library file. (The default name is the same as *library_file*; if the default is used, the original *library_file* is renamed with the extension .BAK.)

response_file is the name of a text file containing LIB parameters in the same order in which they are supplied if entered interactively. The name of the response file must be preceded by the @ symbol.

Description

The Microsoft Library Manager (LIB) creates and modifies library files, checks existing library files for consistency, and prints listings of the contents of library files. The LIB utility does not work with versions of MS-DOS earlier than 2.0.

A library file consists of relocatable object modules that are indexed by their names and public symbols. The Microsoft Object Linker (LINK) uses these files during the creation of an executable (.EXE) program to resolve external references to routines and variables contained in other object modules.

The *library_file* parameter specifies the name of the object module library file to be created or modified. This parameter is required; if it is not included, LIB prompts for it. The default extension for a library file is .LIB.

The /PAGESIZE:*n* switch (abbreviated /P:*n*) sets the page size (in bytes) for a new library file or changes the page size of an existing library file. The value of *n* must be a power of 2 between 16 and 32768, inclusive. The default is 16 for a new library file; for an existing library file, the default is the current page size. Because the index to a library file is contained in a fixed number of pages, setting a larger page size increases the number of index entries (and thus the number of object modules) that a library file can contain but results in more wasted disk space (an average of half a library page per object module).

The *operation* parameter specifies one or more relocatable object modules to add to, replace in, copy from, move from, or delete from *library_file.* Each operation is represented by a code specifying the type of operation, followed by the object module name. When an object module is copied or moved from the library file, the drive and pathname of the object module are set to the default drive, current directory, and specified module name, and the extension of the object module defaults to .OBJ. When an object module is added or replaced, LIB assumes a default extension of .OBJ.

The operation +*name* adds the object module in the file *name*.OBJ to the library file. This operation can also be used to add the contents of another entire object module library file to the library file being updated, in which case the extension .LIB must be included in *name.* The operation –*name* deletes the object module *name* from the library. The operation –+*name* deletes the object module *name* from the library file and replaces it with the contents of the file *name*.OBJ. The operation **name* copies the object module *name* from the library file into the file *name*.OBJ, which LIB creates in the current directory. The operation –**name* also copies the object module *name* from the library file into a .OBJ file but then deletes the module from the library file. (Although *name* must have exactly the same spelling as the name in the library's reference listing, case is not significant.)

Note: LIB does not actually delete object modules from the specified library file. Instead, it marks the selected object modules for deletion, creates a new library file, and copies only the modules not marked for deletion into the new file. Thus, if LIB is terminated for any reason, the original file is not lost. Enough space must be available on the disk for both the original library file and the copy.

The *list_file* parameter specifies the file or character device to receive a reference listing for the library file. Any valid drive, pathname, and extension or any valid character device, such as PRN, is permitted (default = NUL). If this parameter is omitted, no listing is generated.

The reference listing consists of two tables. The first table contains all the public symbols in the object modules in the library, listed alphabetically, with each symbol followed by the name of the object module in which it is referenced. The second table contains the names of all the object modules, listed alphabetically, with each name followed by the offset from the start of the library file, the code and data size, and an alphabetic listing of the public symbols in that object module.

The *new_library_file* parameter specifies the name for the modified library file that is created. If this parameter is omitted, LIB gives the modified library file the same name as the original library file, and the original library file is renamed with a .BAK extension. When a new library file is being created, this parameter is not necessary.

When the command line is used to supply LIB with filenames and switches, typing a semicolon character (;) after any parameter (except *library_file*) causes LIB to use the default values for the remaining parameters. If a semicolon is entered after *library_file*, LIB simply checks the file for consistency and usability. (This is seldom necessary, because LIB checks each object module for consistency before adding it to the library.)

If the LIB command is entered without any parameters, LIB prompts the user for each parameter needed. If there are too many operations to fit on one line, the line can be ended with the ampersand character (&), causing LIB to repeat the *Operations:* prompt. If any response except *library_file* is terminated with a semicolon character, LIB uses the default values for the remaining filenames. When the *library_file* parameter is followed by a semicolon or a semicolon is entered at the *Operations:* prompt, LIB takes no action except to verify that the contents of the specified file are consistent and usable.

The *response_file* parameter allows the automation of complex LIB sessions involving many files. A response file contains ASCII text that corresponds line for line to the responses that are entered in a normal interactive LIB session, in the form

library_file [/P:*n*]
[Y]
[*operations*]
[*list_file*]
[*new_library_file*] [;]

The response file name must be preceded in the command line by the at symbol (@) and can also be preceded by a path and/or drive letter. If *library_file* is a new file, the letter *Y* must appear by itself on the second line of the response file to approve the creation of a library file. The last line of the response file must end with a semicolon or a carriage return. (LIB ignores any lines following a semicolon.) If all the parameters required by LIB are not present in the response file or the response file does not end with a semicolon, LIB prompts the user for the missing information.

Return Codes

0 No error; LIB operation was successful.
1 An error that terminated execution of the LIB utility was encountered.

Examples

To create a library file named MYLIB.LIB and insert the object files VIDEO.OBJ, COMM.OBJ, and DOSINT.OBJ, type

```
C>LIB MYLIB +VIDEO +COMM +DOSINT;  <Enter>
```

To print a listing of the object modules in the library file MYLIB.LIB, type

```
C>LIB MYLIB,PRN  <Enter>
```

If the LIB command is entered without parameters, the user is prompted for the necessary information. For example, if the user wanted to add the module VIDEO.OBJ to the library file SLIBC.LIB, produce a reference listing in the file SLIBC.LST, and produce a new output library file named SLIBC2.LIB, the following dialogue would take place:

```
C>LIB  <Enter>

Microsoft (R) Library Manager  Version 3.06
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.

Library name: SLIBC  <Enter>
Operations: +VIDEO  <Enter>
List file: SLIBC.LST  <Enter>
Output library: SLIBC2  <Enter>
```

Messages

***filename*: cannot access file.**
LIB is unable to access an object module specified in a response file, in the command line, or at the *Operations:* prompt.

***filename*: cannot create extract file**
The object module cannot be copied or moved from the library file into a separate disk file called *filename* because the root directory or disk is full or because *filename* already exists and is read-only.

***filename*: cannot create listing**
The list file specified in the response file, in the command line, or at the *List file:* prompt cannot be created because the root directory or disk is full or because *filename* already exists and is read-only.

***filename*: invalid format (*xxxx*); file ignored.**
The hexadecimal signature byte or word *xxxx* of the specified file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX archive.

***filename*: invalid library header.**
The input library file either is not a library file or is damaged.

***filename*: invalid library header; file ignored.**
The input library file is in the wrong format.

modulename*: invalid object module near *location
The specified object module has an invalid format near the hexadecimal offset indicated.

***modulename*: module not in library; ignored**
The object module specified in the response file, in the command line, or at the *Operations:* prompt is not in the specified input library file.

***modulename*: module redefinition ignored**
An object module was specified to be added to a library file but an object module with the same name was already in the library file, or the same object module was specified twice in an add operation in the command line.

***number*: page size too small; ignored**
The size specified with a /P:*n* switch must be a power of 2 between 16 and 32768 bytes, inclusive.

***symbol* (*modulename*) : symbol redefinition ignored**
The specified symbol was defined in more than one module. Only the first definition of a symbol is accepted. All redefinitions are ignored.

cannot create new library
The root directory is full, or a library file with the same name already exists and is read-only.

cannot open response file
The specified response file cannot be found or does not exist.

cannot rename old library
The old library file cannot be renamed with a .BAK extension because such a file already exists and is read-only.

cannot reopen library
The old library file could not be reopened after it was renamed with the .BAK extension. This error usually indicates damage to the operating system or to the disk directory structure.

comma or new line missing
A comma or carriage return was expected in the command line but was not found.

Do not change diskette in drive *X*:
LIB may have placed important temporary files on the specified disk. Do not remove the disk until the LIB operation is complete or these files may be lost.

error writing to cross-reference file
The disk or root directory is full.

error writing to new library
The new library file cannot be created because the disk is full.

free: not allocated
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

insufficient memory
Not enough memory is available in the transient program area for LIB to successfully perform the requested operations.

internal failure
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Library does not exist. Create?
The specified *library_file* does not exist on disk. Respond with *Y* to create the library file; respond with *N* to terminate the LIB utility.

mark: not allocated
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

option unknown
The command line included a switch other than /P:*n*.

output-library specification ignored
An output library file was specified in addition to a new library file. This is only a warning. The output library file specification will be disregarded.

page size too small
The page size of an input library file was less than 16 bytes, indicating a damaged or otherwise invalid .LIB file. *See* LIB message *number*: page size too small; ignored.

syntax error
The command line included an invalid parameter or switch.

syntax error: illegal file specification
A command operator (such as *, –, or +) was given without an object module name.

syntax error: illegal input
The command line included an invalid parameter or switch.

syntax error: option name missing
The command line included a forward slash (/) that was not followed by P:*n*.

syntax error: option value missing
The /P switch was not followed by the page size value in bytes.

terminator missing
Either a control character (such as Control-Z) was specified at the *Output library:* prompt or the response file line that corresponds to LIB's *Output library:* prompt was not terminated by a carriage return or semicolon.

too many symbols
The maximum number of public symbols allowed in a library file has been exceeded. The limit for all object modules (combined) is 4609.

unexpected end-of-file on command input
The response file did not include all the necessary LIB parameters.

write to extract file failed
The destination disk has insufficient space for the complete object module, or the root directory is full.

write to library file failed
The destination disk has insufficient space to create the new library file, or the root directory is full.

LINK

Create .EXE File

Purpose

Combines relocatable object modules into an executable (.EXE) file. The Microsoft Object Linker (LINK) is supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. This documentation describes LINK version 3.50.

Syntax

LINK

or

LINK *obj_file*[+*obj_file*...][,[*exe_file*]][,[*map_file*]][,[*library*[+*library*...]]] [*options*] [;]

or

LINK @*response_file*

where:

obj_file is the name of a file containing a relocatable object module produced by MASM or by a high-level-language compiler (default extension = .OBJ).

exe_file is the name of the executable file to be produced by LINK (default extension = .EXE).

map_file is the name of the file or character device to receive a listing of the names, load addresses, and lengths of the segments in *exe_file* (default = NUL device; default extension = .MAP).

library is the name of an object module library to be searched to resolve external references in the object file(s) (default extension = .LIB).

response_file is the name of a text file containing LINK parameters in the order in which they are supplied during an interactive LINK session.

options specifies one or more of the following switches. Switches can be either uppercase or lowercase.

/CP:*n* (/CPARMAXALLOC:*n*) Sets the maximum number of extra memory paragraphs required by *exe_file* (default = 65535).

/DS (/DSALLOCATE) Loads the data in DGROUP at the high end of the data segment.

/DO (/DOSSEG) Arranges segments according to the Microsoft language segment-ordering convention.

/E (/EXEPACK) Compresses repetitive sequences of bytes and optimizes *exe_file*'s relocation table.

(more)

/HI (/HIGH) Causes *exe_file* to be loaded as high as possible in memory when *exe_file* is executed.

/HE (/HELP) Lists LINK options on the screen. No other switches or filenames should be used with this switch.

/LI (/LINENUMBERS) Copies line-number information (if available) from *obj_file* to *map_file*. If a map file was not specified, this switch creates one.

/M (/MAP) Copies a list of all public symbols declared in *obj_file* to *map_file*. If a map file was not specified, this switch creates one.

/NOD (/NODEFAULTLIBRARYSEARCH) Causes LINK to ignore any library names inserted in the object file by the language compiler.

/NOG (/NOGROUPASSOCIATION) Causes LINK to ignore GROUP associations when assigning addresses.

/NOI (/NOIGNORECASE) Causes LINK to be case sensitive when resolving external names.

/O:*n* (/OVERLAYINTERRUPT:*n*) Overrides the interrupt number used by the overlay manager (0–255, default = 63, or 3FH). This switch should be used only when linking with a run-time module from a language compiler that supports overlays.

/P (/PAUSE) Causes LINK to pause and prompt the user to change disks before writing the *exe_file*.

/SE:*n* (/SEGMENTS:*n*) Sets the maximum number of segments that can be processed (1–1024, default = 128).

/ST:*n* (/STACK:*n*) Sets the size of the *exe_file*'s stack segment to *n* bytes (1–65535).

Description

LINK combines relocatable object modules into an executable file in the .EXE format. LINK can be used with object files produced by any high-level-language compiler or assembler that supports the Microsoft object module format. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: Programming Tools: Object Modules; The Microsoft Object Linker.

The *obj_file* parameter, which is required, specifies one or more files containing relocatable object modules. If multiple object files are linked, their names should be separated by a plus operator (+) or a space. If an extension is not specified for an object file, LINK supplies the extension .OBJ. Some high-level-language compilers support partitioning of the executable program into a root segment and one or more overlay segments and include a special overlay manager in their libraries; when these compilers are used, the object modules that compose each overlay segment should be surrounded with parentheses in the LINK command line.

The *exe_file* parameter specifies the name of the executable file that is created by LINK. The default is the same filename as the first object file, but with the extension .EXE.

The *map_file* parameter designates the file or character device to receive LINK's listing of the name, load address, and length of each of *exe_file*'s segments. The map file also includes the names and load addresses of any groups in the program, the program entry point, and, if the /M switch is used, all public symbols and their addresses. If the /LI switch is used and if line numbers were inserted into *obj_file* by the compiler, the starting address of each *obj_file* program line is also copied to *map_file*. The default extension for a map file is .MAP. If the /M or /LI switch is used, a map file is created using the name of the specified .EXE file even if *map_file* is not specified. If neither the /M nor the /LI switch is used and *map_file* is not specified, no listing is created.

The *library* parameter specifies the object module library or libraries that will be searched to resolve external references after all the object files are processed. The default extension for library files is .LIB. Multiple library names should be separated by plus operators (+) or spaces. A maximum of 16 search paths can be specified in the LINK command line. If a library name is preceded by a drive and/or path, LINK searches only the specified location. If no drive or path precedes a library name, LINK searches for library files in the following order:

1. Current drive and directory
2. Any other library search paths specified in the command line, in the order they were entered
3. Directories specified in the LIB= environment variable, if one exists

In the following example, LINK searches only the \ALTLIB directory on drive A to find the library MATH.LIB. To find the library COMMON.LIB, LINK searches the current directory on the current drive, then the current directory on drive B, then directory \LIB on drive D, and finally, any directories named in the LIB environment variable.

```
C>LINK TEST,,TEST,A:\ALTLIB\MATH.LIB+COMMON+B:+D:\LIB\  <Enter>
```

If default libraries are specified within the object files through special records inserted by certain high-level-language compilers, those libraries will be searched *after* the libraries named in the command line or response file.

If the LINK command is entered without parameters, LINK prompts the user for each filename needed. The default response for each prompt (except the *obj_file* prompt) is displayed in square brackets and can be selected by pressing the Enter key. If there are too many *obj_file* or *library* names to fit on one line, the line can be terminated by entering a plus operator (+) and pressing the Enter key; LINK then repeats the prompt. If the user ends any response with a semicolon character (;), LINK uses the default values for the remaining fields.

When the command line contains filenames and switches, commas must be used to separate the *obj_file*, *exe_file*, *map_file*, and *library* parameters. If a filename is not supplied, a comma must be used to mark its place. If the user places a semicolon after any parameter in the command line, LINK terminates the command line at the semicolon and uses the default values for any remaining parameters.

The user can automate complex LINK sessions involving multiple files by creating a response file. The *response_file* parameter must be the name of an ASCII file that corresponds line for line to the responses that are entered in a normal interactive LINK session. The last line of the response file must end with a semicolon character (;) or a carriage return. If all parameters required by LINK are not present in the response file and the response file does not end with a semicolon or carriage return, LINK prompts the user for the missing information.

LINK supports many options that can be invoked by including a switch in the command line, as part of the response to a LINK prompt, or in a response file. To simplify this description, these switches are grouped according to their functions.

The /E, /HE, /NOD, /NOI, /P, and /SE:*n* switches affect LINK's general operation. The /E switch compresses repetitive sequences of bytes in *exe_file* and optimizes certain parts of the relocation table in *exe_file*'s header. The /E switch functions exactly like the EXEPACK utility.

Note: The /E switch does not always save a significant amount of disk space and may even increase file size when used with small programs that have few load-time relocations or repeated characters. The Microsoft Symbolic Debugger (SYMDEB) utility cannot be used with packed files.

The /HE switch displays the available options on the screen. No other switches or filenames should be specified if the /HE switch is used. The /NOD switch causes LINK to ignore any default libraries that have been added to the object modules by the high-level-language compiler that produced the modules, thus restricting searches to those libraries specified in the command line or response file. The /NOI switch causes LINK to be case sensitive when resolving external references to symbols between object modules. The /NOI switch is typically used with object files created by high-level-language compilers that differentiate between uppercase and lowercase letters.

The /P switch causes LINK to pause and prompt the user before writing *exe_file* to disk, thus allowing the user to exchange the disk used during the linking operation for another that has more space available. The /SE:*n* switch controls the number of program segments processed by LINK. The *n* must be a decimal, octal, or hexadecimal number from 1 through 1024, inclusive (default = 128). Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x.

The /M and /LI switches affect the production and contents of the optional map file. The /M switch creates a map file with the same name as *exe_file* or, if *exe_file* is not specified, with the same name as the first object file and the extension .MAP. The resulting map file includes a list of all public symbols and their addresses. The /LI switch also creates a map file and includes line-number information if available in the object file. (MASM and some high-level-language compilers do not insert line-number information into object files.)

The /D, /DO, /NOG, and /O:*n* switches affect the structure of the code in *exe_file*. Use of the /D switch places the data in DGROUP at the top (highest address) of the memory segment pointed to by the DS register, rather than at the bottom (the default). The /DO switch arranges the program segments according to a convention expected by all Microsoft language compilers: All segments with the class name CODE are placed first in the executable file; any other segments that do not belong to DGROUP are placed immediately after the CODE segments; all segments belonging to DGROUP are placed at the end of the file. The /NOG switch causes LINK to ignore group associations specified in the object modules when assigning addresses to data and code items; that is, segments that would ordinarily have been collected into the same physical memory segment because of their association within a GROUP are decoupled. The /NOG switch provides compatibility with LINK versions 2.02 and earlier and with early versions of Microsoft language compilers. The /O:*n* switch controls the interrupt number used by the resident overlay manager if the linked program includes overlays. The number *n* can be any decimal, octal, or hexadecimal number in the range 0 through 255 (default = 63, or 3FH). Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x.

Note: MASM and many high-level-language compilers do not include overlay managers in their libraries. Users should check their compiler documentation to determine if the /O:*n* switch can be used.

Warning: Interrupt numbers that conflict with the software interrupts used to obtain MS-DOS or ROM BIOS services or with hardware interrupts assigned to peripheral device controllers should not be used in the /O:*n* switch.

The /C:*n*, /H, and /ST:*n* switches control the information in *exe_file*'s header that affects the behavior of the MS-DOS system loader when the file is read from the disk into RAM for execution. The /C:*n* switch sets the maximum number of 16-byte paragraphs of memory to be made available to the program when it is loaded into memory, in addition to the memory required to hold the program's code, data, and stacks; the default is 65535, which causes the program to be allocated all available memory. The /H switch causes the program to be loaded as high as possible in the transient program area (free memory), rather than as low as possible (the default). The /ST:*n* switch sets the stack size (in bytes) to be allocated for the program when it is loaded and overrides any stack segment size declarations in the original source code. The number *n* can be any decimal, octal, or hexadecimal number from 1 through 65535; however *n* must be large enough to accommodate any initialized data in the stack segment. Octal numbers must have a leading zero; hexadecimal numbers must begin with 0x. If the /ST:*n* switch is not used, LINK calculates a program's stack size, basing the size on the size of any stack segments given in the object files. The /C:*n* and /ST:*n* values in the *exe_file* header can be altered after linking by using the EXEMOD utility.

If LINK is unable to hold in RAM all the data it is processing, it creates a temporary disk file named VM.TMP (Virtual Memory) in the current directory of the default disk drive. If a floppy disk is in the default drive, LINK issues a warning message to prevent the user from changing disks until the LINK session is completed. After LINK finishes processing, it deletes the temporary file.

Warning: Any file named VM.TMP that is already on the disk will be destroyed if LINK creates the temporary disk file.

Return Codes

0 No errors or unresolved references were encountered during creation of *exe_file.*

1 A miscellaneous LINK error occurred that was not covered by the other return codes.

16 A data record was too large to process.

32 No object files were specified in the command line or response file.

33 The map file could not be created.

66 A COMMON area was declared that is larger than 65535 (one segment).

96 Too many libraries were specified.

144 An invalid object module (*obj_file*) was detected.

145 Too many TYPDEFs were found in the specified object modules.

146 Too many group, segment, or class names were found in one object module.

147 Too many segments were found in all the object modules combined, or too many segments were found in one object module.

148 Too many overlays were specified.

149 The size of a segment exceeded 65535.

150 Too many groups or GRPDEFs were found in one object module.

151 Too many external symbols were found in one object module.

177 The size of a group exceeded 65535.

Examples

The simplest use of LINK is to process a single object file to produce an executable file, using all the default values. For example, to process the file SHELL.OBJ, create an executable file named SHELL.EXE, and search only the default libraries, type

```
C>LINK SHELL;  <Enter>
```

The semicolon after the filename causes LINK to use the default values for all other parameters.

To link three object files named SHELL.OBJ, VIDEO.OBJ, and DOSINT.OBJ into an executable file named SHELL.EXE and search the library DEVLIB.LIB on drive B before searching any default libraries, type

```
C>LINK SHELL+VIDEO+DOSINT,,,B:DEVLIB  <Enter>
```

If the LINK command is entered without parameters, LINK prompts the user for the necessary information. For example, the following interactive session links the file

MENUMGR.OBJ into the executable file MENUMGR.EXE, creates a map file named MENUMGR.MAP, and searches the math floating-point emulator library EM.LIB before any default libraries:

```
C>LINK  <Enter>


Microsoft (R) 8086 Object Linker  Version 3.05
Copyright (C) Microsoft Corp 1983,1984,1985.  All rights reserved.

Object Modules [.OBJ]: MENUMGR  <Enter>
Run File [MENUMGR.EXE]:  <Enter>
List File [NUL.MAP]:  MENUMGR  <Enter>
Libraries [.LIB]:  EM  <Enter>
```

Messages

filename is not a valid library

The file specified as an object module library either is corrupt or is not a library in the format created by the Microsoft LIB utility.

About to generate .EXE file
Change diskette in drive *X* and press <ENTER>

The /P switch was used in the command line. LINK is prompting the user to change disks before LINK creates the file containing the executable program.

Ambiguous switch error: "*option*"

A valid switch was not entered after a forward slash (/) in the command line.

Array element size mismatch

A FAR communal array was declared with two or more different array-element sizes (for example, once as an array of characters and once as an array of real numbers). This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support FAR communal arrays; it does not occur with object files produced by MASM.

Attempt to access data outside segment bounds

A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which compiler or assembler produced the invalid object module and notify Microsoft Corporation.

Attempt to put segment *name* in more than one group in file *filename*

A segment was declared to be a member of two groups. Correct the source code and re-create the object modules.

Bad value for cparMaxAlloc

The value specified using the /C:*n* option is not in the range 1 through 65535.

Cannot create temporary file

The destination disk has insufficient space for the temporary file, or the root directory is full.

Cannot find file *filename*
Change diskette and press <ENTER>
The specified object file cannot be found in the current drive.

Cannot find library: *filename*
Enter new file spec:
The specified library file cannot be found or does not exist. Enter the correct drive letter, check the spelling of the filename and path, or make sure that the LIB environment variable has been set up properly.

Cannot nest response files
A response file was named within a response file. Revise the response file to eliminate the nested file.

Cannot open list file
The destination disk has insufficient space for the listing, or the root directory is full.

Cannot open response file: *filename*
LINK cannot find the specified response file.

Cannot open run file
The destination disk has insufficient space for the .EXE file, or the root directory is full.

Cannot open temporary file
The destination disk has insufficient space for the temporary file, or the root directory is full.

Cannot reopen list file
The original disk was not replaced when requested. Restart LINK.

Common area longer than 65536 bytes
The program has more than 64 KB of communal variables. This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support communal variables.

Data record too large
An LEDATA record (in an object module) contains more than 1024 bytes of data. This is a symptom of an error in the compiler used to generate the object module. Document the circumstances and contact Microsoft Corporation.

Dup record too large
An LIDATA record (in an object module) contains more than 512 bytes of data. This error may be caused by a complex structure definition or by a series of deeply nested DUP operators.

File not suitable for /EXEPACK, relink without
The file linked with the /E switch would have been smaller if it had not been compressed. Relink without the /E switch.

Fixup overflow near *number* in segment *name* in *filename* offset *number*
A group is larger than 64 KB, the original source file contains an intersegment short jump or intersegment short call, the name of a data item conflicts with that of a library subroutine, or an EXTRN declaration is placed inside the wrong segment.

Incorrect DOS version, use DOS 2.0 or later
LINK uses the extended file management calls to provide path support and, thus, does not work with versions of MS-DOS earlier than 2.0.

Insufficient stack space
Not enough memory is available to run LINK.

Interrupt number exceeds 255
The number specified in the /O:*n* switch is not in the range 0 through 255.

Invalid numeric switch specification
An incorrect value was entered with one of the LINK options.

Invalid object module
One of the object modules is invalid. Recompile the source file. If the error persists after recompiling, document the circumstances and contact Microsoft Corporation.

NEAR/HUGE conflict
Conflicting NEAR and HUGE definitions were given for a communal variable. This error occurs only with programs produced by the Microsoft C Compiler or other compilers that support communal variables.

Nested left parentheses
An opening (left) parenthesis is needed on the left side of an overlay module.

Nested right parentheses
A closing (right) parenthesis is needed on the right side of an overlay module.

No object modules specified
No object file names were specified in the command line or response file.

Object not found
One of the object files specified in the command line was not found.

Out of space on list file
The destination disk has insufficient space for the listing.

Out of space on run file
The destination disk has insufficient space for the .EXE file.

Out of space on scratch file
The disk in the default drive has insufficient space for temporary files.

Overlay manager symbol already defined: *name*
A symbol name was defined that conflicts with one of the special overlay manager names. Use another symbol name.

Please replace original diskette in drive *X* and press <ENTER>
The /P switch was specified in the command line and the disk to receive the .EXE file produced by LINK has already been inserted. This message indicates that the .EXE file was successfully created and that the original disk should again be placed in the drive.

Relocation table overflow
More than 32768 long calls, long jumps, or other long pointers were found in the program. The program may need to be restructured to reduce the number of FAR references. (Pascal and FORTRAN users should try turning off the debugging option before restructuring the program.)

Response line too long
A line in a response file had more than 127 characters.

Segment limit set too high
The number specified in the /SE:*n* switch was not in the range 1 through 1024.

Segment limit too high
Not enough memory is available for LINK to allocate tables to describe the number of segments requested (default = 128 or the number specified in the /SE:*n* switch). Use the /SE:*n* switch to specify a smaller number of segments, or alter the system configuration to increase the amount of free memory.

Segment size exceeds 64K
The program is a small-model program with more than 64 KB of code or data, a compact-model program with more than 64 KB of code, or a medium-model program with more than 64 KB of data. Selection of a different model or alteration of the program code may be required to successfully complete the LINK process.

Stack size exceeds 65536 bytes
The size specified for the stack in the /ST:*n* switch was too large, or the combined length of multiple declared stack segments exceeded 64 KB.

Symbol already defined: "*symbol*"
One of the special overlay symbols required for overlay support was previously defined.

Symbol defined more than once: "*symbol*" in file
A symbol has been defined more than once in the object module. Remove the extra symbol definition.

Symbol table overflow
The program has more than 256 KB of symbolic information (publics, externals, segments, groups, classes, files, and so on). Eliminate as many public symbols as possible, combine modules and/or segments, and recreate the object files.

Terminated by user
Ctrl-C or Ctrl-Break was pressed, causing the LINK session to be terminated prematurely.

Too many external symbols in one module
An object module contains more than the limit of 1023 external symbols.

Too many group-, segment-, and class-names in one module
One of the object modules for the program contains too many group, segment, and class names. The source file for the object module may need to be divided or restructured.

Too many groups
The program defines more than nine groups (including DGROUP). Groups must be combined or eliminated.

Too many GRPDEFs in one module
LINK encountered more than nine group definitions (GRPDEFs) in a single object module. Reduce the number of GRPDEFs or split the object module.

Too many libraries
More than 16 libraries were specified. Combine libraries or use object modules that require fewer libraries.

Too many overlays
The program defines more than 63 overlays. Reduce the number of overlays.

Too many segments
The program has more than the maximum number of segments as specified by the default of 128 or with the /SE:*n* switch. Use the /SE:*n* switch to specify a greater number of segments.

Too many segments in one module
An object module has more than 255 segments. Split the module or combine segments.

Too many TYPDEFs
An object module contains too many TYPDEF records (these records describe communal variables). This error occurs only with programs produced with the Microsoft C Compiler or other compilers that support communal variables.

Unexpected end-of-file on library
This message may indicate that the disk containing the library in use was removed prematurely.

Unexpected end-of-file on scratch file
The disk containing VM.TMP was removed.

Unmatched left parenthesis
A syntax error was detected in the specification of an overlay structure. Refer to the language compiler manual for instructions on specifying overlays to LINK.

Unmatched right parenthesis
A syntax error was detected in the specification of an overlay structure. Refer to the language compiler manual for instructions on specifying overlays to LINK.

Unrecognized switch error: "*option*"
An unrecognized character was entered after a forward slash (/) in the command line.

Unresolved COMDEF; Microsoft internal error
This is a serious problem. Note the circumstances of the failure and contact Microsoft Corporation.

Unresolved externals: *list*
A symbol was declared external (EXTRN) in one object module but was not declared PUBLIC in the object module in which it was defined, or a necessary library specification was omitted from the command line or response file.

VM.TMP is an illegal file name and has been ignored
VM.TMP was specified as an object file name. If an object file named VM.TMP exists, rename it.

Warning: load-high disables exepack
The /H and /E switches cannot be used at the same time.

Warning: no stack segment
The program contains no segment with the STACK combine type. This message can be ignored if there is a specific reason for not defining a stack (for example, if the .EXE file will subsequently be converted to a .COM file) or for defining one without the STACK combine type.

WARNING: Segment longer than reliable size
Although code segments can be as long as 65536 bytes, code segments longer than 65500 bytes can be unreliable on the Intel 80286 microprocessor. Reduce all code segments to 65500 bytes or less.

Warning: too many public symbols
The /M switch was used to request a sorted listing of public symbols in the map file, but there are too many symbols to sort. LINK will produce an unsorted listing instead.

MAKE

Maintain Programs

Purpose

Interprets a text file of commands to compare dates of files and carry out other operations on the basis of the comparison. MAKE is customarily used to update the executable version of a program after a change to one or more of its source files. The MAKE utility is supplied with the Microsoft Macro Assembler (MASM), C Compiler, and FORTRAN Compiler. This documentation describes MAKE version 4.05.

Syntax

MAKE [/D] [/I] [/N] [/S] [*name=value* . . .] *filename*

where:

filename is an ASCII text file that contains MAKE dependency statements, commands, macro definitions, and inference rules.

name=value declares a MAKE macro, associating a specific value with the dummy parameter *name*.

/D displays the last modification date of each file as it is scanned.

/I causes MAKE to ignore exit codes returned by programs called by *filename*.

/N displays but does not execute the commands in *filename*.

/S selects "silent" mode (commands are not displayed as they are executed).

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/). Versions of MAKE earlier than 4.0 have no switches.

Description

The MAKE utility allows maintenance of complex programs to be automated. Its basic operation is to compare the dates of files and to carry out, or not carry out, an associated list of commands on the basis of the comparison.

The *filename* parameter specifies an ASCII text file often referred to as a make file. By convention, *filename* is the same as the name of the executable program being maintained, but without an extension. A make file can contain the following types of entries:

- Dependency statements
- Commands
- Macro definitions
- Inference rules
- Comments

The basic form of a make file is a dependency statement followed by one or more valid MS-DOS command lines:

targetfile: *dependentfile1* [*dependentfile2*. . .]
    command1
    [*command2*]
    . . .

where *targetfile* designates the file that may need updating, *dependentfile* is a source file or files on which *targetfile* depends, and *command1*, *command2*, and so forth are any valid MS-DOS internal commands or external programs. These commands or programs are executed only if the date and time stamps of any dependent file are more recent than those of the target file or if the target file does not exist. Only one target file can be specified. Any number of dependent files can be included; each dependent filename must be separated from the next by at least one space. If too many dependent files are included to fit on a single line, the line can be terminated with a backslash character (\) and the list continued on the next line.

Any number of MS-DOS command lines can follow a dependency statement. The last command line should be followed by a blank line to set it off from the next MAKE entry. It is recommended that each command line include a leading space or tab character for compatibility with future versions of MAKE and existing versions of XENIX MAKE.

A macro definition takes the form

name=value

where both *name* and *value* are any string. Whenever *name* is referenced in the make file in the form $(*name*), *name* is replaced by the string *value* before the statement that contains it is evaluated or executed. Macro definitions can be nested, although very complex macro definitions can result in the premature termination of the MAKE process because of lack of memory. If *name* is not defined in the file but is defined in the system environment block by a previous SET command, $(*name*) is replaced by the string following the equal sign (=) in the environment block. If the command line contains a parameter of the form *name=value*, the command line overrides any definition of *name* in the make file or in the environment block. Thus, the precedence for macro definitions with the same *name* is

1. Command line
2. Make file
3. Environment block

MAKE contains several special macros that make it more convenient to form commands:

Macro	**Action**
$*	Substitutes as the base portion of *targetfile* (the filename without the extension).
$@	Substitutes as the complete *targetfile* name.
$**	Substitutes as the complete *dependentfile* list.

An inference rule specifies a series of commands to be carried out for a matching dependency statement that is not followed by its own list of commands. Inference rules allow a set of commands to be applied to more than one *targetfile: dependentfile* description, eliminating repetition of the same set of commands for several descriptions. An inference rule takes the form

```
.dependentextension.targetextension:
        command1
        [command2]
        ...
```

Whenever MAKE finds a dependency statement not followed by any commands, the utility first searches the make file for an inference rule. If MAKE doesn't find an inference rule in the make file, the utility then searches the current drive and directory (or any directories specified with the MS-DOS PATH command) for the tools initialization file (TOOLS.INI) and searches the *[make]* section of TOOLS.INI for an inference rule that matches the extensions of the target file and dependent files in the dependency statement.

A make file can contain any number of comment lines. If a comment is placed where MAKE expects to find a command, the comment must be on a separate line and must have the pound character (#) as the first character of the line. Elsewhere, a pound character and following comment text can be placed either on a line alone or after the last dependent file or command listed on a line. Characters appearing on a line after the pound character are ignored during execution.

The /D, /N, and /S switches affect MAKE's output to the display while MAKE is executing. The /D switch causes the last modification date of each file to be displayed as the file is scanned. The /N switch causes the commands in the make file to be expanded and displayed, but not executed; this is useful for determining the result of a specific MAKE process without first examining the file dates and without recompiling or relinking files. The /S switch selects "silent" mode, in which commands are not displayed as they are executed.

The /I switch causes MAKE to ignore error codes returned by the compilers, assemblers, linkers, or other programs called by the make file. When the /I switch is used, the MAKE process proceeds to completion regardless of errors instead of terminating immediately as it ordinarily would, but the resulting files may not be executable.

Return Codes

0	No error; the MAKE process was successful.
1	Processing was terminated because of a fatal error by MAKE or by one of the programs called by MAKE.

Example

Assume that the file SHELL contains the following MAKE dependency statements and commands:

```
video.obj: video.asm
        masm video;

shell.obj: shell.c
        msc shell;

shell.exe: shell.obj video.obj
        link /map shell+video,shell,shell,slibc2
```

The SHELL file asserts that the executable program SHELL.EXE is composed of the files SHELL.OBJ and VIDEO.OBJ, which are in turn compiled or assembled from the source files SHELL.C and VIDEO.ASM. To update the file SHELL.EXE if either of the source files for its constituent modules has been changed, type

```
C>MAKE SHELL  <Enter>
```

Messages

fatal error U1001: macro definition larger than 512
A single macro was defined to have a value string longer than the 512-byte maximum. Rewrite the make file to use two or more short lines instead of one long line.

fatal error U1002: infinitely recursive macro
The macros defined in the make file form a circular chain.

fatal error U1003: out of memory
The make file cannot be processed because insuffcent memory is available in the transient program area. Split the make file into two make files or reconfigure the system to increase available memory.

fatal error U1004: syntax error : macro name missing
A macro name is missing from the left side of the equal sign (=).

fatal error U1005: syntax error : colon missing
A line that should be a dependency statement lacks the colon that separates a target file from its dependent files. MAKE expects any line that follows a blank line to be a dependency statement.

fatal error U1006: *targetname* : macro expansion larger than 512
A single macro expansion, plus the length of any string to which it may be concatenated, is longer than 512 bytes. Rewrite the make file to use two or more short lines instead of one long line.

fatal error U1007: multiple sources
An inference rule has been defined more than once in the make file.

fatal error U1008: *filename* : cannot find file
The specified file does not exist.

fatal error U1009: *command* : argument list too long
A command line in the make file is longer than 128 characters (the maximum MS-DOS allows).

fatal error U1010: *filename* : permission denied
The specified file is read-only.

fatal error U1011: not enough memory
Memory is insufficient in the transient program area to execute a program listed in the make file. Reconfigure the system to increase available memory, if necessary.

fatal error U1012: *filename* : unknown error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

fatal error U1013: *command* : error *returncode*
One of the programs or commands called by MAKE was not able to execute correctly. MAKE terminates and displays the error code from the program that failed.

warning U4000: *filename* : target does not exist
The target file does not already exist. The dependency statement is evaluated as though the target file exists and has a date earlier than that of any of the dependent files.

warning U4001: dependent *filename* does not exist;
target *filename* not built
One of the dependent files does not exist or could not be found, so MAKE terminated without creating a new target file.

warning U4013: *command* : error *returncode* (ignored)
One of the programs or commands called by MAKE did not execute successfully and has returned the specified return code. Because MAKE was run with the /I switch, MAKE ignores the error and continues processing the make file.

warning U4014: usage : make [/n] [/d] [/i] [/s] [name=value ...] file
An error was detected in the MAKE command line.

MAPSYM

Create Symbol File for SYMDEB

Purpose

Processes a map file generated by the Microsoft Object Linker (LINK) to create a special symbol file for use with SYMDEB, the symbolic debugging program. The MAPSYM utility is supplied with the Microsoft Macro Assembler (MASM). This documentation describes MAPSYM version 4.0.

Syntax

MAPSYM [/L] *map_file*

where:

map_file	is a map file produced by LINK (default extension = .MAP).
/L	causes information about the symbol file to be displayed as it is created.

Note: The /L switch can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

LINK combines relocatable object records (produced by MASM or a high-level-language compiler) into an executable program, which is stored in a specially formatted file with a .EXE extension. LINK can also produce an optional map file that contains information about public symbols and addresses in the linked program. The map file is an ordinary ASCII text file and has a default extension of .MAP.

To create a map file to use with MAPSYM, the LINK command line should include the /MAP switch, which creates the file, and the /LINENUMBERS switch, which includes line numbers. *See* PROGRAMMING UTILITIES: LINK.

The MAPSYM utility processes a map file into a special symbol file that can be used by SYMDEB. A drive and pathname can be specified if the map file is not in the current directory. If a file extension is not specified, .MAP is assumed.

The symbol file created by MAPSYM is placed in the current directory and has the same name as the map file but has the extension .SYM. It can contain a maximum of 1024 segments (or as many segments as can fit into available memory) and 10,000 symbols per segment. *See* PROGRAMMING UTILITIES: SYMDEB.

When the /L switch precedes *map_file* in the command line, MAPSYM displays the names of groups defined in the program described by the map and symbol files, plus the program's starting address. The /L switch does not affect the format of the symbol file that is generated.

Return Codes

0 No error; the MAPSYM process was successful.

1 Processing was terminated because of a write failure, because the map file specified does not exist, or because the symbol file could not be created.

4 Processing was terminated because an unexpected end-of-file mark was detected, because too many segments exist in the map file, because no public symbols exist in the map file, or because not enough memory is available to create the symbol file.

Example

To convert the file HELLO.MAP, which was produced by assembling and linking the file HELLO.ASM, to a symbol file that can be used by SYMDEB, type

```
C>MAPSYM /L HELLO  <Enter>
```

MAPSYM displays the following:

```
Microsoft (R) Symbol File Generator  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985.  All rights reserved.
Building: HELLO.SYM
HELLO.MAP
        Program entry point at 0000:0100
HELLO     0 segment
```

The symbol file produced by MAPSYM symbol has the name HELLO.SYM.

Messages

Can't create: <*filename*>
The drive specified does not exist, the current disk or directory is full, or the output file already exists and is read-only.

Can't open MAP file: <*filename*>
The file named in the command line does not exist.

DOS 2.0 or later required
MAPSYM does not work with versions of MS-DOS earlier than 2.0.

mapsym: out of memory
System memory is insufficient to process the map file.

mapsym: segment table (*n*) exceeded.
More than 1024 segments have been used in the map file. The number displayed is the total number of segments in the map file.

No public symbols
Re-link file with the /M switch!
The map file created by LINK does not include a list of public names. The .EXE file must be relinked using the /MAP switch to generate a map file that can be used with MAPSYM.

Unexpected eof reading: <*filename*>
The map file contains no symbols, is corrupt, or is otherwise invalid. The .EXE file must be relinked and a new map file generated.

usage: MAPSYM [/l] maplist
A syntax error was detected in the command line.

Write fail on: <*filename*>
An error occurred during the creation of the output file.

MASM

Microsoft Macro Assembler

Purpose

Translates an assembly-language source program into a relocatable object module. MASM is part of the Microsoft Macro Assembler (MASM) retail package. This documentation describes MASM version 4.0.

Syntax

MASM

or

MASM *source_file* [,[*object_file*][,[*list_file*][,[*cref_file*]]]] [*options*] [;]

where:

source_file is the name of the file containing the assembly-language source code (default extension = .ASM).

object_file is the name of the file to receive the assembled object module (default extension = .OBJ).

list_file is the name of the file or device to receive the assembly listing (default = NUL). (If destination = file, default extension = .LST.)

cref_file is the name of the cross-reference file to receive information for later processing by the CREF utility (default = NUL). (If destination = file, default extension = .CRF.)

options is one or more switches from the list below.

/A Writes the program segments in alphabetic order.

/B*n* Sets the size of the source-file buffer in kilobytes (1–63, default = 32).

/C Creates a cross-reference (.CRF) file.

/D Adds a first-pass program listing to *list_file* if a list file was specified (default = second-pass listing only).

/D*symbol* Defines *symbol* as a null text string.

/E Assembles code for an 8087/80287 emulator.

/I*path* Defines a directory to be searched for *include* files.

/L Creates a list (.LST) file with line-number information.

/ML Preserves case sensitivity in all symbol names.

(more)

/MU Converts all lowercase names to uppercase names.

/MX Preserves lowercase in public and external names only.

/N Suppresses generation of tables of macros, structures, records, groups, segments, and symbols at the end of the list file.

/P Checks for impure code in 80286 protected mode; has no effect unless the .286P directive is included in the source file.

/R Assembles code for an 8087/80287 math coprocessor.

/S Arranges program segments in order of occurrence.

/T Selects terse mode, suppressing all messages generated during assembly except error messages.

/V Selects verbose mode, displaying the number of lines and symbols at the end of assembly.

/X Includes false conditionals in the list file.

/Z Displays source lines with errors during assembly.

Note: Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

Description

MASM translates assembly-language source code into relocatable object modules. The object modules can then be placed in a library file or processed by the Microsoft Object Linker (LINK) to create an executable program.

The *source_file* parameter is the only required filename. It specifies a file containing the assembly-language source code in ASCII text. If no extension is specified, MASM uses .ASM. If no source file is entered in the command line, MASM prompts for a source file name.

The *object_file* parameter specifies the file that will contain the assembled relocatable object code. If this parameter is not supplied, MASM uses the same filename as *source_file* but substitutes the extension .OBJ.

The *list_file* parameter specifies a destination file or device for the optional program listing. The listing contains the original source code, the assembled machine code, macro definitions and expansions, and other useful information, formatted into pages with titles, dates, and page numbers. If the destination of the listing is a file, the file's default extension is .LST. If the *list_file* parameter is not included in the command line, MASM sends the listing to NUL (that is, a listing is not produced).

The *cref_file* parameter specifies the name of a cross-reference file to receive information to be processed by the CREF utility. If a file extension is not specified, MASM uses .CRF. If the *cref_file* parameter is not included in the command line, MASM sends the file to NUL (that is, no cross-reference file is generated).

If the MASM command is entered without parameters, MASM prompts the user for each filename. The default response for each prompt (except the source file prompt) is displayed in square brackets and can be selected by pressing the Enter key.

After the source file is specified, if MASM encounters a semicolon character (;) in the command line or at any prompt, it uses default values for the remaining parameters. MASM ignores any parameters specified after the semicolon.

MASM does two passes to translate the assembly-language code in the source file into relocatable object code. Any errors detected during translation are displayed on standard output and included in the program listing (if one is requested). Two types of errors may be detected: warning errors and severe errors. If MASM encounters a warning error, it still creates the object file, although the resulting file may be unusable. If MASM encounters a severe error, it does not create the object file. After a file has been successfully assembled without errors, the LINK utility can be used to convert the resulting object file into an executable program file.

MASM supports a wide variety of options that can be selected by including switches in the command line or by responding to any prompt.

The /A and /S switches determine the order of segments in the resulting object module file. The /A switch places the segments into the object file in alphabetic order. The /S switch (the default) arranges the segments in the same order they occur in the source file.

The /B*n*, /D*symbol*, and /I*path* switches have rather general effects on the behavior of MASM. The /B*n* switch sets the size (in kilobytes) of the source file's RAM buffer; the value of *n* must be between 1 and 63, inclusive (default = 32). If the RAM buffer is large enough, the entire source file can be kept resident in memory, reducing disk activity during passes. The /D*symbol* switch defines a null text-string symbol from the command line. This symbol can be referenced inside the program with the IFDEF directive to control the conditional assembly of portions of the program. The /I*path* switch specifies a directory that will be searched for files named in assembler INCLUDE statements if those statements do not include an explicit directory. As many as 10 such search paths can be specified with individual /I*path* switches.

The /E and /R switches affect the generation of code for the 8087/80287 emulator or 8087/80287 math coprocessor. (Support for the 80287 is included with MASM versions 3.0 and later.) The /E switch generates software interrupts to floating-point-processor emulator routines. A subprogram assembled with the /E switch can be linked to C, Pascal, and FORTRAN programs and can use the emulator libraries. The /R switch produces in-line machine instructions for the math coprocessor when floating-point mnemonics are used.

The /ML, /MU, and /MX switches control MASM's handling of uppercase and lowercase names. The /ML switch makes MASM case sensitive; that is, it makes MASM differentiate a name in uppercase letters from the same name in lowercase letters. (The /ML switch should not be used if the source file contains 8087 WAIT instructions and MASM 4.0 is being used to translate the file.) The /MU switch (the default) makes MASM case insensitive; all lowercase letters are converted to uppercase for purposes of assembly. The /MX switch makes MASM case sensitive for public and external names only (names defined with PUBLIC or EXTRN directives). The /MX switch is often used to process assembly-language functions for C programs.

The /P switch checks for impure code segments that will cause problems if the assembled program is run in 80286 protected mode. The switch checks by flagging any instruction that will change a memory location addressed through the processor's CS register. The /P switch has no effect unless the assembly-language source file includes the .286P directive.

The /C, /D, /L, /N, and /X switches control the contents of the program listing and other optional files that are generated as a result of assembly. The /C switch causes the creation of a cross-reference (.CRF) file and the addition of line numbers to the list (.LST) file (if one exists). The /C switch should be included in the command line if the cross-reference file will be used later with the CREF utility to produce a cross-reference listing. The /D switch includes a listing from the first pass as well as a listing from the second pass in the list file if a list file was specified (default = second-pass listing only). By comparing the two listings, the user can isolate an instruction causing a phase error. (A phase error occurs when MASM makes assumptions about addresses, values, or data types on the first pass that are not valid in the second pass.) The /L switch creates a list file with line-number information and gives it the same name as the source file, with the extension .LST. The /N switch suppresses generation of tables — symbols, segments, groups, structures, records, and macros — at the end of a program listing. The /X switch includes statements inside false conditional statements in the list file, allowing conditionals that do not generate code to be displayed. /X has no effect if the .SFCOND or the .LFCOND directive is used in the source file; if the .TFCOND directive is used, the effects of /X are reversed.

Note: The effects of /X are also reversed in MASM version 1.2. In that version, statements within a false conditional are included in the list file by default, and /X will suppress them.

The /T, /V, and /Z switches affect MASM's display on standard output. The /T (terse) switch suppresses messages to standard output, except for messages indicating warning errors or severe errors. The /V (verbose) switch displays information about the number of source lines and symbols at the end of the assembly, in addition to displaying the normal error and symbol space information. The /Z switch displays the actual source lines producing assembly errors (rather than displaying just the error type and line number).

Note: Versions of MASM earlier than 4.0 always show both the source line and the error message.

Return Codes

0 No errors were found during assembly.
1 An error was detected in one of the command-line parameters.
2 The assembly-language source file could not be opened.
3 The list file could not be created.
4 The object file could not be created.
5 The cross-reference file could not be created.
6 An *include* file could not be opened.
7 At least one severe error was detected during assembly. (MASM deletes the invalid object file.)
8 The assembly was terminated because a memory allocation error occurred.
10 An error occurred in defining a symbol (with the /D*symbol* switch) from the command line.
11 Assembly was interrupted by the user's pressing Ctrl-C or Ctrl-Break.

Examples

To assemble the source file CLEAN.ASM in the current drive and directory and place the resulting relocatable object module in the file CLEAN.OBJ without producing a listing or a cross-reference file, type

```
C>MASM CLEAN;  <Enter>
```

The semicolon after the first parameter causes MASM to use the default values for the rest of the parameters.

To assemble the source file CLEAN.ASM, put the object code in a file named CLEAN.OBJ, create a list file named CLEAN.LST, and place information for later processing by the CREF utility in the cross-reference file CLEAN.CRF, type

```
C>MASM CLEAN,CLEAN,CLEAN,CLEAN  <Enter>
```

or

```
C>MASM CLEAN,,CLEAN,CLEAN  <Enter>
```

To use MASM interactively, enter its name without parameters:

```
C>MASM  <Enter>
```

MASM then prompts for all the necessary information. For example, the interactive session on the next page assembles the file HELLO.ASM into the file HELLO.OBJ, producing no listing or .CRF file.

```
C>MASM  <Enter>
Microsoft (R) Macro Assembler  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights reserved.

Source filename: [.ASM]: HELLO  <Enter>
Object filename: [HELLO.OBJ]:  <Enter>
Source listing [NUL.LST]:  <Enter>
Cross-reference [NUL.CRF]:  <Enter>

  51004 Bytes symbol space free

    0 Warning Errors
    0 Severe Errors
```

Messages

8087 opcode can't be emulated
An 8087 opcode or the operands used with it produced an instruction the emulator cannot support.

Already defined locally
An attempt was made to define a symbol as EXTRN that had already been defined locally.

Already had ELSE clause
An attempt was made to define an ELSE clause within an existing ELSE clause. (ELSE cannot be nested without nesting IF... ENDIF.)

Already have base register
More than one base register was specified within an operand.

Already have index register
More than one index register was specified within an operand.

Block nesting error
A segment, structure, macro, IRC, IRP, REPT, or nested procedure was not terminated properly.

Byte register is illegal
A byte register was used incorrectly in an instruction.

Can't override ES segment
An attempt was made to override the ES segment in an instruction in which this override is invalid.

Can't reach with segment reg
No ASSUME directive was given to make the variable reachable.

Can't use EVEN on BYTE segment
An EVEN directive was used on a segment declared to be a byte segment.

Circular chain of EQU aliases
An alias EQU ultimately points to itself.

Constant was expected
A constant was expected, but an item was received that does not evaluate to a constant.

CS register illegal usage
The CS register was used incorrectly in one of the instructions.

Data emitted with no segment
Code that is not located within a segment attempted to generate data.

Directive illegal in STRUC
All statements within STRUC blocks must be either comments preceded by a semicolon character (;) or one of the define directives (DB, DW, and so on).

Division by 0 or overflow
An expression was encountered that resulted in either a division by 0 or a number too large to be represented.

DUP is too large for linker
Nesting of DUP operators was such that a record too large for LINK was created.

End of file, no END directive
No END statement was encountered, or a nesting error occurred.

Extra characters on line
Superfluous characters were detected on a line after sufficient information to define an instruction was interpreted.

extra file name ignored
The command line contained more than four filename parameters.

Field cannot be overridden
An attempt was made to give a value to a field that cannot be overridden with a STRUC initialization statement.

Forced error
An error was forced with the .ERR directive.

Forced error - expression equals 0
An error was forced with the .ERRE directive.

Forced error - expression not equal 0
An error was forced with the .ERRNZ directive.

Forced error - pass1
An error was forced with the .ERR1 directive.

Forced error - pass2
An error was forced with the .ERR2 directive.

Forced error - string blank
An error was forced with the .ERRB directive.

Forced error - string not blank
An error was forced with the .ERRNB directive.

Forced error - strings different
An error was forced with the .ERRDIF directive.

Forced error - strings identical
An error was forced with the .ERRIDN directive.

Forced error - symbol defined
An error was forced with the .ERRDEF directive.

Forced error - symbol not defined
An error was forced with the .ERRNDEF directive.

Forward reference is illegal
An item was referenced in the operand of an EQU or equal-sign (=) directive before it was defined.

Illegal register value
A specified register value does not fit into the *reg* field (that is, the value is greater than 7).

Illegal size for item
The size of the referenced item is invalid. This error also frequently occurs when an attempt is made to assemble source code written for assemblers with less strict type-checking than that of the Microsoft Macro Assembler (such as early versions of the IBM assembler). The problem can usually be solved by overriding the type of the operand with the PTR operator.

Illegal use of external
A variable that was declared external was used incorrectly.

Illegal use of register
An attempt was made to use a register with an instruction in which a register cannot be used.

Illegal value for DUP count
The DUP count was not a constant that evaluates to a positive integer greater than zero.

Improper operand type
An operand was used in a way that prevents opcode generation.

Improper use of segment register
An attempt was made to use a segment register in an instruction in which use of a segment register is not permitted.

Impure memory reference
An attempt was made to store data in the code segment when the .286P directive and the /P switch were in effect.

Index displ. must be constant
An index displacement was used incorrectly or did not evaluate to an absolute number or memory address.

Internal error
An internal logic error was detected in the assembler. Document the circumstances and contact Microsoft Corporation.

Label can't have seg. override
A segment override was used incorrectly.

Left operand must have segment
The content of the right operand requires that a segment be specified in the left operand.

Line too long expanding *symbol*
A symbol defined by an EQU or equal-sign (=) directive is so long that expanding it will cause the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

Missing data; zero assumed
An operand is missing from a statement and MASM assumes its value is zero. This is a warning error; the object file is not deleted as it is with severe errors.

More values than defined with
Too many initial values were given when defining a variable using a REC or STRUC type.

Must be associated with code
A data-related item was used where a code-related item was expected.

Must be associated with data
A code-related item was used where a data-related item was expected.

Must be AX or AL
A register other than AX or AL was specified where only these are acceptable.

Must be in segment block
An attempt was made to generate code by instructions that were not contained within a segment.

Must be index or base register
An instruction requires a base or index register, and some other register was specified within square brackets ([]).

Must be record field name
A record field name was expected, but something else was encountered.

Must be record or fieldname
A record name or field name was expected, but something else was encountered.

Must be register
A register was expected as the operand, but something else was encountered.

Must be segment or group
A segment or group was expected, but something else was encountered.

Must be structure field name
A structure field name was expected, but something else was encountered.

Must be symbol type
A BYTE, WORD, DWORD, or similar designation was expected, but something else was encountered.

Must be var, label or constant
A variable, label, or constant was expected, but something else was encountered.

Must have opcode after prefix
A REP, REPE, REPNE, REPZ, or REPNZ instruction was not followed by the mnemonic for a string operation.

Near JMP/CALL to different CS
An attempt was made to do a NEAR jump or call to a location in a code segment defined with a different ASSUME:CS.

No immediate mode
Immediate data was supplied as an operand for an instruction that cannot use immediate data. For example, immediate data cannot be moved directly with a MOV instruction to a segment register; it must first be moved into a general register and then copied to the segment register.

No or unreachable CS
An attempt was made to jump to a label that is unreachable.

Normal type operand expected
A STRUC, BYTE, WORD, or some other invalid operand was encountered when a variable label was expected.

Not in conditional block
An ENDIF or ELSE statement was encountered, and no previous conditional-assembly directive was active.

Not proper align/combine type
The SEGMENT parameters are incorrect. Check the align and combine types to be sure they are valid.

One operand must be const
The addition operator was used incorrectly.

Only initialize list legal
An attempt was made to use a STRUC name without angle brackets (<>).

Operand combination illegal
A two-operand instruction was specified and the combination specified was invalid.

Operand must have segment
A SEG directive was used incorrectly.

Operand must have size
An operand was encountered that needed a specified size, but none had been provided. Often this error can be remedied by using the PTR operator to specify a size type.

Operand not in IP segment
An operand cannot be accessed because it is not in the segment last assigned to CS with an ASSUME directive.

Operand types must match
MASM encountered different kinds or sizes of arguments in a case where they must match.

Operand was expected
MASM expected an operand, but an operator was encountered.

Operands must be same or 1 abs
The subtraction operator was used incorrectly.

Operator was expected
MASM expected an operator, but an operand was encountered.

Out of memory
System memory is insufficient to complete the assembly. If a listing (.LST) or cross-reference (.CRF) file was being generated, retry the assembly, generating only an object file. It may also be necessary to modify the source program to reduce the load on the symbol table (by shortening names or reducing the number of EQU statements or macros, for example).

Override is of wrong type
An attempt was made to use a data item of incorrect size in a STRUC initialization statement.

Override value is wrong length
The override value for a structure field is too large to fit in the field.

Override with DUP is illegal
An attempt was made to use DUP to override in a STRUC initialization statement.

Phase error between passes
The program has ambiguous instruction directives that caused the location of a label in the program to change in value between the first and second passes of MASM. A common cause is a forward reference to a typed data item in the instructions preceding the label that generated the phase error message. Use the /D switch to produce a first-pass listing to aid in resolving phase errors between passes.

Redefinition of symbol
This message is displayed during first pass upon the second declaration of a symbol that has been defined in more than one place.

Reference to mult defined
The instruction references a symbol that has been defined more than once.

Register already defined
An internal error was detected. Note the circumstances of the failure and contact Microsoft Corporation.

Relative jump out of range
A conditional jump references a label that is out of the allowed range of –128 to +127 bytes relative to the current instruction. The problem usually can be corrected by reversing the condition of the jump and using an unconditional jump (JMP) to the out-of-range label.

Segment parameters are changed
The list of parameters encountered for a SEGMENT was not identical to the list specified the first time the segment was used.

Shift count is negative
A shift expression was generated that resulted in a negative shift count.

Should have been group name
A group name was expected, but something else was encountered.

Symbol already different kind
An attempt was made to redefine an already defined symbol.

Symbol has no segment
An attempt was made to use a variable with SEG that has no known segment.

Symbol is already external
An attempt was made to redefine a symbol as local that has already been defined as external.

Symbol is multi-defined
This message is displayed during the second pass upon each declaration of a symbol that has been defined in more than one place.

Symbol is reserved word
An attempt was made to use a reserved MASM word as a symbol.

Symbol not defined
A symbol that had not been defined was used.

Symbol type usage illegal
A PUBLIC symbol was used incorrectly.

Syntax error
The syntax of the statement does not match any recognizable syntax.

Type illegal in context
The type specified is of an unacceptable size.

Unable to open input file *filename*
The specified source file cannot be found.

unknown switch *letter*
The command line included an invalid switch.

Unknown symbol type
MASM does not recognize the size type specified in a label or external declaration. Rewrite with a valid type such as BYTE, WORD, or NEAR.

Value is out of range
A value is too large for its expected use.

Wrong type of register
A directive or instruction expected one type of register, but another type was encountered.

DEBUG

Program Debugger

Purpose

Allows the controlled execution of a program for debugging purposes or the alteration of the binary contents of any file. The DEBUG utility is supplied with the MS-DOS distribution disks.

Syntax

DEBUG

or

DEBUG *filename* [*parameter*...]

where:

filename is the name of the file that contains data to be modified or a program to be debugged. If *filename* includes an extension, it must be specified.

parameter... is one or more filenames or switches required by a program being debugged.

Description

The DEBUG program allows a file to be loaded, examined, and altered. If the file is not a .EXE file or a .HEX file, it may also be written back to disk. If the file contains a program, the program can be disassembled, modified, traced one instruction at a time, or executed at full speed with preset breakpoints. DEBUG can also be used to read from and write to input/output (I/O) ports and to read, modify, and write absolute disk sectors.

The command line typically includes the *filename* parameter, which is the name of an executable program (with the extension .COM or .EXE) to be loaded into DEBUG's memory buffer. Files with the extension .EXE are loaded in a manner compatible with the MS-DOS loader; if necessary, the contents of the file are relocated so that the program is ready to execute. Files with the extension .HEX are converted to binary images and loaded at the internally specified address. All other files are assumed to be direct memory images and are read directly into memory starting at offset 100H.

An appropriate program segment prefix (PSP) is synthesized at the head of DEBUG's buffer for use by the target program (the program being debugged). The PSP includes a command tail at offset 80H and default file control blocks (FCBs) at offsets 5CH and 6CH, constructed from the optional parameters following *filename*.

After DEBUG is loaded and the first file named in the command line is also located and loaded, DEBUG displays its special prompt character, a hyphen (-), and awaits a command. DEBUG commands consist of a single letter, usually followed by one or more parameters. Uppercase and lowercase characters are treated the same except when they are contained in strings enclosed within single or double quotation marks. All commands are executed by pressing the Enter key.

The DEBUG commands are

Command	Action
A	Assemble machine instructions (versions 2.0 and later).
C	Compare memory areas.
D	Display memory.
E	Enter data.
F	Fill memory.
G	Go execute program.
H	Perform hexadecimal arithmetic.
I	Input from port.
L	Load file or sectors.
M	Move (copy) data.
N	Name file or command-tail parameters.
O	Output to port.
P	Proceed through loop or subroutine (versions 3.0 and later).
Q	Quit debugger.
R	Display or modify registers.
S	Search memory.
T	Trace program execution.
U	Disassemble (unassemble) program.
W	Write file or sectors.

The parameters for a DEBUG command include addresses, ranges, 8-bit or 16-bit hexadecimal values, and lists. Multiple parameters can be separated by spaces, tabs, or commas, but separators are *required* only between hexadecimal values.

An address can be a simple offset or a complete address in the form *segment:offset*. The offset is always a hexadecimal number in the range 00H through FFFFH; the segment can be either a hexadecimal value in the same range or a two-character segment register name (CS, DS, ES, or SS). If the segment portion of an address is absent, DEBUG uses DS unless an A, G, L, T, U, or W command is used, in which case DEBUG uses CS.

A range specifies an area of memory and can be expressed as either two addresses or a starting address and a length. A segment can be included only in the first element of a range; an error message is displayed if a segment is found in the second address. A length is represented by the letter L, followed by a hexadecimal value between 00H and FFFFH that indicates the number of bytes following the starting address that the command should operate on.

Note: Any length that causes an address to exceed 16 bits will generate an error.

A byte, or 8-bit, value is entered as one or two hexadecimal digits, whereas a word, or 16-bit, value is entered as one to four hexadecimal digits. Leading zeros can be omitted.

A list is composed of one or more byte values or strings, separated by spaces, commas, or tabs. A string is one or more ASCII characters enclosed within single or double quotation marks. Case is significant within a string. If the same type of quote character that is used to delimit the string occurs inside the string itself, the character must be doubled inside the string in order to be interpreted correctly. For example:

```
"This ""string"" is OK."
```

When used, a list must be the last parameter in the command line.

DEBUG responds to an invalid command by pointing to the approximate location of the error with a caret character (^) and displaying the word *Error.* For example:

```
-D CS:0100,CS:0200  <Enter>
              ^ Error
```

DEBUG maintains a set of virtual CPU registers for a program being debugged. These registers can be examined and modified with DEBUG commands. When a program is first loaded for debugging, the virtual registers are initialized with the following values:

Register	.COM Program	.EXE Program
AX	Valid drive error code	Valid drive error code
BX	Upper half of program size	Upper half of program size
CX	Lower half of program size	Lower half of program size
DX	Zero	Zero
SI	Zero	Zero
DI	Zero	Zero
BP	Zero	Zero
SP	FFFEH or top of available memory minus 2	Size of stack segment
IP	100H	Offset of entry point within target program's code segment
CS	PSP	Base of target program's code segment
DS	PSP	PSP
ES	PSP	PSP
SS	PSP	Base of target program's stack segment

Note: DEBUG checks the first three parameters in the command line. If the second and third parameters are filenames, DEBUG checks any drive specifications with those filenames to verify that they designate valid drives. Register AX contains one of the following codes:

Code	Meaning
0000H	The drives specified with the second and third filenames are both valid, or only one filename was specified in the command line.
00FFH	The drive specified with the second filename is invalid.
FF00H	The drive specified with the third filename is invalid.
FFFFH	The drives specified with the second and third filenames are both invalid.

DEBUG also maintains a set of virtual flags, which may be set or cleared. The flags are

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

Before DEBUG transfers control to the target program, it saves the actual CPU registers and then loads them with the current values of the virtual registers. Conversely, when control reverts to DEBUG from the target program, the returned register contents are stored back in the virtual register set for inspection and alteration by the user.

Examples

To load the file SHELL.EXE in the current directory for execution under the control of DEBUG, type

```
C>DEBUG SHELL.EXE  <Enter>
```

To use the DEBUG program to inspect or modify memory or to read, modify, and write absolute disk sectors, simply type

```
C>DEBUG  <Enter>
```

Message

File not found
The filename supplied as the first parameter in the DEBUG command line cannot be found.

DEBUG: A

Assemble Machine Instructions

Purpose

Allows entry of assembler mnemonics and translates them into executable machine code.

Syntax

A [*address*]

where:

address is the starting location for the assembled machine code.

Description

The Assemble Machine Instructions (A) command accepts assembly-language statements, rather than hexadecimal values, for the Intel 8086/8088 microprocessors and the Intel 8087 math coprocessor and then assembles each statement into executable machine code.

The *address* parameter specifies the location where entry of assembly-language mnemonics will begin. If *address* is omitted, DEBUG uses the address following the last instruction generated the last time the A command was used. If the A command has not been used, DEBUG uses the current value of the target program's CS:IP registers.

After an A command is entered, DEBUG prompts for each assembly-language statement by displaying the address, in the form of a segment and an offset, in which the assembled code will be stored. When the Enter key is pressed, the assembly-language statement is translated, and each byte of the resulting machine instruction is stored sequentially in memory (overwriting existing information), beginning at the displayed address. The address following the last byte of the machine instruction is then displayed so that the user can enter the next assembly-language statement. Pressing the Enter key alone in response to the address prompt terminates the A command.

The syntax of assembly-language statements accepted by the DEBUG A command differs slightly from that of the usual Microsoft Macro Assembler programming statements. The differences can be summarized as follows:

- All numbers are assumed to be hexadecimal integers and should be entered without a trailing H character.
- Segment overrides must be specified by preceding the entire instruction with CS:, DS:, ES:, or SS:.
- File control directives (NAME, PAGE, TITLE, and so forth), macro definitions, record structures, and conditional assembly directives are not supported by DEBUG.
- Specific hexadecimal values, rather than program labels, must be included.

- When the data type (word or byte) is not implicit in the instruction, the type must be specified by preceding the operand with BYTE PTR (or BY) or WORD PTR (or WO).
- The size of the string in a string operation must be specified by adding a B (byte) or W (word) to the string instruction mnemonic (for example, LODSB or LODSW).
- The DB and DW instructions accept a parameter of the type *list* and assemble byte and word values directly.
- The WAIT or FWAIT opcodes for 8087 assembler statements are not generated by default, so they must be coded explicitly.
- Memory locations are differentiated from immediate operands by enclosing memory addresses in square brackets.
- Repeat prefixes, such as REP, REPZ, or REPNZ, can be entered either alone on the line preceding the statement they affect or immediately preceding the statement on the same line.
- Although the assembler generates the optimal form (SHORT, NEAR, or FAR) for jumps or calls, depending on the destination address, these designations can be overridden by preceding the operand with a NEAR (or NE) or FAR (no abbreviation) prefix.
- The mnemonic for a FAR RETURN is RETF.

Examples

To begin assembling code at address CS:0100H, type

```
-A 100  <Enter>
```

To assemble the instruction sequence

```
LODS  WORD PTR [SI]
XCHG  BX,AX
JMP   [BX]
```

beginning at address CS:0100H, the following dialogue would take place:

```
-A 100  <Enter>
1983:0100  LODSW  <Enter>
1983:0101  XCHG BX,AX  <Enter>
1983:0103  JMP [BX]  <Enter>
1983:0105  <Enter>
```

To continue assembling at the location following the last instruction generated by a previous A command, type

```
-A  <Enter>
```

DEBUG: C

Compare Memory Areas

Purpose

Compares two areas of memory and reports any differences.

Syntax

C *range address*

where:

range	is the starting and ending addresses or the starting address and length of the first area of memory to be compared.
address	is the starting address of the second area of memory to be compared.

Description

The Compare Memory Areas (C) command compares the contents of two areas of memory. The location and contents of any differing bytes are displayed in the following format:

address1 byte1 byte2 address2

If no differences are found, the DEBUG prompt returns.

The *range* parameter specifies the starting and ending addresses or the starting address and length in bytes of the first area of memory to be compared. The *address* parameter specifies the beginning address of the second area of memory to be compared. If a segment is not included in *range* or *address*, DEBUG uses DS.

Example

To compare the 64 bytes beginning at CS:CE00H with the 64 bytes beginning at CS:CF0AH, type

```
-C CS:CE00 CE3F CS:CF0A  <Enter>
```

or

```
-C CS:CE00 L40 CS:CF0A  <Enter>
```

If any differences are found, DEBUG displays them in the following format:

```
2124:CE06  00  FF  2124:CF10
```

DEBUG: D

Display Memory

Purpose

Displays the contents of an area of memory in hexadecimal and ASCII format.

Syntax

D [*range*]

where:

range is the starting and ending addresses or the starting address and length of the area to be displayed.

Description

The Display Memory (D), or Dump, command displays the contents of a specified range of memory addresses in hexadecimal and ASCII format.

The *range* parameter gives the starting and ending addresses or the starting address and length in bytes of the memory to be displayed. If *range* does not include a segment, DEBUG uses DS.

If *range* is omitted the first time the D command is used, the display starts at the target program's CS:IP registers. If *range* was specified in a preceding D command, the memory address following the last address displayed by that command is used. If a length is not explicitly stated in a D command, 128 bytes are displayed.

Each line displays a segment and offset, followed by the contents of 16 bytes of memory represented as hexadecimal values and separated by spaces (except the eighth and ninth values, which are separated by a dash), followed by the ASCII character equivalents (if any) of the same 16 bytes. In the ASCII portion, nonprinting characters are displayed as periods.

Examples

To display the contents of the 128 bytes of memory beginning at 7F00:0100H, type

```
-D 7F00:0100  <Enter>
```

The contents of the memory addresses are displayed in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C    device...'9...¦
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44   9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3   .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16   .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52   $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00   LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0   .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68   ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

In this case, the contents of memory addresses 7F00:0180H through 7F00:01FFH are displayed.

DEBUG: E

Enter Data

Purpose

Enters data into memory.

Syntax

E *address* [*list*]

where:

address is the first memory location for data entry.
list specifies the data to be entered into successive bytes of memory, starting at *address*.

Description

The Enter Data (E) command allows data to be entered into successive memory locations. The data can be entered in either hexadecimal or ASCII format. Data previously stored in the specified locations is lost.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, DEBUG uses DS. The address is incremented for each byte of data stored.

The *list* parameter is one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes to memory are made unless an error is detected in the command line, in which case an error message is displayed and the E command is terminated. If *list* is omitted from the command line, the user is prompted byte by byte for data to be entered into memory, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for that byte can be entered as one or two hexadecimal digits (extra characters are ignored) or the contents can be left unchanged. Pressing the spacebar displays the contents of the next byte. Entering a minus sign or hyphen character (-) instead of pressing the spacebar displays the contents of the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Pressing the Enter key without pressing the spacebar or entering any data terminates data entry.

Text strings can be entered only by using the *list* parameter; they cannot be entered in response to an address prompt.

Examples

To store the byte values 00H, 0DH, and 0AH in the three bytes beginning at DS:1FB3H, type

```
-E 1FB3 00 0D 0A  <Enter>
```

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-E ES:C14 "MAIN MENU"  <Enter>
```

DEBUG: F

Fill Memory

Purpose

Stores a repetitive data pattern in an area of memory.

Syntax

F *range list*

where:

range	is the starting and ending addresses or starting address and length of the memory to be filled.
list	is the data to be entered.

Description

The Fill Memory (F) command fills an area of memory with the data from a list. The data can be entered in either hexadecimal or ASCII format. Any data previously stored at the specified locations is lost. If an error message is displayed, the original values in memory remain unchanged.

The *range* parameter specifies the starting and ending addresses or the starting address and hexadecimal length in bytes of the area of memory to be filled. If *range* does not specify a segment, DEBUG uses DS.

The *list* parameter specifies one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

If the area to be filled is larger than the data list, the list is repeated as often as necessary to fill the area. If the data list is longer than the area of memory to be filled, it is truncated to fit into the area.

Examples

To fill the area of memory from DS:0B10H through DS:0B4FH with the value 0E8H, type

```
-F B10 B4F E8  <Enter>
```

or

```
-F B10 L40 E8  <Enter>
```

To fill the 16 bytes of memory beginning at address CS:1FA0H by replicating the 2-byte sequence 0DH 0AH, type

```
-F CS:1FA0 1FAF 0D 0A  <Enter>
```

or

```
-F CS:1FA0 L10 0D 0A  <Enter>
```

To fill the area of memory from ES:0B00H through ES:0BFFH by replicating the text string *BUFFER*, type

```
-F ES:B00 BFF "BUFFER"  <Enter>
```

or

```
-F ES:B00 L100 "BUFFER"  <Enter>
```

DEBUG: G

Go

Purpose

Transfers control from DEBUG to the program being debugged.

Syntax

G [=*address*] [*break0* [... *break9*]]

where:

address	is the location DEBUG begins execution.
break0... break9	specify from 1 to 10 temporary breakpoints.

Description

The Go (G) command transfers control from DEBUG to the program being debugged. If no breakpoints are set, the program executes until it crashes or finishes, in which latter case the message *Program terminated normally* is displayed and control returns to DEBUG. (After this message is displayed, the program may need to be reloaded before it can be executed again.)

The *address* parameter can specify any location in memory. If no segment is specified, DEBUG uses the target program's CS register. If *address* is omitted, DEBUG transfers to the current address in the target program's CS:IP registers. An equal sign (=) must precede *address* to distinguish it from the breakpoints *break0... break9*.

The parameters *break0... break9* are addresses that represent from 1 to 10 temporary breakpoints that can be set as part of the G command. A breakpoint is an address at which execution stops. Breakpoints can be placed in any order, because execution stops at the first breakpoint address encountered, regardless of the position of that breakpoint in the list. Each breakpoint address must contain the first byte of an 8086 opcode. DEBUG installs breakpoints by replacing the first byte of the machine instruction at each breakpoint address with an INT 03H instruction (opcode 0CCH). If the program encounters a breakpoint, execution is suspended and control returns to DEBUG. DEBUG then restores the original machine code to the breakpoint addresses; displays the contents of the registers, the status of the flags, and the instruction pointed to by CS:IP; and displays the DEBUG prompt. If the program executes to completion without encountering any of the breakpoints or stops for any reason other than because it encountered a breakpoint, DEBUG does not replace the INT 03H instructions with the original machine code, and the Load File or Sectors (L) command must be used to reload the original program.

The G command requires that the target program's SS:SP registers point to a valid stack that has at least 6 bytes of stack space available. When the G command is executed, it pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the target program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.

Examples

To begin execution of the program in DEBUG's buffer at location CS:110AH and set breakpoints at CS:12FCH and CS:1303H, type

```
-G =110A 12FC 1303  <Enter>
```

To resume execution of the program after a breakpoint has been encountered and control has been returned to DEBUG, type

```
-G  <Enter>
```

Messages

bp Error

More than 10 breakpoints were specified in a G command. The command must be entered again with 10 or fewer breakpoints.

Program terminated normally

No breakpoints were encountered and the target program executed to completion. If breakpoints were set, the original program should be restored with the L command.

DEBUG: H

Perform Hexadecimal Arithmetic

Purpose

Displays the sum and difference of two hexadecimal numbers.

Syntax

H *value1 value2*

where:

value1 and *value2* are any two hexadecimal numbers from 0 through FFFFH.

Description

The Perform Hexadecimal Arithmetic (H) command displays the sum and the difference of two 16-bit hexadecimal numbers—that is, the result of the operations *value1+value2* and *value1−value2*. If *value2* is greater than *value1*, the difference of the two values is displayed as a two's complement number. This command is convenient for quickly calculating addresses and other values during an interactive debugging session.

Examples

To display the sum and the difference of the values 4B03H and 104H, type

```
-H 4B03 104  <Enter>
```

This produces the following display:

```
4C07  49FF
```

If the addition produces an overflow, the four least significant digits are displayed. For example, the command line

```
-H FFFF 2  <Enter>
```

produces the following display:

```
0001 FFFD
```

If the second number is bigger than the first, the difference is displayed in two's complement form. For example, the command line

```
-H 1 2  <Enter>
```

produces the following display:

```
0003 FFFF
```

DEBUG: I

Input from Port

Purpose

Reads and displays 1 byte from an input/output (I/O) port.

Syntax

I *port*

where:

port is an I/O port address from 0 through FFFFH.

Description

The Input from Port (I) command reads the specified I/O port address and displays the data as a two-digit hexadecimal number.

Warning: The I command should be used with caution because it directly accesses the computer hardware and no error checking is performed. Input operations directed to the ports assigned to some peripheral device controllers may interfere with the proper operation of the system. If no device has been assigned to the specified I/O port or if the port is write-only, the value displayed by an I command is unreliable.

Example

To read and display the contents of I/O port 10AH, type

```
-I 10A  <Enter>
```

An example of the output of this command is

```
FF
```

DEBUG: L

Load File or Sectors

Purpose

Loads a file or individual sectors from a disk into DEBUG's memory.

Syntax

L [*address*]

or

L *address drive start number*

where:

address	is the memory location for the data to be read from the disk.
drive	is the number of the disk drive to read (0 = drive A, 1 = drive B, 2 = drive C, and so on).
start	is the hexadecimal number of the first logical sector to load (0–FFFFH).
number	is the hexadecimal number of consecutive sectors to load (0–FFFFH).

Description

The Load File or Sectors (L) command loads a file or individual sectors from a disk. When the L command is entered without parameters or with only an address, the file specified in the DEBUG command line or the one in the most recent Name File or Command-Tail Parameters (N) command line is loaded from the disk into memory. If no segment is specified in *address*, DEBUG uses CS. If the file's extension is .EXE, the file is placed in DEBUG's target program buffer at the load address specified in the .EXE file's header. If the file's extension is .COM, the file is loaded at offset 100H. (If for some reason an address other than 100H is entered for a .EXE or .COM file, an error message is displayed; if the address is 100H, the specification is ignored.) The length of the file or, in the case of a .EXE file, the actual length of the program (the length of the file minus the header) is placed in the target program's BX and CX registers, with the most significant 16 bits in register BX.

The L command can also be used to bypass the MS-DOS file system and directly access logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to load (*number*) must all be specified in the command line.

Note: The L command should not be used to access logical sectors on network drives.

Examples

To load the file specified in the DEBUG command line or in the most recent N command into DEBUG's target program buffer, type

```
-L  <Enter>
```

To load eight sectors from drive B, starting at logical sector 0, to memory location CS:0100H, type

```
-L 100 1 0 8  <Enter>
```

Messages

Disk error reading drive *X*
The specified drive does not exist or the disk in the specified drive is defective.

File not found
The file specified in the most recent N command cannot be found.

DEBUG: M

Move (Copy) Data

Purpose

Copies the contents of one area of memory to another.

Syntax

M *range address*

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be copied.

address is the first byte in which the copied data will be placed.

Description

The Move (Copy) Data (M) command copies data from one memory location to another without altering the data in the original location. If the source and destination areas overlap, the data is copied so that the resulting copy is correct; the data in the *original* location is changed where the two areas overlap.

The *range* parameter specifies either the starting and ending addresses or the starting address and length of the memory to be copied. The *address* parameter is the first byte in which the copy will be placed. If *range* does not contain an explicit segment, DEBUG uses DS; if *address* does not contain a segment, DEBUG uses the segment used for *range.*

Example

To copy the data in locations DS:0800H through DS:08FFH to locations DS:0900H through DS:09FFH, type

```
-M 800 8FF 900  <Enter>
```

or

```
-M 800 L100 900  <Enter>
```

DEBUG: N

Name File or Command-Tail Parameters

Purpose

Inserts filenames and/or switches into the simulated program segment prefix (PSP).

Syntax

N *parameter* [*parameter*. . .]

where:

parameter is one or more filenames or switches to be placed in the simulated PSP.

Description

The Name File or Command-Tail Parameters (N) command is used to enter one or more parameters into the simulated PSP that is built at the base of the buffer holding the program to be debugged. The N command can also be used before the Load File or Sectors (L) and Write File or Sectors (W) commands to name the file to be read from or written to a disk.

The count of the characters following the N command is placed at DS:0080H in the simulated PSP, and the characters themselves are copied into the PSP starting at offset 81H. The string is terminated by a carriage return (0DH), which is not included in the count. If the first and second parameters follow the naming conventions for MS-DOS files, they are parsed into the default file control blocks (FCBs) in the simulated PSP at offsets 5CH and 6CH, respectively. (Switches specified as parameters are stored in the PSP starting at offset 81H along with the rest of the command line but are not included in the FCBs.)

If the N command line contains only one filename, any parameters placed in the default FCBs by a previous N command are destroyed. If the drive specified with the first filename parameter is invalid, the AL register is set to 0FFH. If the drive specified with the second filename parameter is invalid, the AH register is set to 0FFH. The existence of a file specified with the N command is not verified until it is loaded with the L command.

Examples

Assume that DEBUG was started without specifying the name of a target program in the command line. To load the program CLEAN.COM for execution under the control of DEBUG, use the N and L commands together as follows:

```
-N CLEAN.COM  <Enter>
-L  <Enter>
```

Then, to place the parameter MYFILE.DAT in the simulated PSP's command tail and parse MYFILE.DAT into the first default FCB, type

```
-N MYFILE.DAT  <Enter>
```

Finally, to execute the program CLEAN.COM, type

```
-G  <Enter>
```

The result is the same as if the CLEAN.COM program had been run from the MS-DOS command level with the entry

```
C>CLEAN MYFILE.DAT  <Enter>
```

except that the program is executing under the control of DEBUG and within DEBUG's memory buffer.

DEBUG: O

Output to Port

Purpose

Writes 1 byte to an input/output (I/O) port.

Syntax

O *port byte*

where:

port	is an I/O port address from 0 through FFFFH.
byte	is a value from 0 through 0FFH to be written to the I/O port.

Description

The Output to Port (O) command writes 1 byte of data to the specified I/O port address. The data value must be in the range 00H through 0FFH.

Warning: The O command should be used with caution because it directly accesses the computer hardware and no error checking is performed. Attempts to write to some port addresses, such as those for ports connected to peripheral device controllers, timers, or the system's interrupt controller, may cause the system to crash or damage data stored on disk.

Example

To write the value C8H to I/O port 10AH, type

```
-O 10A C8  <Enter>
```

DEBUG: P

Proceed Through Loop or Subroutine

Purpose

Executes a loop, repeated string instruction, software interrupt, or subroutine call to completion.

Syntax

P [=*address*] [*number*]

where:

address is the location of the first instruction to be executed.
number is the number of instructions to execute.

Description

The Proceed Through Loop or Subroutine (P) command transfers control from DEBUG to the target program. The program executes without interruption until the loop, repeated string instruction, software interrupt, or subroutine call at *address* is completed or until the specified number of machine instructions have been executed. Control then returns to DEBUG, and the contents of the target program's registers and the status of the flags are displayed.

If the *address* parameter does not include an explicit segment, DEBUG uses the target program's CS register; if *address* is omitted entirely, execution begins at the address specified by the target's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number.*

If the instruction at *address* is not a loop, repeated string instruction, software interrupt, or subroutine call, the P command functions just like the Trace Program Execution (T) command. The optional *number* parameter specifies the number of instructions to be executed before control returns to DEBUG. If *number* is omitted, DEBUG executes only one instruction. After each instruction is executed, DEBUG displays the contents of the target program's registers, the status of the flags, and the next instruction to be executed.

Warning: The P command cannot be used to trace through ROM.

Example

Assume that the target program's location CS:143FH contains a CALL instruction. To execute the subroutine that is the destination of CALL and then return control to DEBUG, type

```
-P =143F  <Enter>
```

DEBUG: Q

Quit

Purpose

Ends a DEBUG session.

Syntax

Q

Description

The Quit (Q) command terminates the DEBUG program and returns control to MS-DOS or the command shell that invoked DEBUG. Any changes to a program or other file that were not saved on disk with the Write File or Sectors (W) command are lost.

Example

To exit DEBUG, type

```
-Q  <Enter>
```

DEBUG: R

Display or Modify Registers

Purpose

Displays the contents of one or all registers and the status of the CPU flags and allows them to be modified.

Syntax

R [*register*]

where:

register is the two-character name of an Intel 8086/8088 register from the following list:

```
AX BX CX DX SP BP SI DI
DS ES SS CS IP PC
```

or the character F, which specifies the CPU flags.

Description

The Display or Modify Registers (R) command displays the target program's register contents and the status of the CPU flags and allows them to be modified.

If R is entered without a *register* parameter, the contents of all registers and the status of the CPU flags are displayed, followed by a disassembly of the machine instruction currently pointed to by the target program's CS:IP registers.

If *register* is included in the R command line, the contents of the specified register are displayed; then DEBUG prompts with a colon character (:) for a new value. The value is entered by typing one to four hexadecimal digits and then pressing the Enter key. Pressing the Enter key without entering any values leaves the register contents unchanged.

Note: The register name PC is not fully supported in some versions of DEBUG, so the register name IP should be used instead.

Specifying the character F instead of a register name causes DEBUG to display the status of the program's CPU flags as two-character codes from the following list:

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)

(more)

Flag Name	Value If Set (1)	Value If Clear (0)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

After displaying the flag values, DEBUG displays a hyphen (-) prompt on the same line. Any or all flags can then be altered by typing one or more codes (in any order and optionally separated by spaces) from the list above and pressing the Enter key. Pressing the Enter key without entering any codes leaves the status of the flags unchanged.

Examples

To display the contents of the target program's CPU registers and the status of the CPU flags, followed by the disassembled mnemonic for the next instruction to be executed (pointed to by CS:IP), type

```
-R  <Enter>
```

This produces a display in the following format:

```
AX=0000  BX=0000  CX=00A1  DX=0000  SP=FFFE  BP=0000 SI=0000  DI=0000
DS=19A5  ES=19A5  SS=19A5  CS=19A5  IP=0100   NV UP EI PL NZ NA PO NC
19A5:0100 BF8000       MOV   DI,0080
```

To display the value of the target program's BX register, type

```
-R BX  <Enter>
```

If BX contains 0200H, for example, DEBUG displays that value and then issues a prompt in the form of a colon:

```
BX 0200
:
```

The contents of BX can then be altered by typing a new value and pressing the Enter key or left unchanged by pressing the Enter key alone.

To set the direction and carry flags, first type

```
-R F  <Enter>
```

DEBUG displays the flag values, followed by a hyphen (-) prompt:

```
NV UP EI PL NZ NA PO NC  -
```

The direction and carry flags can then be set by entering

```
-DN CY  <Enter>
```

Messages

bf Error
Bad flag: An invalid code for a CPU flag was entered.

br Error
Bad register: An invalid register name was entered.

df Error
Double flag: Two values for the same CPU flag were entered in the same command.

DEBUG: S

Search Memory

Purpose

Searches memory for a pattern of 1 or more bytes.

Syntax

S *range list*

where:

range specifies the starting and ending addresses or the starting address and length of the area to be searched.

list is 1 or more consecutive byte values and/or a string to be searched for.

Description

The Search Memory (S) command searches a designated range of memory for a specified list of consecutive byte values and/or a text string. The starting address of each set of matching bytes is displayed. The contents of the searched area are not altered.

The *range* parameter specifies the starting and ending addresses or the starting address and length in bytes of the area to be searched. If a segment is not included in *range*, DEBUG uses DS. If a segment is specified for the starting address, DEBUG uses the same segment for the ending address. If a starting address and length in bytes is specified, the starting address plus the length minus 1 cannot exceed FFFFH.

The *list* parameter specifies one or more consecutive hexadecimal byte values and/or a string to be searched for, separated by spaces, commas, or tab characters. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

Examples

To search for the string *Copyright* in the area of memory from DS:0000H through DS:1FFFH, type

```
-S 0 1FFF 'Copyright'  <Enter>
```

or

```
-S 0 L2000 "Copyright"  <Enter>
```

If matches are found, DEBUG displays the starting address of each:

```
20A8:0910
20A8:094F
20A8:097C
```

To search for the byte sequence *3BH 06H* in the area of memory from CS:0100H through CS:12A0H, type

```
-S CS:100 12A0 3B 06  <Enter>
```

or

```
-S CS:100 L11A1 3B 06  <Enter>
```

DEBUG: T

Trace Program Execution

Purpose

Executes one or more instructions, displaying the CPU status after each instruction.

Syntax

T [=*address*] [*number*]

where:

address is the location of the first instruction to be executed.
number is the number of machine instructions to be executed.

Description

The Trace Program Execution (T) command executes one or more instructions, starting at the specified address, and after each instruction displays the contents of the CPU registers, the status of the flags, and the instruction pointed to by CS:IP.

Warning: The T command should not be used to execute any instructions that change the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles) or to trace calls made to MS-DOS through Interrupt 21H. The Go (G) command should be used instead.

The *address* parameter points to the first instruction to be executed. If *address* does not include a segment, DEBUG uses the target program's CS register; if *address* is omitted entirely, execution begins at the address specified by the target program's CS:IP registers. If *address* is included, it must be preceded by an equal sign (=) to distinguish it from *number.*

The *number* parameter specifies the hexadecimal number of instructions to be executed before the DEBUG prompt is redisplayed (default = 1). Pressing Ctrl-C or Ctrl-Break interrupts execution of a sequence of T instructions. Consecutive instructions can then be executed individually by entering T commands with no parameters. Pressing Ctrl-S suspends execution and pressing any key then resumes the trace.

Note: The T command can be used to trace through ROM.

Example

To execute one instruction at location CS:1A00H and then return control to DEBUG, displaying the contents of the CPU registers and the status of the flags, type

```
-T =1A00  <Enter>
```

DEBUG: U

Disassemble (Unassemble) Program

Purpose

Disassembles machine instructions into assembly-language mnemonics.

Syntax

U [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the machine code to be disassembled.

Description

The Disassemble (Unassemble) Program (U) command translates machine instructions into assembly-language mnemonics.

The *range* parameter specifies the starting and ending addresses or starting address and length in bytes of the machine instructions to be disassembled. If *range* does not specify a segment, DEBUG uses CS. Note that if the starting address does not fall on an 8086 instruction boundary, the disassembly will be incorrect.

If *range* does not include a length or ending address, 32 (20H) bytes of memory are disassembled beginning at the specified starting address. If *range* is omitted, 32 bytes of memory are disassembled, starting at the address following the last instruction disassembled by the previous U command. If a U command has not been used before and *range* is omitted, disassembly begins at the address specified by the target program's CS:IP registers.

Note: The actual number of bytes displayed may vary slightly from the amount specified in *range* or from the default of 32 bytes because the length of instructions may vary. Also, the U command does not understand instructions specific to the 80186, 80286, and 80386 microprocessors. It displays such instructions as DBs.

Successive 32-byte fragments of code can be disassembled by entering additional U commands without parameters.

Example

To disassemble 8 bytes of machine instructions starting at CS:0100H, type

```
-U 100 107  <Enter>
```

or

```
-U 100 L8  <Enter>
```

DEBUG: W

Write File or Sectors

Purpose

Writes a file or individual sectors to disk.

Syntax

W [*address*]

or

W *address drive start number*

where:

address	is the first memory location of the data to be written.
drive	is the number of the destination disk drive (0 = drive A, 1 = drive B, 2 = drive C, and so on).
start	is the number of the first logical sector to write (0–FFFFH).
number	is the number of consecutive sectors to be written (0–FFFFH).

Description

The Write File or Sectors (W) command transfers a file or individual sectors from memory to the disk.

When the W command is entered without parameters or with only an address, the number of bytes specified by the contents of registers BX:CX is written from memory into the file named in the most recently used Name File or Command-Tail Parameters (N) command or the first file specified in the DEBUG command line if the N command has not been used. Files with a .EXE or .HEX extension cannot be written with the DEBUG W command.

Note: If a Trace Program Execution (T), Go (G), or Proceed Through Loop or Subroutine (P) command has been used or the contents of the BX or CX registers have been changed, the contents of BX:CX must be restored before the W command is used.

When *address* is not included in the command line, the target program's CS:0100H is assumed.

The W command can also be used to bypass the MS-DOS file system and directly access logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to be written (*number*) must all be provided in the command line in hexadecimal format. The W command should not be used to write sectors on network drives.

Warning: Extreme caution must be used with the W command. The disk's file structure can easily be damaged if the wrong parameters are entered.

Example

Assume that the interactive Assemble Machine Instructions (A) command was used to create a program in DEBUG's memory buffer that is 32 (20H) bytes long, beginning at offset 0100H. This program can be written to the file QUICK.COM by using the DEBUG Name File or Command-Tail Parameters (N), Display or Modify Registers (R), and Write File or Sectors (W) commands sequentially. First, use the N command to specify the name of the file to be written:

```
-N QUICK.COM  <Enter>
```

Next, use the R command to set registers BX and CX to the length to be written. Register BX contains the upper, or most significant half, of the length, whereas register CX contains the lower, or least significant half. Type

```
-R CX  <Enter>
```

DEBUG displays the contents of register CX and prompts with a colon (:). Enter the length after the prompt:

```
:20  <Enter>
```

To use the R command again to set register BX to zero, type

```
-R BX  <Enter>
```

followed by

```
:0  <Enter>
```

Finally, to create the disk file QUICK.COM and write the program into it, type

```
-W  <Enter>
```

DEBUG responds:

```
Writing 0020 bytes
```

Messages

EXE and HEX files cannot be written
Files with a .EXE or .HEX extension cannot be written to disk with the W command.

Writing *nnnn* bytes
After a successful write operation, DEBUG displays in hexadecimal format the number of bytes written to disk.

SYMDEB

Symbolic Debugger

Purpose

The Symbolic Debugger (SYMDEB) allows a file to be loaded, examined, altered, and written back to disk. If the file contains a program, the program can be disassembled, modified, traced one instruction at a time, or executed at full speed with breakpoints. SYMDEB can also be used to read, modify, and write absolute disk sectors.

The SYMDEB utility is supplied with the Microsoft Macro Assembler (MASM) versions 4.0 and earlier. This documentation describes SYMDEB version 4.0.

Syntax

SYMDEB

or

SYMDEB [*options*] [*symfile* [*symfile*...]] [*filename* [*parameter*...]]

where:

symfile is the name of a symbol file created with the MAPSYM utility (extension = .SYM).

filename is the name of the binary or executable program file to be debugged.

parameter is a command-line parameter required by the program being debugged.

options is one or more of the following switches. Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

/I (IBM) specifies that the computer is IBM compatible.

/K enables the interactive breakpoint key (Scroll Lock).

/N enables the use of nonmaskable interrupt break systems on IBM-compatible computers (requires special hardware).

/S enables the Screen Swap (\) command on IBM-compatible computers (the /I switch is also required).

/"commands" specifies one or more SYMDEB commands, separated by semicolons and enclosed in quotation marks.

Description

The SYMDEB commands and capabilities are a superset of those in DEBUG. SYMDEB is also able to load and interpret special symbol files that correlate line numbers, symbols, and memory addresses. With the aid of such files, SYMDEB enables the user to specify addresses with labels, variable names, and expressions, rather than only with absolute hexadecimal addresses. SYMDEB's command repertoire also includes I/O redirection commands, floating-point number entry and display commands, and source-code display capabilities that are not present in DEBUG.

The SYMDEB command line typically includes the *filename* parameter, which is the name of an executable program (with the extension .COM or .EXE) to be loaded into SYMDEB's memory buffer. Files with the extension .EXE are loaded in a manner compatible with the MS-DOS loader. Files with the extension .HEX are converted to binary images and loaded at the internally specified address. All other files are assumed to be direct memory images and are read directly into memory starting at offset 100H. If SYMDEB is entered by itself, no file information is read into memory. An appropriate program segment prefix (PSP) is synthesized at the head of SYMDEB's buffer for use by the target program; the PSP includes a command tail at offset 80H and default file control blocks (FCBs) at offsets 5CH and 6CH, constructed from the optional parameters following *filename*. If necessary, contents of the file are relocated so that the file is ready to execute.

The command line can also contain the names of one or more *symfiles*, symbol files that contain symbol and line-number information for the object modules that constitute the program being debugged. A symbol file is created with the MAPSYM utility from a map file produced by the Microsoft Object Linker (LINK). A symbol file always has the extension .SYM. *See* PROGRAMMING UTILITIES: MAPSYM; LINK.

The four command-line switches /I, /K, /N, and /S provide SYMDEB with information about the computer on which the utility is running. The /I switch is used when the computer is IBM compatible; this causes SYMDEB to take full advantage of special hardware features such as the 8259 Programmable Interrupt Controller or the memory-mapped video display. The /K switch enables the interactive breakpoint key (Scroll Lock), which can then be pressed at any time to interrupt a program that is being traced under the control of SYMDEB.

Note: The /K switch is not necessary on an IBM PC/AT, because the Sys Req key is always active as an interactive break key.

The /N switch enables the use of the nonmaskable interrupt as a breakpoint signal on IBM-compatible computers; this interrupt is triggered by hardware-assisted debugging packages such as Periscope and Atron Corporation's Software Probe. The /S switch enables the Screen Swap (\) command, which allows the output from the program being traced to be maintained and displayed on demand on a virtual screen separate from the SYMDEB commands and messages.

Note: The /I, /N, and /S switches are unnecessary on personal computers built by IBM Corporation; SYMDEB automatically enables the capabilities provided by those switches when SYMDEB finds the IBM copyright notice in the machine's ROM.

After SYMDEB and any files named in the command line are loaded, SYMDEB displays its special prompt character, a hyphen (-), and awaits a command. SYMDEB commands consist of one or two letters, usually followed by one or more parameters. SYMDEB treats uppercase and lowercase characters equivalently except when they are contained in strings enclosed within single or double quotation marks. SYMDEB does not execute commands until the Enter key is pressed.

The SYMDEB commands discussed in this section are

Command	Action
A	Assemble machine instructions.
BC	Clear breakpoints.
BD	Disable breakpoints.
BE	Enable breakpoints.
BL	List breakpoints.
BP	Set breakpoints.
C	Compare memory areas.
D	Display memory.
DA	Display ASCII.
DB	Display bytes.
DD	Display doublewords.
DL	Display long reals.
DS	Display short reals.
DT	Display 10-byte reals.
DW	Display words.
E	Enter data.
EA	Enter ASCII string.
EB	Enter bytes.
ED	Enter doublewords.
EL	Enter long reals.
ES	Enter short reals.
ET	Enter 10-byte reals.
EW	Enter words.
F	Fill memory.
G	Go execute program.
H	Perform hexadecimal arithmetic.
I	Input from port.
K	Perform stack trace.
L	Load file or sectors.
M	Move (copy) data.
N	Name file or command-tail parameters.
O	Output to port.
P	Proceed through loop or subroutine.
Q	Quit debugger.
R	Display or modify registers.
S	Search memory.

(more)

Command	Action
S+	Enable source display mode.
S–	Disable source display mode.
S&	Enable source and machine code display mode.
T	Trace program execution.
U	Disassemble (unassemble) program.
V	View source code.
W	Write file or sectors.
X	Examine symbol map.
XO	Open symbol map.
Z	Set symbol value.
<	Redirect SYMDEB input.
>	Redirect SYMDEB output.
=	Redirect SYMDEB input and output.
{	Redirect target program input.
}	Redirect target program output.
~	Redirect target program input and output.
\	Swap screen.
.	Display source line.
?	Help or evaluate expression.
!	Escape to shell.
*	Enter comment.

One or more SYMDEB commands, separated by semicolons and enclosed in double quotation marks, can be included in the original SYMDEB command line in the form /"*commands*" (for example, /"r;d;q"). These commands, which must precede the filename of the program being debugged, are carried out immediately when SYMDEB is loaded. (This is a convenient way to invoke SYMDEB and execute a series of batch commands.)

The parameters for a SYMDEB command include symbols; line numbers; addresses; ranges; and 8-bit, 16-bit, 32-bit, or floating-point values, expressions, and lists. Multiple parameters can be separated by spaces, tabs, or commas.

A symbol is a name that represents a register, an absolute value, a segment address, or a segment offset. A symbol consists of one or more characters but always begins with a letter, an underscore (_), a question mark (?), an at sign (@), or a dollar sign ($). The names of the various 8086/8088/80286 registers and CPU flags are built into SYMDEB and can be used at any time. Other symbols can be used only when one or more symbol files have been loaded in conjunction with the program to be debugged.

Note: SYMDEB regards symbols whose spellings differ only in case as the same symbol. A unique symbol name that does not conflict with programming instructions, register names, or hexadecimal numbers should always be used.

In MASM programs, symbols must be declared PUBLIC in the source code in order to be accessible during debugging (except for segment and group names, which are PUBLIC by default). In programs compiled with the current versions of Microsoft C, FORTRAN, and Pascal, all symbols are passed through for debugging if the proper compilation switch is used; however, familiarity with the compiler's particular naming conventions is necessary (for example, the Microsoft C Compiler adds an underscore character to the beginning of every symbol).

A line number is a combination of decimal numbers, filenames, and symbols that specifies a unique line of text in a program source file. Line numbers always start with a dot character (.) and take one of the following forms:

.[*filename*:]*linenumber*
.+*displacement*
.–*displacement*
.*symbol*[+*displacement*]
.*symbol*[–*displacement*]

The second and third variations specify a line relative to the current line number; the fourth and fifth specify a line number relative to a designated symbol. Line numbers can be used only with programs developed with compilers that generate line-number information. Programs developed with MASM or an incompatible compiler cannot generate line numbers.

An address identifies a unique location in memory. An address can be a simple offset or a complete address consisting of two 16-bit values in the form segment:offset. Each component can be a valid symbol (including CS, DS, ES, or SS, in the case of segments), a 16-bit hexadecimal number in the range 0 through FFFFH, or a symbol plus or minus a displacement. When the segment portion of an address is absent, the segment specified in the previous instance of the same command is used; if no segment was previously specified, SYMDEB uses DS unless an A, G, L, P, T, U, or W command is used, in which case SYMDEB uses CS.

A range specifies an area of memory or a number of data items and can be expressed as either two addresses or a starting address and a length. A length is represented by the letter L followed by a hexadecimal value in the range 0 through FFFFH. The meaning of the length varies with the SYMDEB command used: The length can signify a number of bytes, words, doublewords, real numbers, machine instructions, or source-code lines. If a command requires a range and the ending address is not supplied, SYMDEB usually assumes 128 bytes.

A value represents an integral number and is a combination of one or more digits. The default base for values is hexadecimal, except in the case of floating-point numbers, but other bases can be used by appending a radix character (Y for binary, O or Q for octal, T for decimal, H for hexadecimal) in either uppercase or lowercase. For example, the following values are equivalent:

0040	0100Q
0040H	0100O
0064t	1000000Y

Doubleword (32-bit) values are entered as two hexadecimal integers separated by a colon character (:). Real numbers are always entered in decimal radix, with or without a decimal point or exponent. Leading zeros can be omitted.

An expression is a combination of symbols, numeric constants, and operators that evaluates to an 8-, 16-, or 32-bit value. An expression can be used in place of a simple value in any command. Unary address operators use DS as the default segment for addresses. Expressions are evaluated in order of operator precedence; operators with equal precedence are evaluated from left to right. Parentheses can be used to override the normal operator precedence.

The available unary operators, listed in order of precedence from highest to lowest, are

Operator	**Meaning**
+	Unary plus
–	Unary minus
NOT	One's (bitwise) complement
SEG	Segment address of operand
OFF	Offset of operand
BY	Low-order byte from specified address
WO	Low-order word from specified address
DW	Doubleword from specified address
POI	Pointer from specified address (same as DW)
PORT	Byte input from specified port
WPORT	Word input from specified port

The available binary operators, listed in order of precedence from highest to lowest, are

Operator	**Meaning**
*	Multiplication
/	Integer division
MOD	Modulus
:	Segment override
+	Addition
–	Subtraction
AND	Bitwise Boolean AND
XOR	Bitwise Boolean Exclusive OR
OR	Bitwise Boolean Inclusive OR

A list is composed of one or more values, expressions, or strings, separated by spaces or commas. A string is one or more ASCII characters, enclosed within single or double quotation marks. Case is significant within a string. If the same type of quote character that is used to delimit the string occurs inside the string, the character must be doubled inside the string in order to be interpreted correctly (for example,"A ""quoted"" word").

In a few cases, SYMDEB displays a specific and informative error message in response to an invalid command. In general, though, SYMDEB responds in a generic fashion, pointing to the approximate location of the error with a caret character (^), followed by the word *Error.* For example:

```
-D CS:100,CS:80  <Enter>
             ^ Error
```

SYMDEB maintains a set of virtual CPU registers and flags for a program being debugged. These registers can be examined and modified with SYMDEB commands. When a program is first loaded for debugging, the virtual registers are initialized with the following values:

Register	**.COM Program**	**.EXE Program**
AX	Valid drive code	Valid drive code
BX	Upper half of program size	Upper half of program size
CX	Lower half of program size	Lower half of program size
DX	Zero	Zero
SI	Zero	Zero
DI	Zero	Zero
BP	Zero	Zero
SP	FFFEH or top of available memory minus 2	Size of stack segment
IP	100H	Offset of entry point within target program's code segment
CS	PSP	Base of target program's code segment
DS	PSP	PSP
ES	PSP	PSP
SS	PSP	Base of target program's stack segment

Note: SYMDEB checks the first three parameters in the command line. If the second and third parameters are filenames, SYMDEB checks any drive specifications with those filenames to verify that they designate valid drives. Register AX contains one of the following codes:

Code	**Meaning**
0000H	The drives specified with the second and third filenames are both valid, or only one filename was specified in the command line.
00FFH	The drive specified with the second filename is invalid.
FF00H	The drive specified with the third filename is invalid.
FFFFH	The drives specified with the second and third filenames are both invalid.

Before SYMDEB transfers control to the target program, it saves the actual CPU registers and then loads them with the current values of the virtual registers; conversely, when control reverts to SYMDEB from the target program, the returned register contents are stored back into the virtual register set for inspection and alteration by the SYMDEB user.

Examples

To prepare the program CLEAN.ASM for debugging with SYMDEB, declare all vital labels, procedures, and variable names in the source program PUBLIC. To assemble the program, type

```
C>MASM CLEAN;  <Enter>
```

This produces the relocatable object module CLEAN.OBJ. Then, to link the object module, type

```
C>LINK /MAP CLEAN;  <Enter>
```

This results in the executable program file CLEAN.EXE and the map file CLEAN.MAP.

Note: The /MAP switch must be used even if a map file is specified in the command line. Finally, to create the symbol information file required by SYMDEB, type

```
C>MAPSYM CLEAN  <Enter>
```

At this point, begin symbolic debugging by typing

```
C>SYMDEB CLEAN.SYM CLEAN.EXE  <Enter>
```

Any run-time command-line parameters required by the CLEAN program may be placed in the SYMDEB command line after the filename CLEAN.EXE.

To prepare the program SHELL.C for debugging with SYMDEB, first compile the program with the switches that disable optimization and cause line-number information to be written to the relocatable object module:

```
C>MSC /Zd /Od SHELL;  <Enter>
```

Next, to convert the object module to an executable program and create a map file with line-number information, type

```
C>LINK /MAP /LI SHELL;  <Enter>
```

To create the symbol information file required by SYMDEB for symbolic debugging, type

```
C>MAPSYM SHELL  <Enter>
```

To begin debugging, type

```
C>SYMDEB SHELL.SYM SHELL.EXE  <Enter>
```

To use the SYMDEB utility to inspect or modify memory or to read, modify, and write absolute disk sectors, type

```
C>SYMDEB  <Enter>
```

Message

File not found
The filename supplied as the first parameter in the SYMDEB command line cannot be found.

SYMDEB: A

Assemble Machine Instructions

Purpose

Allows entry of assembler mnemonics and translates them into executable machine code.

Syntax

A [*address*]

where:

address is the starting location for the assembled machine code.

Description

The Assemble Machine Instructions (A) command accepts assembly-language statements, rather than hexadecimal values, for the Intel 8086/8088, 80186, and 80286 (running in real mode) microprocessors and the Intel 8087 and 80287 math coprocessors and assembles each statement into executable machine language.

The *address* parameter specifies the location where entry of assembly-language mnemonics will begin. If *address* is omitted, SYMDEB uses the last address generated by the previous A command; if there was no previous A command, SYMDEB uses the current value of the target program's CS:IP registers.

After the user enters an A command, SYMDEB prompts for each assembly-language statement by displaying the address (a segment and an offset) in which the assembled code will be stored. When the user presses the Enter key, SYMDEB translates the assembly-language statement and stores each byte of the resulting machine instruction sequentially in memory (overwriting any existing information), beginning at the displayed address. SYMDEB then displays the address following the last byte of the machine instruction to prompt the user to enter the next assembled instruction. The user can terminate assembly mode by pressing the Enter key in response to the address prompt.

The assembly-language statements accepted by the SYMDEB A command have some slight syntactic differences and restrictions compared with the Microsoft Macro Assembler programming statements. These differences can be summarized as follows:

- All numbers are assumed to be hexadecimal integers unless otherwise specified with a radix character suffix.
- Segment overrides must be specified by preceding the entire instruction with CS:, DS:, ES:, or SS:.
- File control directives (NAME, PAGE, TITLE, and so forth), macro definitions, record structures, and conditional assembly directives are not supported by SYMDEB.

- When the data type (word or byte) is not implicit in the instruction, the type must be specified by preceding the operand with BYTE PTR (or BY), WORD PTR (or WO), DWORD PTR (or DW), QWORD PTR (or QW), or TBYTE PTR (or TB).
- In a string operation, the size of the string must be specified with a B (byte) or W (word) added to the string instruction mnemonic (for example, LODSB or LODSW).
- The DB and DW instructions accept a parameter of the type *list* and assemble byte and word values directly into memory.
- The WAIT or FWAIT opcodes for 8087/80287 assembler statements are not generated by the system and must be coded explicitly. (Note: 8087/80287 instructions can be assembled if the system is not equipped with a math coprocessor, but the system will crash if an attempt is made to execute them.)
- Addresses must be enclosed in square brackets to be differentiated from immediate operands.
- Repeat prefixes such as REP, REPZ, and REPNZ can be entered either alone on a line preceding the statement they affect or on the same line immediately preceding the statement.
- The assembler will generate the optimal form (SHORT, NEAR, or FAR) for jumps or calls, depending on the destination address, but these can be overridden if the operand is preceded with a NEAR (or NE) or FAR prefix.
- The mnemonic for a FAR RETURN is RETF.

Examples

To begin assembling code at address CS:0100H, type

```
-A 100  <Enter>
```

To assemble the instruction sequence

```
LODS WORD PTR [SI]
XCHG BX,AX
JMP [BX]
```

beginning at address CS:0100H, the following dialogue would take place:

```
-A 100  <Enter>
1983:0100  LODSW  <Enter>
1983:0101  XCHG BX,AX  <Enter>
1983:0103  JMP [BX]  <Enter>
1983:0105  <Enter>
```

To continue assembling at the last address generated by a previous A command (1983:0105H in the preceding example), type

```
-A  <Enter>
```

SYMDEB: BC

Clear Breakpoints

Purpose

Permanently removes sticky breakpoints.

Syntax

BC *

or

BC *list*

where:

*	represents all sticky breakpoints.
list	is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Clear Breakpoints (BC) command permanently clears the sticky breakpoints previously set with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk character (*) follows the BC command, SYMDEB deletes all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BC command, SYMDEB selectively deletes sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the BP command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Sticky breakpoints can be temporarily disabled with the Disable Breakpoints (BD) command and subsequently re-enabled with the Enable Breakpoints (BE) command.

Examples

To clear sticky breakpoints 0, 4, and 8, type

```
-BC 0 4 8  <Enter>
```

To clear all sticky breakpoints, type

```
-BC *  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BC command was entered without parameters.

SYMDEB: BD

Disable Breakpoints

Purpose

Temporarily disables sticky breakpoints.

Syntax

BD *

or

BD *list*

where:

* represents all sticky breakpoints.
list is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Disable Breakpoints (BD) command temporarily disables the sticky breakpoints previously set with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk character (*) follows the BD command, SYMDEB disables all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BD command, SYMDEB selectively disables sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the BP command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Sticky breakpoints disabled with the BD command can be re-enabled with the Enable Breakpoints (BE) command. The Clear Breakpoints (BC) command can be used to permanently delete a sticky breakpoint.

Examples

To disable sticky breakpoints 0, 4, and 8, type

```
-BD 0 4 8  <Enter>
```

To disable all sticky breakpoints, type

```
-BD *  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BD command was entered without parameters.

SYMDEB: BE

Enable Breakpoints

Purpose

Enables disabled sticky breakpoints.

Syntax

BE *

or

BE *list*

where:

*	represents all sticky breakpoints.
list	is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Enable Breakpoints (BE) command enables the sticky breakpoints disabled with the Disable Breakpoints (BD) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk (*) character follows the BE command, SYMDEB enables all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BE command, SYMDEB selectively enables sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the Set Breakpoints (BP) command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Examples

To enable sticky breakpoints 0, 4, and 8, type

```
-BE 0 4 8  <Enter>
```

To enable all sticky breakpoints, type

```
-BE *  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BE command was entered without parameters.

SYMDEB: BL

List Breakpoints

Purpose

Displays information about all sticky breakpoints.

Syntax

BL

Description

The List Breakpoints (BL) command lists the current status of each sticky breakpoint created with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

The BL command lists each sticky breakpoint number, its status code, its address in the target program, the number of passes remaining, and the initial number of passes specified with the BP command (in parentheses). If source display mode was selected with the Enable Source Display Mode (S+) command, SYMDEB also displays the source-file name and the line number that corresponds to each breakpoint location. Breakpoint status codes are

e	Enabled
d	Disabled
v	Virtual

(A virtual breakpoint is a sticky breakpoint set at a symbol contained in a .EXE file that has not yet been loaded into SYMDEB.)

Example

To view the current status of all breakpoints, type

```
-BL  <Enter>
```

If the BP commands

```
-BP0 _TEXT:_main  <Enter>
-BP1 _TEXT:_printf  <Enter>
```

were previously entered, the BL command displays

```
0 e 456E:0010 [_TEXT:_main] dump.C:32
1 e 456E:0612 [_TEXT:_printf]
```

SYMDEB: BP

Set Breakpoints

Purpose

Sets sticky breakpoint locations within the program being debugged.

Syntax

BP[*n*] *address* [*passcount*] ["*commands*"]

where:

n	is the sticky breakpoint number (0–9).
address	is the location of the breakpoint in the target program.
passcount	is the number of times the instruction at *address* should be executed before the breakpoint is taken.
"*commands*"	is one or more SYMDEB commands, separated by semicolons. The entire list must be enclosed in double quotation marks. (Limit = 30 characters.)

Description

The Set Breakpoints (BP) command sets a sticky breakpoint in the program being debugged. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes. When the target program reaches the breakpoint, execution of the program is suspended and control returns to SYMDEB. SYMDEB displays the contents of the registers and flags, followed by a prompt so that the user can enter more commands.

The optional *n* parameter associates an integer in the range 0 through 9, called the breakpoint number, with the sticky breakpoint location. If *n* is omitted, the next available breakpoint number is used. No space is allowed between BP and *n*.

The *address* parameter must point to the first byte of a machine instruction in the program. This parameter may be a symbol, a literal address, or a source-code line number. If a segment is not included, SYMDEB uses the target program's CS register.

The optional *passcount* parameter is the number of times execution should pass through the specified location before the break is taken and control is returned to SYMDEB. The value of *passcount* must be a hexadecimal number in the range 0 through FFFFH (default = 0).

The optional "*commands*" parameter is one or more SYMDEB commands with their associated parameters. Each command must be separated from the others by a semicolon character (;) and the entire list enclosed in double quotation marks ("). A maximum of 30 characters can be specified within the quotation marks. The commands are executed whenever the break is taken.

Examples

To set a sticky breakpoint at location *next_file* in the target program and dump the contents of memory locations DS:0000H through DS:00FFH when the breakpoint is reached, type

```
-BP NEXT_FILE "DB DS:0 L100"  <Enter>
```

To associate the breakpoint number 4 with the location CS:4230H in the program being debugged and pass the breakpoint 16 (10H) times before suspending execution of the program, type

```
-BP4 CS:4230 10  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint command too long!
The "*commands*" parameter exceeded 30 characters.

Breakpoint error!
The BP command was entered without an *address* parameter.

Breakpoint redefined!
A new address was assigned to an existing breakpoint number, or an attempt was made to create a breakpoint with the same address as an existing breakpoint.

Duplicate breakpoint ignored!
An attempt was made to change an existing breakpoint to a breakpoint already specified in the breakpoint list.

Too many breakpoints!
No more sticky breakpoints are available.

SYMDEB: C

Compare Memory Areas

Purpose

Compares two areas of memory and reports any differences.

Syntax

C *range address*

where:

range specifies the starting and ending addresses or the starting address and length of the first area of memory to be compared.

address points to the beginning of the second area of memory to be compared.

Description

The Compare Memory Areas (C) command compares the contents of two areas of memory. The location and contents of any differing bytes are listed in the following form:

address1 byte1 byte2 address2

If no differences are found, the SYMDEB prompt returns.

The *range* parameter specifies the first through last addresses or the starting address and length in bytes of the first area of memory to be compared.

The *address* parameter points to the beginning of the second area of memory to be compared, which is the same size as *range*. If a segment is not included in either *range* or *address*, SYMDEB uses DS.

Example

To compare the 64 bytes beginning at CS:CE00H with the 64 bytes beginning at CS:CF0AH, type

```
-C CS:CE00,CE3F CS:CF0A  <Enter>
```

or

```
-C CS:CE00 L40 CS:CF0A  <Enter>
```

If any differences are found, SYMDEB displays them in the following format:

```
2124:CE06  00  FF  2124:CF10
```

SYMDEB: D

Display Memory

Purpose

Displays the contents of an area of memory.

Syntax

D [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Memory (D) command displays the contents of a specified range of memory addresses in the same format used in the most recent Display command (DA, DB, DD, DL, DS, DT, or DW). If no Display command has previously been entered, the memory is displayed in hexadecimal bytes and their ASCII equivalents (the DB format).

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed or the starting address followed by the length of the area, expressed by an L and the hexadecimal number of data items to be displayed. When *range* does not include a segment, SYMDEB uses DS.

The size in bytes of each item and the default value for the length depend on the type of Display command used: the Display Byte (DB), Display Doubleword (DD), and Display Word (DW) commands default to a length of 128 (80H) bytes; Display ASCII (DA) displays 128 bytes or up to a null byte, whichever is smaller; Display Short Reals (DS), Display Long Reals (DL), and Display 10-Byte Reals (DT) default to the display of one floating-point number.

If a Display command has not previously been used and *range* is omitted from a D command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a D command, the display starts at the memory address following the last address displayed by the most recent Display command.

Examples

Assume that the only Display commands used during this SYMDEB session are D and DB. To display the contents of the 128 bytes of memory beginning at offset 100H in the program's DGROUP, type

```
-D DGROUP:0100  <Enter>
```

SYMDEB displays the contents of the range of memory addresses in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C   device...'9...¦
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44  9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3  .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16  .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52  $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00  LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0  .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68  ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 7F00:0180H through 7F00:01FFH.

SYMDEB: DA

Display ASCII

Purpose

Displays the contents of memory in ASCII format.

Syntax

DA [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display ASCII (DA) command displays the contents of a specified range of memory addresses in ASCII format.

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed in ASCII format or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of bytes. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DA command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DA command, the display starts at the memory address following the last address displayed by the most recent Display command.

When a range is not explicit in a DA command, the display terminates after 128 bytes or when a null (zero) byte is encountered. If a range is specified, the entire range is displayed, including any null bytes, with nonprinting characters displayed as period (.) characters.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory (or less if a null byte was encountered) represented as an ASCII string.

See also PROGRAMMING UTILITIES: SYMDEB:EA.

Examples

If memory beginning at location 7F00:0100H contains the characters *This is a test string* followed by a null (zero) byte, the command

```
-DA 7F00:0100  <Enter>
```

produces the following display:

```
7F00:0100  This is a test string
```

To view additional memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DB

Display Bytes

Purpose

Displays the contents of memory as hexadecimal bytes and their equivalent ASCII characters.

Syntax

DB [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Bytes (DB) command displays the contents of a specified range of memory addresses as hexadecimal bytes and their ASCII character equivalents. This is the default format for the Display Memory (D) command.

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of bytes. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DB command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DB command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DB command, the display terminates after 128 bytes.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as hexadecimal values separated by spaces (except the eighth and ninth values, which are separated by a dash), followed by their ASCII character equivalents (if any). In the ASCII section, nonprinting characters are displayed as periods.

See also PROGRAMMING UTILITIES: SYMDEB:EB.

Examples

To display the contents of the 128 bytes of memory beginning at 7F00:0100H, type

```
-DB 7F00:0100  <Enter>
```

The contents of the range of memory addresses are displayed in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C   device...'9...¦
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44  9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3  .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16  .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52  $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00  LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0  .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68  ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 7F00:0180H through 7F00:01FFH.

SYMDEB: DD

Display Doublewords

Purpose

Displays the contents of memory in hexadecimal doubleword format.

Syntax

DD [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Doublewords (DD) command displays the contents of a specified range of memory addresses 4 bytes at a time, as if they were FAR memory pointers (offset followed by segment in reverse byte order).

The *range* parameter specifies the starting and ending addresses of the memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of doublewords. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DD command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DD command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DD command, 32 doublewords (128 bytes) are displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as 4 paired 16-bit segments and offsets. The 4 bytes that make up the segment and offset of each doubleword pointer are displayed in reverse order from their actual storage in memory.

See also PROGRAMMING UTILITIES: SYMDEB:ED.

Examples

To see how DD represents the 4 bytes that make up a doubleword, first type

```
-DB 100  <Enter>
```

This produces the following output:

```
3929:0100  CF 0B 9D 0D 33 0E C3 0E-F2 0E 06 0F 39 0F 49 0F 0...3.C.r...9.I.
```

Then type

```
-DD 100  <Enter>
```

This produces the following output:

```
3929:0100  0D9D:0BCF 0EC3:0E33 0F06:0EF2 0F49:0F39
```

Notice that DD switches the order of the first 2 bytes in a 4-byte set and designates them as the offset; then it switches the order of the second 2 bytes in the 4-byte set and designates them as the segment address.

To display the contents of the first 128 (80H) bytes of the system interrupt vector table, which is based at address 0000:0000H, type

```
-DD 0:0  <Enter>
```

This produces the following output:

```
0000:0000  2075:03D2 0070:01F0 16F3:2C1B 0070:01F0
0000:0010  0070:01F0 F000:FF54 F000:9805 F000:9805
0000:0020  0AE3:0395 16F3:2BAD F000:9805 F000:9805
0000:0030  0972:0B40 F000:9805 F000:EF57 0070:01F0
0000:0040  0AE3:03D6 F000:F84D F000:F841 0070:0D43
0000:0050  F000:E739 F000:F859 F000:E82E F000:EFD2
0000:0060  F000:E76C 0070:0ADD F000:FE6E 1078:3BEC
0000:0070  F000:FF53 F000:F0E4 0000:0522 F000:0000
```

To view the next 128 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 0000:0080H through 0000:00FFH.

SYMDEB: DL

Display Long Reals

Purpose

Displays the contents of memory as long (64-bit) floating-point numbers.

Syntax

DL [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Long Reals (DL) command displays the contents of a specified range of memory addresses 8 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 64-bit floating-point numbers. The decimal values have the form

+¦–0.*decimaldigits*E+¦–*mantissa*

The sign of the number (+ or –) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits*; this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or –) and digits.

The *range* parameter specifies the starting and ending addresses of the memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 8-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DL command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DL command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DL command, one 64-bit floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 8 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:EL.

Examples

Assume that the memory beginning at location DS:0100H contains the value $6.624 * 10^{-27}$ (Planck's constant, in erg-seconds) as a 64-bit floating-point number. The command

```
-DL 100  <Enter>
```

produces the following output:

```
43E8:0100  5F A2 20 73 75 66 80 3A  +0.6624E-26
```

To view the next 8 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DS

Display Short Reals

Purpose

Displays the contents of memory as short (32-bit) floating-point numbers.

Syntax

DS [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Short Reals (DS) command displays the contents of a specified range of memory addresses 4 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 32-bit floating-point numbers. The decimal values have the form

+¦–0.*decimaldigits*E+¦–*mantissa*

The sign of the number (+ or –) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits* (only the first 7 digits are significant); this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or –) and digits.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 4-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DS command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DS command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DS command, one 32-bit floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 4 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:ES.

Examples

Assume that the memory beginning at location 43E8:0100H contains the value $6.02 * 10^{+23}$ (Avogadro's number) as a 32-bit floating-point number. The command

```
-DS 43E8:100  <Enter>
```

produces the following output:

```
43E8:0100  F9 F4 FE 66  +0.6020000172718952E+24
```

To view the next 4 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DT

Display 10-Byte Reals

Purpose

Displays the contents of memory as 10-byte (80-bit) floating-point numbers.

Syntax

DT [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display 10-Byte Reals (DT) command displays the contents of a specified range of memory addresses 10 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 80-bit floating-point numbers. (This format is ordinarily used by the Intel 8087 math coprocessor only for intermediate results during chained floating-point calculations.) The decimal value has the form

+¦–0.*decimaldigits*E+¦–*mantissa*

The sign of the number (+ or –) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits*; this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or –) and digits.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 10-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DT command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DT command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DT command, one 10-byte floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 10 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:ET.

Examples

Assume that the memory beginning at location DS:0100H contains the value 2.99 $*10^{+10}$ (the speed of light in centimeters per second) as an 80-bit floating-point number. The command

```
-DT 100  <Enter>
```

produces the following output:

```
43E8:0100  00 00 00 00 60 B9 C5 DE 21 40  +0.299E+11
```

To view the next 10 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DW

Display Words

Purpose

Displays the contents of memory as 2-byte (16-bit) words.

Syntax

DW [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Word (DW) command displays the contents of a specified range of memory addresses 2 bytes at a time, as 16-bit hexadecimal integers.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of words of memory to be displayed. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DW command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DW command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DW command, 64 words are displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as eight 4-digit hexadecimal numbers. The 2 bytes that make up each word are displayed in reverse order from their actual storage in memory. That is, the first byte in a 2-byte word is displayed after the second byte.

See also PROGRAMMING UTILITIES: SYMDEB:EW.

Examples

To display the contents of the 64 words of memory beginning at DS:0080H in word format, type

```
-DW 80  <Enter>
```

This produces the following output:

```
1FEE:0080  6977 646E 776F 5C73 696C 0062 494C 3D42
1FEE:0090  3A63 6D5C 6373 6C5C 6269 633B 5C3A 6977
1FEE:00A0  646E 776F 5C73 696C 0062 4D54 3D50 3A63
1FEE:00B0  745C 6D65 0070 4554 504D 633D 5C3A 6574
1FEE:00C0  706D 4400 4149 3D4C 3A63 645C 6169 006C
1FEE:00D0  4350 3346 3D32 3A63 665C 726F 6874 705C
1FEE:00E0  3363 0032 4350 3350 3D32 3A63 665C 726F
1FEE:00F0  6874 705C 756C 3373 0032 5255 3146 3D30
```

To view the next 64 words of memory in the same format, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 1FEE:0100H through 1FEE:017FH.

SYMDEB: E

Enter Data

Purpose

Enters data into memory.

Syntax

E *address* [*list*]

where:

address	is the first memory location for storage.
list	is the data to be placed into successive bytes of memory, starting at *address*.

Description

The Enter Data (E) command enters into memory one or more data items, using the same format as the most recent Enter command (EA, EB, ED, EL, ES, ET, or EW). If no Enter command has previously been used, the data can be entered as either hexadecimal values or ASCII strings (the EA or EB format). Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter must meet the requirements of the last Enter command used. All SYMDEB Enter commands are described in alphabetic order on the following pages. If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the current contents of *address* are displayed, followed by a period (.), and the user is prompted for new data. If no value is entered and the Enter key is pressed, the original value remains unchanged and the Enter command is terminated.

Examples

The following two examples assume that no previous Enter commands have been used or that the most recent Enter command was EA or EB.

To store the byte values 00H, 0DH, and 0AH into the 3 bytes beginning at DS:1FB3H, type

```
-E 1FB3 00 0D 0A  <Enter>
```

If the command

```
-E 2C3 ABC  <Enter>
```

is entered and the last Enter command used was EA or EB, the value BCH is stored at DS:2C3H, and the leading 'A' character on the hexadecimal number 'ABC' is ignored.

SYMDEB: EA

Enter ASCII String

Purpose

Enters an ASCII string or hexadecimal byte values into memory.

Syntax

EA *address* [*list*]

where:

address	is the first memory location for storage.
list	is one or more ASCII strings or hexadecimal byte values.

Description

The Enter ASCII String (EA) command enters data into successive memory bytes. The data can be entered as either hexadecimal byte values or ASCII strings. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. The EA command functions exactly like the Enter Bytes (EB) command.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter is one or more ASCII strings and/or hexadecimal byte values, separated by spaces, commas, or tab characters. Extra or trailing characters are ignored. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the user is prompted byte by byte for new data, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for that byte can be entered as one or two hexadecimal digits (extra characters are ignored), or the contents can be left unchanged. To display the next byte, the user presses the spacebar. If the user enters a minus sign, or hyphen character (-), instead of pressing the spacebar, SYMDEB backs up to the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Data entry is terminated by pressing the Enter key without pressing the spacebar or entering any data.

Text strings can be used only as part of the *list* parameter in an EA command line; they cannot be entered in response to an address prompt.

Example

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-EA ES:C14 "MAIN MENU"  <Enter>
```

SYMDEB: EB

Enter Bytes

Purpose

Enters hexadecimal byte values or ASCII strings into memory.

Syntax

EB *address* [*list*]

where:

address is the first memory location for storage.
list is one or more hexadecimal byte values or ASCII strings.

Description

The Enter Bytes (EB) command enters data into successive memory bytes. The data can be entered as either hexadecimal byte values or ASCII strings. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. The EB command functions exactly like the Enter ASCII String (EA) command.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter is one or more hexadecimal byte values and/or ASCII strings, separated by spaces, commas, or tab characters. Extra or trailing characters are ignored. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the user is prompted byte by byte for new data, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for the byte can be entered as one or two hexadecimal digits (extra characters are ignored), or the contents can be left unchanged. To display the next byte, the user presses the spacebar. If the user enters a minus sign, or hyphen character (-), instead of pressing the spacebar, SYMDEB backs up to the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Data entry is terminated by pressing the Enter key without pressing the spacebar or entering any data.

Text strings can be used only as part of the *list* parameter in an EB command line; they cannot be entered in response to an address prompt.

Examples

To store the byte values 00H, 0DH, and 0AH into the 3 bytes beginning at DS:1FB3H, type

```
-EB 1FB3 00 0D 0A  <Enter>
```

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-EB ES:C14 "MAIN MENU"  <Enter>
```

SYMDEB: ED

Enter Doublewords

Purpose

Enters hexadecimal doubleword values into memory.

Syntax

ED *address* [*value*]

where:

address	is the first memory location for storage.
value	is a doubleword (32-bit) hexadecimal value.

Description

The Enter Doublewords (ED) command enters into memory 32-bit hexadecimal doubleword values in the form of FAR memory pointers (offset followed by segments in reverse byte order). Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is one doubleword value, entered as two 16-bit hexadecimal words separated by a colon character (:). Each value is entered in the form segment:offset. The offset portion is stored at *address*, and the segment portion is stored at *address+2*, both in reverse byte order. For example, a value of AABB:CCDDH would be stored in memory as DDH, CCH, BBH, and AAH, starting at *address*. Multiple values cannot be used in an ED command line; SYMDEB ignores any values after the first value.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed, followed by a period. The user can then enter a new doubleword value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the ED command. If a new value is entered, SYMDEB increments *address* and displays the next doubleword value.

Example

To store the doubleword value F000:1392H at the address DS:0200H, type

```
-ED 200 F000:1392  <Enter>
```

SYMDEB: EL

Enter Long Reals

Purpose

Enters 64-bit floating-point numbers into memory.

Syntax

EL *address* [*value*]

where:

address	is the first memory location for storage.
value	is a 64-bit floating-point decimal number.

Description

The Enter Long Reals (EL) command enters into memory 64-bit floating-point numbers in decimal format. Any data previously stored at the specified memory locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an EL command line; SYMDEB ignores any values after the first value.

The 64-bit floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the long floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed. The user can enter a new value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the EL command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next long real number.

Example

To store an approximation of the value *pi* (π) in the form of a 64-bit floating-point number at address DS:0020H, type

```
-EL 20 +0.3141592653589793E+1  <Enter>
```

or

```
-EL 20 3.141592653589793  <Enter>
```

SYMDEB: ES

Enter Short Reals

Purpose

Enters 32-bit floating-point numbers into memory.

Syntax

ES *address* [*value*]

where:

address	is the first memory location for storage.
value	is a 32-bit floating-point decimal number.

Description

The Enter Short Reals (ES) command enters into memory 32-bit floating-point numbers in decimal format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an ES command line; SYMDEB ignores any values after the first value.

The 32-bit floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the short floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

Note: For short floating-point values, the last nine *decimaldigits* are not significant. This can be demonstrated by using the Display Short Reals (DS) command to check the new value in memory.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed. The user can then enter a new value and press the Enter key or leave the contents unchanged by pressing the

Enter key alone, which also terminates the ES command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next short floating-point number.

Example

To store an approximation of the value *pi* (π) in the form of a 32-bit floating-point number at address DS:0020H, type

```
-ES 20 +0.31415927E+1  <Enter>
```

or

```
-ES 20 3.1415927  <Enter>
```

SYMDEB: ET

Enter 10-Byte Reals

Purpose

Enters 10-byte (80-bit) floating-point numbers into memory.

Syntax

ET *address* [*value*]

where:

address	is the first memory location for storage.
value	is an 80-bit floating-point decimal number.

Description

The Enter 10-Byte Reals (ET) command enters into memory 10-byte (80-bit) floating-point numbers in decimal format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. (This 10-byte format is ordinarily used by the Intel 8087 math coprocessor only for intermediate results during chained floating-point calculations.)

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an ET command line; SYMDEB ignores any values after the first value.

The 10-byte floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the 10-byte floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

If *value* is omitted from the command, SYMDEB prompts the user for new data, starting at *address*. The current contents are displayed. The user can enter a new value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the ET command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next 10-byte floating-point number.

Example

To store an approximation of the value *pi* (π) in the form of an 80-bit floating-point number at address DS:0020H, type

```
-ET 20 +0.31415926535897932384E+1  <Enter>
```

or

```
-ET 20 3.1415926535897932384  <Enter>
```

SYMDEB: EW

Enter Words

Purpose

Enters word values into memory.

Syntax

EW *address* [*value*]

where:

address is the first memory location for storage.
value is a word (16-bit) hexadecimal value.

Description

The Enter Words (EW) command enters into memory 16-bit hexadecimal word values. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is one word value in the range 0 through FFFFH. The value is stored in reverse byte order. For example, a value of AABBH would be stored in memory as BBH and AAH, starting at *address.* Multiple values cannot be used in an EW command line; SYMDEB ignores any values after the first value.

If *value* is omitted from the command line, SYMDEB prompts the user word by word for new data, starting at *address.* The current contents are displayed, followed by a period. The user can enter a new word value as one to four hexadecimal digits and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the EW command. If a new value is entered, SYMDEB increments *address* and displays the next word value.

Example

To store the word value 1355H at the address DS:1C00H, type

```
-EW 1C00 1355 <Enter>
```

SYMDEB: F

Fill Memory

Purpose

Stores a repetitive data pattern into an area of memory.

Syntax

F *range list*

where:

range specifies the starting and ending addresses or the starting address and length of memory to be filled.

list is the data to be used to fill memory.

Description

The Fill Memory (F) command fills an area of memory with the data from a list. The data can be entered in either hexadecimal or ASCII format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *range* parameter specifies the starting and ending addresses or the starting address and hexadecimal length in bytes of the area of memory to be filled. If *range* does not include an explicit segment, SYMDEB uses DS.

The *list* parameter is one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

If the area to be filled is larger than the data list, the list is repeated as often as necessary to fill the area. If the data list is longer than the area of memory to be filled, the list is truncated to fit.

Examples

To fill the area of memory from DS:0B10H through DS:0B4FH with the value 0E8H, type

```
-F B10 B4F E8  <Enter>
```

or

```
-F B10 L40 E8  <Enter>
```

To fill the 16 bytes of memory beginning at address CS:1FA0H by replicating the 2-byte sequence 0DH 0AH, type

```
-F CS:1FA0 1FAF 0D 0A  <Enter>
```

or

```
-F CS:1FA0 L10 0D 0A  <Enter>
```

To fill the area of memory from ES:0B00H through ES:0BFFH by replicating the text string *BUFFER*, type

```
-F ES:B00 BFF "BUFFER"  <Enter>
```

or

```
-F ES:B00 L100 "BUFFER"  <Enter>
```

SYMDEB: G

Go

Purpose

Transfers execution control from SYMDEB to the target program being debugged.

Syntax

G[=*address*] [*break0* [... *break9*]]

where:

address	is the location at which to begin execution.
break0 ... *break9*	specify from 1 to 10 breakpoints.

Description

The Go (G) command transfers control from SYMDEB to the target program. If no breakpoints are set, the program will execute until it crashes or until it reaches a normal termination, in which case the message *Program terminated normally* is displayed and control returns to SYMDEB. (After this message has been displayed, it may be necessary to reload the program before it can be executed again.)

The *address* parameter can be any location in memory. If no segment is specified, SYMDEB uses the target program's CS register. If *address* is omitted, SYMDEB transfers to the current address in the target program's CS:IP registers. An equal sign (=) must precede *address* to distinguish it from the breakpoints *break0* ... *break9*.

The parameters *break0* ... *break9* specify from 1 to 10 breakpoints that can be set as part of the G command. Breakpoints can be placed in any order, because execution stops at the first breakpoint address encountered, regardless of the position of that breakpoint in the list. Each of the breakpoint addresses must contain the first byte of an 8086 opcode. SYMDEB installs breakpoints by replacing the first byte of the machine instruction at each breakpoint address with an Interrupt 03H instruction (opcode 0CCH). If the program encounters a breakpoint, program execution is suspended and control returns to SYMDEB. SYMDEB then restores the original machine code in the breakpoint locations, displays the contents of the current registers and flags and the instruction pointed to by CS:IP, and issues the standard SYMDEB prompt. If the target program executes to completion and terminates without encountering any of the breakpoints or is halted by some means other than a breakpoint, the Interrupt 03H instructions are not replaced with the original machine code and the Load File or Sectors (L) command must be used to reload the original program.

The G command requires that the target program's SS:SP registers point to a valid stack that has at least 6 bytes of stack space available. When the G command is executed, it pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.

The G command also recognizes any sticky breakpoints set with the Set Breakpoint (BP) command. These sticky breakpoints are not counted as part of the transient breakpoints specified in the G command line and are not removed after a breakpoint has been encountered.

Examples

To begin execution of the program in SYMDEB's buffer at location CS:110AH, setting breakpoints at CS:12FCH and CS:1303H, type

```
-G =110A 12FC 1303  <Enter>
```

To resume execution of the program following a breakpoint, type

```
-G  <Enter>
```

To begin execution at the label *main,* setting breakpoints at the procedures *fopen()* and *printf(),* type

```
-G =_main _fopen _printf  <Enter>
```

Messages

Program terminated normally
The program being debugged executed successfully without encountering any breakpoints and performed a normal termination with Interrupt 20H, Interrupt 21H Function 00H, or Interrupt 21H Function 4CH. If any breakpoints were set, the original program should be reloaded with the Load File or Sectors (L) command.

Too many breakpoints!
More than 10 breakpoints were specified in a Go (G) command. Enter the command again with 10 or fewer breakpoints.

SYMDEB: H

Perform Hexadecimal Arithmetic

Purpose

Displays the sum and difference of two hexadecimal numbers.

Syntax

H *value1 value2*

where:

value1 and *value2* are any two hexadecimal numbers in the range 0 through FFFFH.

Description

The Perform Hexadecimal Arithmetic (H) command displays the sum and difference of two 16-bit hexadecimal numbers — that is, the result of the operations *value1+value2* and *value1–value2*. If *value2* is greater than *value1*, SYMDEB displays their difference as a two's complement hexadecimal number. This command is convenient for performing quick calculations of addresses and other values during an interactive debugging session.

Examples

To display the sum and difference of the values 4B03H and 104H, type

```
-H 4B03 104  <Enter>
```

This produces the following display:

```
4C07  49FF
```

If the addition produces an overflow, the four least significant digits are displayed. For example, the command line

```
-H FFFF 2  <Enter>
```

produces the following display:

```
0001 FFFD
```

If *value2* is greater than *value1*, the difference is displayed in two's complement form. For example, the command line

```
-H 1 2  <Enter>
```

produces the following display:

```
0003 FFFF
```

SYMDEB: I

Input from Port

Purpose

Reads and displays 1 byte from an input/output (I/O) port.

Syntax

I *port*

where:

port	is a 16-bit I/O port address in the range 0 through FFFFH.

Description

The Input from Port (I) command performs a read operation on the specified I/O port address and displays the data as a two-digit hexadecimal number.

Warning: This command must be used with caution because it involves direct access to the computer hardware and no error checking is performed. Input operations directed to the ports assigned to some peripheral device controllers may interfere with the proper operation of the system. If no device has been assigned to the specified I/O port or if the port is write-only, the value that will be displayed by an I command is unpredictable.

Example

To read and display the contents of I/O port 10AH, type

```
-I 10A  <Enter>
```

An example of the result of this command is

```
FF
```

SYMDEB: K

Perform Stack Trace

Purpose

Displays the current stack frame.

Syntax

K [*number*]

where:

number is the number of parameters supplied to the current procedure.

Description

The Perform Stack Trace (K) command displays the contents of the current stack frame. The first line of the display shows the name of the current procedure, parameters to the procedure, and the filename and line number of the call to the procedure. The subsequent lines trace the flow of execution that led to the current procedure.

In cases where SYMDEB cannot determine the number of parameters for a procedure by inspection of the stack frame (for example, if the number of parameters sent to a procedure varies), the *number* option can be used in the command to force the display of one or more parameters.

The K command can be used only on procedures that follow the calling conventions used by Microsoft high-level-language compilers.

Examples

Assume that a breakpoint has been set within the C library *printf()* routine, that the breakpoint has been reached, and that the SYMDEB prompt has reappeared. The command

```
-K  <Enter>
```

produces the following output:

```
_TEXT:_printf(00D4,0000,0000) from .dump.C:108
_TEXT:_dump_para(0000,0000,0FB8) from .dump.C:92
_TEXT:_dump_rec(0FB8,0001,0000,0000) from .dump.C:61
_TEXT:_main(?)
```

In this example, the breakpointed procedure *printf()* was called by the routine *dump_para()* with three parameters. *Dump_para()* was called by *dump_rec()*, which in turn was called by *main()*. Because SYMDEB cannot determine the depth of the stack frame for the routine *main()*, it displays no parameters for it. The display of at least two parameters for every procedure can be forced by the command

```
-K 2  <Enter>
```

which produces the following example display:

```
_TEXT:_printf(00D4,0000,0000) from .dump.C:108
_TEXT:_dump_para(0000,0000,0FB8) from .dump.C:92
_TEXT:_dump_rec(0FB8,0001,0000,0000) from .dump.C:61
_TEXT:_main(0002,1044)
```

From a knowledge of C conventions, it follows that the first parameter for *main()* is *argc*, or the number of tokens in the command line that invoked the program being debugged; the second parameter is the offset within DGROUP of *argv*, or an array of pointers to each token.

SYMDEB: L

Load File or Sectors

Purpose

Loads a file or individual sectors from a disk.

Syntax

L [*address*]

or

L *address drive start number*

where:

address is the starting address in memory that data read from a disk is placed into.
drive is the decimal number (0-3) of the disk to read (0 = drive A, 1 = drive B, 2 = drive C, 3= drive D).
start is the hexadecimal number of the first sector to load (0–FFFFH).
number is the hexadecimal number of consecutive sectors to load (0–FFFFH).

Description

The Load File or Sectors (L) command loads a file or individual sectors from a disk.

When the L command is entered without parameters or with an address alone, the file specified in the SYMDEB command line or with the most recent Name File or Command-Tail Parameters (N) command is loaded from the disk into memory. If no segment is specified in *address*, SYMDEB uses CS. If the file's extension is .EXE, the file is placed in SYMDEB's target program buffer at the load address specified in the .EXE file's header; if the file's extension is .COM, the file is loaded at offset 100H. (If for some reason an address is entered for a .EXE or .COM file and the address is anything but 100H, an error message is displayed; if the address is 100H, it will be ignored.) If the file has a .HEX extension, the .HEX file's starting address is added to *address* before loading the file. If *address* is not specified, the .HEX file is placed at its own starting address. The length of the file or, in the case of a .EXE file, the actual length of the program (the length of the file minus the header) is placed in the target program's BX and CX registers, with the most significant 16 bits in register BX.

The L command can also be used to bypass the MS-DOS file system and obtain direct access to logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to read (*number*) must all be specified in the command line.

Note: The L command should not be used to access logical sectors on network drives.

Examples

To load the file specified in the SYMDEB command line or in the most recent N command into SYMDEB's target program buffer, type

```
-L  <Enter>
```

To load eight sectors from drive B, starting at logical sector 0, to memory location CS:0100H in SYMDEB's memory buffer, type

```
-L 100 1 0 8  <Enter>
```

Messages

Disk error reading disk *X*
A hardware-related disk error, such as a checksum error or seek incomplete, was encountered during the execution of an L command.

File not found
The file specified in the most recent N command cannot be found.

SYMDEB: M

Move (Copy) Data

Purpose

Copies the contents of one area of memory to another.

Syntax

M *range address*

where:

range	specifies the starting and ending addresses or the starting address and length of the area of memory to be copied.
address	is the first byte of the destination of the copy operation.

Description

The Move (Copy) Data (M) command copies data from one location in memory to another without altering the data in the original location. If the source and destination areas overlap, the data is copied in the correct order so that the resulting copy is correct; the data in the original location is changed only when the two areas overlap.

The *range* parameter specifies the starting and ending addresses or the starting address and length of the memory to be copied. The *address* parameter is the first byte in which the copy will be placed. If *range* does not contain an explicit segment, SYMDEB uses DS; if *address* does not contain a segment, SYMDEB uses the same segment used for *range.*

Example

To copy the data in locations DS:0800H through DS:08FFH to locations DS:0900H through DS:09FFH, type

```
-M 800 8FF 900  <Enter>
```

or

```
-M 800 L100 900  <Enter>
```

SYMDEB: N

Name File or Command-Tail Parameters

Purpose

Inserts parameters into the simulated program segment prefix (PSP).

Syntax

N *parameter* [*parameter*. . .]

where:

parameter is a filename or switch to be placed into the simulated PSP.

Description

The Name File or Command-Tail Parameters (N) command is used to enter one or more parameters into the simulated PSP that is built at the base of the buffer holding the program to be debugged. The N command can also be used before the Load File or Sectors (L) and Write File or Sectors (W) commands to name a file to be read from a disk or written to a disk.

The count of the characters following the N command is placed at DS:0080H in the simulated PSP and the characters themselves are copied into the PSP starting at DS:0081H. The string is terminated by a carriage return (0DH), which is not included in the count. If the second and third parameters follow the naming conventions for MS-DOS files, they are parsed into the default file control blocks (FCBs) in the simulated PSP, at offset 5CH and offset 6CH, respectively. Note that this is different from the N command in DEBUG, which loads the first and second parameters into the default FCBs. (Switches and other filenames specified as parameters are stored in the PSP starting at offset 81H along with the rest of the command line but are not parsed into the default FCBs.)

If the N command line contains only one filename, any parameters placed in the default FCBs by a previous N command are destroyed. If the drive included with the second filename parameter is invalid, the AL register is set to 0FFH. If the drive included with the third filename parameter is invalid, the AH register is set to 0FFH. The existence of a file specified with the N command is not verified until it is loaded with the L command.

The filename at DS:0081H specifies the file that is read or written by a subsequent L or W command.

Example

Assume that SYMDEB was started without specifying the name of a target program in the command line. To load the program CLEAN.COM for execution under the control of

SYMDEB and include the parameter MYFILE.DAT in the simulated PSP's command tail and FCB, use the N and L commands together as follows:

```
-N CLEAN.COM  MYFILE.DAT  <Enter>
-L  <Enter>
```

To execute the program CLEAN.COM, type

```
-G  <Enter>
```

The net effect is the same as if the CLEAN.COM program had been run from the MS-DOS command level with the command line

```
C>CLEAN MYFILE.DAT  <Enter>
```

except that the program is executing under the control of SYMDEB and within SYMDEB's memory buffer.

SYMDEB: O

Output to Port

Purpose

Writes 1 byte to an input/output (I/O) port.

Syntax

O *port byte*

where:

port	is a 16-bit I/O port address in the range 0 through FFFFH.
byte	is a value to be written to the I/O port (0–0FFH).

Description

The Output to Port (O) command writes 1 byte of data to the specified I/O port address. The data value must be in the range 00H through 0FFH.

Warning: This command must be used with caution because it involves direct access to the computer hardware and no error checking is performed. Attempts to write to some port addresses, such as those for ports connected to peripheral device controllers, timers, or the system's interrupt controller, may cause the system to crash or may even result in damage to data stored on disk.

Example

To write the value C8H to I/O port 10AH, type

```
-O 10A C8  <Enter>
```

SYMDEB: P

Proceed Through Loop or Subroutine

Purpose

Executes a loop, string instruction, software interrupt, or subroutine to completion.

Syntax

P[=*address*] [*number*]

where:

address is the location of the first instruction to be executed.
number is the number of instructions to execute.

Description

The Proceed Through Loop or Subroutine (P) command transfers control to the target program. The program executes without interruption until the loop, repeated string instruction, software interrupt, or subroutine call at *address* is completed or until the specified number of machine instructions have been executed. Control then returns to SYMDEB and the current contents of the target program's registers and flags are displayed.

Warning: The P command should not be used to execute any instruction that changes the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles) and cannot be used to trace through ROM. Use the Go (G) command instead.

If the *address* parameter does not contain a segment, SYMDEB uses the target program's CS register; if *address* is omitted, execution begins at the current address specified by the target's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number.*

The *number* parameter specifies the number of instructions to be executed before control returns to SYMDEB. If *number* is omitted, one instruction is executed.

When the Enable Source Display Mode (S+) command is selected, the P command operates directly on source-code lines, passing over function or procedure calls. (The S+ command can be used only with programs created by high-level-language compilers that insert line-number information into object modules.)

When source display mode is disabled with the S– command or when the program being debugged does not have a .SYM file or has been created with the Microsoft Macro Assembler (MASM) or with a compiler that does not support line numbers in relocatable object modules, the P command behaves like the Trace Program Execution (T) command except that when P encounters a loop, repeated string instruction, software interrupt, or subroutine call, it executes it to completion and then returns to the instruction following the call. For example, if the user wants to trace the first three instructions in a program and if the second instruction is a subroutine call, a P3 command executes the first instruction, goes to the second instruction, identifies it as a CALL instruction, jumps to the subroutine and executes the entire subroutine, comes back and executes the third instruction, and then stops. A T3 command, on the other hand, executes the first instruction, executes the second, executes the first instruction of the subroutine as its third instruction, and then stops. If the instruction at *address* is not a loop, repeated string instruction, software interrupt, or subroutine call, the P command functions just like the T command. After each instruction is executed, SYMDEB displays the current contents of the target program's registers and flags and the next instruction to be executed.

Examples

Assume that the program being debugged was compiled with Microsoft C, a .SYM file was loaded with the executable program to provide line-number information, and source-code display has been enabled with the S+ command. To execute the machine instructions corresponding to the next four lines of source code, type

```
-P 4  <Enter>
```

Assume that the target program was created with MASM and location CS:143FH contains a CALL instruction. To execute the subroutine that is the destination of CALL at full speed and then return control to SYMDEB, type

```
-P =143F  <Enter>
```

SYMDEB: Q

Quit

Purpose

Ends a SYMDEB session.

Syntax

Q

Description

The Quit (Q) command terminates the SYMDEB program and returns control to MS-DOS or the command shell that invoked SYMDEB. Any changes made to a program or other file that were not previously saved to disk with the Write File or Sectors (W) command are lost when the Q command is used.

Example

To exit SYMDEB, type

```
-Q  <Enter>
```

SYMDEB: R

Display or Modify Registers

Purpose

Displays one or all registers and allows a register to be modified.

Syntax

R

or

R *register*[[=] *value*]

where:

register is the two-character name of an Intel 8086/8088 register from the following list:

```
AX BX CX DX SP BP SI DI
DS ES SS CS IP PC
```

or the character F, to indicate the CPU flags.

= is an optional equal sign preceding *value.*

value is a 16-bit integer (0–FFFFH) that will be assigned to the specified register.

Description

The Display or Modify Registers (R) command allows the target program's register contents and CPU flags to be displayed and modified.

If R is entered without a *register* parameter, the current contents of all registers and CPU flags are displayed, followed by a disassembly of the machine instruction currently pointed to by the target program's CS:IP registers.

A register can be assigned a new value in a single command by entering both *register* and *value* parameters, optionally separated by an equal sign (=). If a register is named but no value is supplied, SYMDEB displays the current contents of the specified register and then prompts with a colon character (:) for a new value to be placed in the register. The user can enter the value in any valid radix or as an expression and then press the Enter key. If no radix is appended to the new value, hexadecimal is assumed. If the user presses the Enter key alone in response to the prompt, no changes are made to the register contents.

Note: The PC register name is not supported properly in some versions of SYMDEB, so the IP register name should always be used instead.

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

After displaying the current flag values, SYMDEB again displays its prompt (-). Any or all of the individual flags can then be altered by typing one or more two-character flag codes (in any order and optionally separated by spaces) from the list above and then pressing the Enter key. If the user responds to the prompt by pressing the Enter key without entering any codes, no changes are made to the status of the flags.

Examples

To display the current contents of the target program's CPU registers and flags, followed by the disassembled mnemonic for the next instruction to be executed (pointed to by CS:IP), type

```
-R  <Enter>
```

This produces the following display:

```
AX=0000  BX=0000  CX=00A1  DX=0000  SP=FFFE  BP=0000 SI=0000  DI=0000
DS=19A5  ES=19A5  SS=19A5  CS=19A5  IP=0100   NV UP EI PL NZ NA PO NC
19A5:0100 BF8000       MOV   DI,0080
```

If the source display mode is enabled, the R command displays the following:

```
AX=0000  BX=1044  CX=0000  DX=0102  SP=103C  BP=0000  SI=00EA  DI=115E
DS=2143  ES=2143  SS=2143  CS=1F6E  IP=0010   NV UP EI PL ZR NA PE NC
32:     int    argc;
_TEXT:_main:
1F6E:0010 55             PUSH    BP                            ;BR0
```

This format includes the source code that corresponds to the next instruction to be executed.

To set the contents of register AX to FFFFH without displaying its current value, type

```
-R AX=FFFF  <Enter>
```

or

```
-R AX -1  <Enter>
```

To display the current value of the target program's BX register, type

```
_R BX  <Enter>
```

If BX contains 200H, for example, SYMDEB displays that value and then issues a prompt in the form of a colon:

```
BX 0200
:
```

The contents of BX can then be altered by typing a new value and pressing the Enter key, or the contents can be left unchanged by pressing the Enter key alone.

To set the direction and carry flags, first type

```
_R F  <Enter>
```

SYMDEB displays the current flag values, followed by a prompt in the form of a hyphen character (-). For example:

```
NV UP EI PL NZ NA PO NC  -
```

The direction and carry flags can then be set by entering

```
_DN CY  <Enter>
```

on the same line as the prompt.

Messages

Bad Flag!
An invalid code for a CPU flag was entered.

Bad Register!
An invalid register name was entered.

Double Flag!
Two values for the same CPU flag were entered in the same command.

SYMDEB: S

Search Memory

Purpose

Searches memory for a pattern of one or more bytes.

Syntax

S *range list*

where:

range	is the starting and ending address or the starting address and length in bytes of the area to be searched.
list	is one or more byte values or a string to be searched for.

Description

The Search Memory (S) command searches a designated range of memory for a sequence of byte values or text strings and displays the starting address of each set of matching bytes. The contents of the searched area are not altered.

The *range* parameter specifies the starting and ending address or the starting address and length in bytes of the area to be searched. If a segment is not included in *range*, SYMDEB uses DS. If a segment is specified only for the starting address, SYMDEB uses the same segment for the ending address. If a starting address and length in bytes are specified, the starting address plus the length less 1 cannot exceed FFFFH.

The *list* parameter is one or more hexadecimal byte values and/or strings separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

Examples

To search for the string *Copyright* in the area of memory from DS:0000H through DS:1FFFH, type

```
-S 0 1FFF 'Copyright'  <Enter>
```

or

```
-S 0 L2000 "Copyright"  <Enter>
```

If a match is found, SYMDEB displays the address of each occurrence:

```
20A8:0910
20A8:094F
20A8:097C
```

To search for the byte sequence *3BH 06H* in the area of memory from CS:0100H through CS:12A0H, type

```
-S CS:100 12A0 3B 06  <Enter>
```

or

```
-S CS:100 L11A1 3B 06  <Enter>
```

SYMDEB: S+

Enable Source Display Mode

Purpose

Displays source-code lines, rather than machine instructions.

Syntax

S+

Description

The Enable Source Display Mode (S+) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S+ command causes source code, rather than disassembled machine instructions, to be displayed by those commands.

The S+ command is useful only if the program being debugged was created with a high-level-language compiler capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). When debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to LINK, the S+ command has no effect.

Example

To enable the display of source-code statements during debugging, type

```
-S+  <Enter>
```

SYMDEB: S–

Disable Source Display Mode

Purpose

Displays disassembled machine instructions, rather than source-code lines.

Syntax

S–

Description

The Disable Source Display Mode (S–) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S– command causes disassembled machine instructions, rather than source code, to be displayed by those commands. By default, SYMDEB displays disassembled machine instructions when debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to the Microsoft Object Linker (LINK).

Example

To disable the display of source-code statements during debugging, type

```
-S-  <Enter>
```

SYMDEB: S&

Enable Source and Machine Code Display Mode

Purpose

Displays both source-code lines and disassembled machine instructions.

Syntax

S&

Description

The Enable Source and Machine Code Display Mode (S&) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S& command causes both the disassembled machine instructions and the corresponding source-code lines to be displayed by those commands.

The S& command is useful only if the program being debugged was created with a high-level-language compiler capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). When debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to LINK, the S& command has no effect.

Example

To enable the display of both source-code statements and disassembled machine-code statements during debugging, type

```
-S&  <Enter>
```

SYMDEB: T

Trace Program Execution

Purpose

Executes one or more machine instructions in single-step mode.

Syntax

T[=*address*] [*number*]

where:

address is the location of the first instruction to be executed.
number is the number of machine instructions to be executed.

Description

The Trace Program Execution (T) command executes one or more machine instructions, starting at the specified address. If source display mode has been enabled with the S+ command, each trace operation executes the machine code corresponding to one source statement and displays the lines from the source code. If source display mode has been disabled with the S– command, each trace operation executes an individual machine instruction and displays the contents of the CPU registers and flags after execution.

Warning: The T command should not be used to execute any instruction that changes the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles). Use the Go (G) command instead.

The *address* parameter points to the first instruction to be executed. If *address* does not include a segment, SYMDEB uses the target program's CS register; if *address* is omitted entirely, execution is begun at the current address specified by the target program's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number.*

The *number* parameter specifies the hexadecimal number of source-code statements or machine instructions to be executed before the SYMDEB prompt is displayed again (default = 1). If source display mode is enabled, the *number* parameter is required. Execution of a sequence of instructions using the T command can be interrupted at any time by pressing Ctrl-C or Ctrl-Break and can be paused by pressing Ctrl-S (pressing any key resumes the trace).

Examples

To execute one instruction at location CS:1A00H and then return control to SYMDEB, displaying the contents of the CPU registers and flags, type

```
-T =1A00  <Enter>
```

Consecutive instructions can then be executed by entering repeated T commands with no parameters.

If source display mode has been enabled with a previous S+ command, to begin execution at the label *main* and continue through the machine code corresponding to four source-code statements, type

```
-T =_main 4  <Enter>
```

SYMDEB: U

Disassemble (Unassemble) Program

Purpose

Disassembles machine instructions into assembly-language mnemonics.

Syntax

U [*range*]

where:

range specifies the starting and ending addresses or the starting address and the number of instructions of the machine code to be disassembled.

Description

The Disassemble (Unassemble) Program (U) command translates machine instructions into their assembly-language mnemonics.

The *range* parameter specifies the starting and ending addresses or the starting address and number of machine instructions to be disassembled. If *range* does not include an explicit segment, SYMDEB uses CS. Note that the resulting disassembly will be incorrect if the starting address does not fall on an 8086 instruction boundary.

If *range* does not include the number of machine instructions to be executed or an ending address, eight instructions are disassembled. If *range* is omitted completely, eight instructions are disassembled starting at the address following the last instruction disassembled by the previous U command, if a U command has been used; if no U command has been used, eight instructions are disassembled starting at the address specified by the current value of the target program's CS:IP registers.

The display format for the U command depends on the current source display mode setting and on whether the program was developed with a compatible high-level-language compiler. If the source display mode setting is S– or the program was developed with the Microsoft Macro Assembler (MASM) or a noncompatible high-level-language compiler, the display contains only the address and the disassembled equivalent of each instruction within *range*. (For 8-bit immediate operands, SYMDEB also displays the ASCII equivalent as a comment following a semicolon.) If the setting is S+ or S& and a compatible symbol file containing line-number information was loaded with the program being debugged, the display contains both the source-code lines and their corresponding disassembled machine instructions.

Note: The 80286 instructions that are considered privileged when the microprocessor is running in protected mode are not supported by SYMDEB's disassembler.

Examples

To disassemble four machine instructions starting at CS:0100H, type

```
-U 100 L4  <Enter>
```

This produces the following display:

```
44DC:0100 EC             IN     AL,DX
44DC:0101 B80200         MOV    AX,0002
44DC:0104 E86102         CALL   0368
44DC:0107 57             PUSH   DI
```

Successive eight-instruction fragments of machine code can be disassembled by entering additional U commands without parameters.

When a program is being debugged with a symbol file that contains line-number information and source display mode has been enabled, disassembled machine code is accompanied by the corresponding source code:

```
43:         if (argc != 2)
28A5:0031 837E0402       CMP    Word Ptr [BP+04],+02
28A5:0035 7503           JNZ    _main+2A (003A)
28A5:0037 E91400         JMP    _main+3E (004E)
44:              {  fprintf(stderr,"\ndump: wrong number of parameters\n");
28A5:003A B83600         MOV    AX,0036
28A5:003D 50             PUSH   AX
28A5:003E B8F600         MOV    AX,00F6
28A5:0041 50             PUSH   AX
28A5:0042 E8AC04         CALL   _fprintf
28A5:0045 83C404         ADD    SP,+04
45:                 return(1);
28A5:0048 B80100         MOV    AX,0001
28A5:004B E9AA00         JMP    _main+E8 (00F8)
```

SYMDEB: V

View Source Code

Purpose

Displays lines from the source-code file for the program being debugged.

Syntax

V *address* [*length*]

or

V [.*sourcefile*:*linenumber*]

where:

address	is the location of an executable instruction in the target program.
length	is an ending address or the number of source-code lines.
.*sourcefile*	is the base name of the source file of the program being debugged, preceded by a period (.).
linenumber	is the first literal line number of .*sourcefile* to be displayed.

Description

The View Source Code (V) command displays lines of source code for the program being debugged, beginning at the location specified by *address*. If *address* does not include a segment, SYMDEB uses the target program's CS register.

The optional *length* parameter can be an ending address or an L followed by a hexadecimal number of source-code lines. If *length* is not specified, eight lines of source code are displayed.

If the .*sourcefile* parameter is specified, followed by a colon character (:) and a line number, eight lines of source code are displayed, starting at *linenumber*. If the V command is entered without parameters after the .*sourcefile*:*linenumber* parameter has been specified, eight lines are displayed from the current source file, beginning with the line after the last line displayed with the V command. The .*sourcefile* parameter must be the name of a high-level-language source file in the current directory. Pathnames and extensions are not supported. The *length* option cannot be used with the .*sourcefile* parameter.

Warning: Specifying a file that does not exist in the current directory may cause the system to crash.

The V command can be used only with programs created by a high-level-language compiler that is capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). The current source display mode setting (S–, S+, or S&) has no effect on the V command.

Examples

Assume that the program DUMP.EXE is being debugged with the aid of the symbol file DUMP.SYM and that the source file DUMP.C is available in the current directory. To display eight lines of source code beginning at the label *_main,* type

```
-V _main  <Enter>
```

This produces the following output:

```
32:         int   argc;
33:         char  *argv[];
34:
35:     {   FILE *dfile;                      /* control block for input file */
36:         int status = 0;                   /* status returned from file read */
37:         int file_rec = 0;                 /* file record number being dumped */
38:         long file_ptr = 0L;               /* file byte offset for current rec */
39:         char file_buf[REC_SIZE];          /* data block from file */
```

To view eight lines of source code from the file DUMP.C, beginning with line 20, type

```
-V .DUMP:20  <Enter>
```

Message

Source file for *filename* (cr for none)?
The current directory does not contain the source file specified with the *.sourcefile* parameter. Enter the correct filename or press Enter to indicate no source file.

SYMDEB: W

Write File or Sectors

Purpose

Writes a file or individual sectors to disk.

Syntax

W [*address*]

or

W *address drive start number*

where:

address	is the first location in memory of the data to be written.
drive	is the number of the destination disk drive (0 = drive A, 1 = drive B, 2 = drive C, 3 = drive D).
start	is the number of the first logical sector to be written (0–FFFFH).
number	is the number of consecutive sectors to be written (0–FFFFH).

Description

The Write File or Sectors (W) command transfers a file or individual sectors from memory to disk.

When the W command is entered without parameters or with an address alone, the number of bytes specified by the contents of registers BX:CX are written from memory to the file named by the most recent Name File or Command-Tail Parameters (N) command or to the first file specified in the SYMDEB command line if the N command has not been used.

Note: If a Go (G), Proceed Through Loop or Subroutine (P), or Trace Program Execution (T) command was previously used or the contents of the BX or CX registers were changed, BX:CX must be restored before the W command is used.

When *address* is not included in the command line, SYMDEB uses the target program's CS:0100H. Files with a .EXE or .HEX extension cannot be written with the W command.

The W command can also be used to bypass the MS-DOS file system and obtain direct access to logical sectors on the disk. To use the W command in this way, the memory address (*address*), disk unit number (*drive*), starting logical sector number (*start*), and number of sectors to be written (*number*) must all be provided in the command line in hexadecimal format.

Warning: Extreme caution should be used with the W command. The disk's file structure can easily be damaged if the command is entered incorrectly. The W command should not be used to write logical sectors to network drives.

Example

Assume that the interactive Assemble Machine Instructions (A) command was used to create a program in SYMDEB's memory buffer that is 32 (20H) bytes long, beginning at offset 100H. This program can be written into the file QUICK.COM by sequential use of the Name File or Command-Tail Parameters (N), Display or Modify Registers (R), and Write File or Sectors (W) commands. First, use the N command to specify the name of the file to be written:

```
-N QUICK.COM  <Enter>
```

Next, use the R command to set registers BX and CX to the length to be written. Register BX contains the upper half or most significant part of the length; register CX contains the lower half or least significant part. Type

```
-R CX  <Enter>
```

SYMDEB displays the current contents of register CX and issues a colon character (:) prompt . Enter the length after the prompt:

```
:20  <Enter>
```

To use the R command again to set the BX register to zero, type

```
-R BX  <Enter>
```

Then type

```
:0  <Enter>
```

To create the disk file QUICK.COM and write the program into it, type

```
-W  <Enter>
```

SYMDEB responds:

```
Writing 0020 bytes
```

Messages

EXE and HEX files cannot be written
Files with a .EXE or .HEX extension cannot be written to disk with the W command.

Writing *nnnn* bytes
After a successful write operation, SYMDEB displays in hexadecimal format the number of bytes written to disk.

SYMDEB: X

Examine Symbol Map

Purpose

Displays names and addresses in the symbol maps.

Syntax

X[*]

or

X? [*map*!] [*segment*:] [*symbol*]

where:

map!	is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).
segment:	is the name of a segment within the currently open or specified *map*, followed by a colon character (:).
symbol	is a symbol name within the specified *segment*.

Description

The Examine Symbol Map (X) command displays the addresses and names of symbols in the currently open symbol maps. (SYMDEB maintains a symbol map for each symbol file specified in the SYMDEB command line.)

If the X command is followed by the asterisk wildcard character (*), the map names, segment names, and segment addresses for all currently loaded symbol maps are displayed. If X is entered alone, the information is displayed only for the active symbol map.

Information from the symbol maps can be displayed selectively by following the X? command with the *map*!, *segment*:, and *symbol* parameters. The three parameters may be used individually or in combination, but at least one parameter must be specified.

The *map*! parameter must be terminated by an exclamation point and consists of the name, without the extension, of a previously loaded symbol file. If *map*! is omitted, SYMDEB uses the currently open symbol map. If more than one .SYM file is specified in the command line, the one with the same name as the program being debugged is opened first.

The *segment*: parameter must be terminated with a colon; it is the name of a segment declared within the specified or currently open symbol map.

The *symbol* parameter is the name of a label, variable, or other object within the specified *segment*.

Any or all parameters can consist of or include the asterisk wildcard character. For example, X?* displays everything in the current map.

Examples

Assume that the program DUMP.EXE is being debugged with the symbol file DUMP.SYM. If the following is typed

```
-X  <Enter>
```

SYMDEB displays:

```
[456E DUMP]
      [456E _TEXT]
       4743 DGROUP
```

This indicates that the program contains one executable code segment (named _TEXT), which is loaded at segment 456EH, and one NEAR DATA group and segment (named DGROUP), which is loaded at segment 4743H.

To display the addresses of all procedures in the same example program whose names begin with the character *f,* type

```
-X? _TEXT:_F*  <Enter>
```

This produces the following listing:

```
_TEXT: (456E)
0428 _fclose            04CB _fopen             04F1 _fprintf
0528 _fread             0ACB _fflush            0BC2 _free
19AD _flushall
```

Note: Unlike the Microsoft C Compiler, SYMDEB is not case sensitive.

SYMDEB: XO

Open Symbol Map

Purpose

Selects the active symbol map and/or segment.

Syntax

XO [*map*!] [*segment*]

where:

map! is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).

segment is the name of the segment that will become the active segment in the current symbol map.

Description

The Open Symbol Map (XO) command selects the active symbol map and/or the active segment within the current symbol map to be used during debugging.

The optional *map*! parameter must be terminated by an exclamation point and must be the name, without the extension, of a symbol file specified in the original SYMDEB command line. If *map*! is omitted, no changes are made to the active symbol map.

The optional *segment* parameter must be the name of a segment within the current or specified symbol map. All segments in the active symbol map are accessible; the active segment is searched first for symbols specified in other SYMDEB commands. If *segment* is omitted and a new active symbol map is specified, the segment with the smallest address in the new active symbol map will become the active segment.

Examples

Assume that the program SHELL.EXE has been loaded with the two symbol files SHELL.SYM and VIDEO.SYM. To use the information loaded from VIDEO.SYM as the active symbol map for debugging, type

```
-XO VIDEO!  <Enter>
```

Subsequent entry of the command

```
-XO _TEXT  <Enter>
```

causes the segment _TEXT within the symbol map VIDEO to be searched first for symbol names.

Message

Symbol not found
The specified symbol map or segment does not exist.

SYMDEB: Z

Set Symbol Value

Purpose

Assigns a value to a symbol.

Syntax

Z [*map*!] *symbol value*

where:

map! is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).

symbol is an existing symbol name in the active symbol map or in the symbol map specified by *map*!.

value is the new address of *symbol* (0–FFFFH).

Description

The Set Symbol Value (Z) command allows the address associated with a name in one of the loaded symbol maps to be overridden by a new value.

Note that altering the address of a symbol at debugging time will not affect other addresses or values that were derived from the value of the same symbol at compilation or assembly time.

The optional *map*! parameter must be terminated by an exclamation point and must be the name, without the extension, of a symbol file specified in the original SYMDEB command line. If *map*! is omitted, SYMDEB uses the active symbol map.

The *symbol* parameter specifies the name of a label, variable, or other object in *map*! or the active symbol map.

The *value* parameter specifies a new address to be associated with *symbol*.

To debug programs created with older versions of FORTRAN and Pascal (Microsoft versions earlier than 3.3 or IBM versions earlier than 2.0), the user must start SYMDEB, locate the first procedure of the program being debugged, and then use the Z command to set the address of DGROUP to the current value of the DS register. (Later versions of FORTRAN and Pascal do this by default.)

Examples

To change the segment address for the symbol DGROUP to 5000H, type

```
-Z DGROUP 5000  <Enter>
```

The actual data associated with the label DGROUP must be moved to the new address before debugging can continue.

To change the segment address for the symbol CODE in the inactive symbol map COUNT to 0F00H, type

```
-Z COUNT! CODE F00  <Enter>
```

SYMDEB: <

Redirect SYMDEB Input

Purpose

Redirects input to SYMDEB.

Syntax

< *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect SYMDEB Input (<) command causes SYMDEB to read its commands from the specified text file or character device, rather than from the keyboard (CON).

The *device* parameter specifies the name of any MS-DOS device or file from which commands will be read. If the *device* parameter is a filename, the file must be an ASCII text file and each command in the file must be on a separate line.

If input will be taken from a terminal attached to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

When SYMDEB commands are redirected from a file, the last entry in the file must be either the < CON command, which restores the keyboard as the input device, or the Quit (Q) command. Otherwise, SYMDEB will lock and the system will have to be restarted.

Examples

Assume that the text file FILL.TXT contains the following SYMDEB commands:

```
F CS:0100 L100 00
D CS:0100 L100
R
Q
```

To process FILL.TXT during a SYMDEB session (which in turn exits SYMDEB with the Quit [Q] command), type

```
-< FILL.TXT  <Enter>
```

Assume that the text file SEARCH.TXT contains the following SYMDEB commands:

```
S BUFFER L2000 "error"
< CON
```

To process SEARCH.TXT during a SYMDEB session and return control to the console, type

```
-< SEARCH.TXT  <Enter>
```

SYMDEB: >

Redirect SYMDEB Output

Purpose

Redirects SYMDEB's output to a device or file.

Syntax

> *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect SYMDEB Output (>) command causes SYMDEB to send all its messages to the specified device or file, rather than to the video display (CON). This is useful for creating a record of a debugging session that can be viewed later with an editor or listed on a printer.

After SYMDEB output is redirected, commands typed on the keyboard are not echoed to the video display. Therefore, the user must know in advance which commands to use and which parameters to supply.

The *device* parameter specifies the name of an MS-DOS device or file to receive SYMDEB's output. If output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Output can be restored to the video display by entering the > CON command or by terminating SYMDEB with the Quit (Q) command.

Examples

To cause SYMDEB to send all prompts and messages to the file SESSION.TXT, type

```
-> SESSION.TXT  <Enter>
```

After this command, new commands are still accepted by SYMDEB, but the keypresses are not echoed to the screen until the command

```
-> CON  <Enter>
```

is entered or SYMDEB is terminated with the Quit (Q) command.

To cause SYMDEB to send all its prompts and messages to the standard printing device, PRN, type

```
-> PRN  <Enter>
```

SYMDEB: =

Redirect SYMDEB Input and Output

Purpose

Redirects both input and output for SYMDEB.

Syntax

= *device*

where:

device is the name of any MS-DOS device.

Description

The Redirect SYMDEB Input and Output (=) command causes SYMDEB to read its commands from and send its output to the specified device, rather than reading from the keyboard and sending output to the video display (CON). This command is especially useful for debugging programs that run in graphics mode; the SYMDEB commands can be entered on a terminal attached to the computer's serial port while the graphics program has the full use of the system's video display.

The *device* parameter specifies the name of any MS-DOS device. If input and output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Input and output can be restored to the standard settings with the = CON command.

Example

To redirect SYMDEB's input and output to the first serial communications port (COM1), type

```
-= COM1  <Enter>
```

SYMDEB: {

Redirect Target Program Input

Purpose

Redirects input to the program being debugged.

Syntax

{ *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect Target Program Input ({) command causes read operations by the program being debugged to be taken from the specified file or device when the program is executed, rather than from the keyboard (CON).

The *device* parameter specifies the name of an MS-DOS device or file from which the target program will read. If the *device* parameter is a filename, the file must be an ASCII text file and each command in the file must be on a separate line.

If input will be taken from a terminal attached to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To cause input for the program being debugged to be taken from the file TEST.TXT, type

```
-{ TEST.TXT  <Enter>
```

SYMDEB: }

Redirect Target Program Output

Purpose

Redirects the output of the program being debugged.

Syntax

} *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect Target Program Output (}) command causes write operations by the program being debugged to be redirected to the specified device or file when the program is executed, rather than to the video display (CON). This is useful for capturing the output of a program in a file for later listing on a printer.

The *device* parameter specifies the name of an MS-DOS device or file to receive the target program's output. If output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To send the output from the program being debugged to the file SESSION.TXT, type

```
-} SESSION.TXT  <Enter>
```

SYMDEB: ~

Redirect Target Program Input and Output

Purpose

Redirects both input and output for the program being debugged.

Syntax

~ *device*

where:

device is the name of any MS-DOS device.

Description

The Redirect Target Program Input and Output (~) command causes all read and write operations by the program being debugged to be redirected to the specified character device.

The *device* parameter specifies the name of an MS-DOS device that the target program will read from and write to. If input and output are redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To redirect input and output for the program being debugged to the first serial communications port (COM1), type

```
-~ COM1  <Enter>
```

SYMDEB: \

Swap Screen

Purpose

Exchanges the SYMDEB display for the target program's display.

Syntax

\

Description

The Swap Screen (\) command causes the SYMDEB status display to be exchanged for the virtual screen used by the program being debugged. After the program's output has been inspected on the virtual screen, the SYMDEB display can be restored by pressing any key. This command is useful for debugging programs that perform direct screen access or run in graphics mode.

Note: Any information on the display when SYMDEB was invoked will also appear on the virtual screen. When SYMDEB is terminated, the current display is set to match the virtual screen.

The Swap Screen command is available only if the /S switch (or the /I switch, if the computer is IBM compatible) preceded the names of the symbol and program files in the original SYMDEB command line.

Example

To exchange the SYMDEB status display for the virtual screen of the program being debugged, type

```
-\  <Enter>
```

To restore the SYMDEB display, press any key.

SYMDEB: .

Display Source Line

Purpose

Displays the current source-code line.

Syntax

```
.
```

Description

The Display Source Line (.) command displays the line from the source-code file that corresponds to the machine instruction currently pointed to by the target program's CS:IP registers.

The . command is independent of the current Source Display Mode status (S+, S–, or S&). However, if the program being debugged was not created with a high-level-language compiler that inserts line numbers into the object modules, the . command has no effect.

Example

To display the source-code line corresponding to the next instruction to be executed, type

```
-.  <Enter>
```

This produces output in the following form:

```
56:        printf( '\nDump of file: %s ', argv[1] );
```

SYMDEB: ?

Help or Evaluate Expression

Purpose

Displays the help screen or the value of an expression.

Syntax

? [*expression*]

where:

expression is any valid combination of symbols, addresses, numbers, and operators.

Description

When ? is entered alone, a help screen summarizing all valid SYMDEB commands, operators, and types is displayed.

When ? is followed by the *expression* parameter, *expression* is evaluated and the value is displayed. The *expression* parameter can include any valid combination of symbols, addresses, numbers, and operators.

The form and content of the resulting display depends on the type of expression entered. If *expression* is a symbol or an address (optionally including operators), the value is shown first as a FAR address pointer in the form segment:offset, then as a 32-bit hexadecimal number representing the value's physical location in memory (followed by its decimal equivalent in parentheses), and finally as the physical location's ASCII character equivalents displayed as a string enclosed in quotation marks (which have no practical value if *expression* is an address or symbol).

If *expression* includes numbers (interpreted as signed hexadecimal values unless a radix is specified) and operators, the resulting value is shown first as a 16-bit hexadecimal value, then as a 32-bit hexadecimal value (followed by its decimal equivalent in parentheses), and finally as the value's ASCII character equivalents displayed as a string enclosed in quotation marks.

(The ASCII characters within the string are displayed as dots if their value is less than 20H [32] or greater than 7EH [126].)

Examples

Assume that the pointer array *argv* in the program DUMP.C is located at address 4743:029CH. The command

```
-? _argv+4  <Enter>
```

produces the following display:

```
4743:02A0h  000476D0  (292560)
```

To display the result of an exclusive OR operation between the values 0FCH and 14H, type

```
-? FC XOR 14  <Enter>
```

SYMDEB displays

```
00E8h  000000E8  (232)
```

SYMDEB: !

Escape to Shell

Purpose

Invokes the MS-DOS command processor.

Syntax

! [*command*]

where:

command is the name of any MS-DOS command, program, or batch file and its required parameters.

Description

The Escape to Shell (!) command loads a copy of the system's command processor (COMMAND.COM), optionally passing it the name of a program or batch file to be executed. This allows MS-DOS functions such as listing or copying files to be carried out without losing the context of the debugging session.

If the ! command is entered alone, an additional copy of COMMAND.COM gains control and displays the system prompt. Control can be returned to SYMDEB by leaving the new shell with the EXIT command.

If the ! character is followed by a *command* parameter that specifies any valid MS-DOS command, program name, or batch-file name, the specified command is executed immediately and control returns directly to SYMDEB.

The SYMDEB statement connector (;) cannot be used on the same line as the ! command; all text encountered after this command is passed to COMMAND.COM and is interpreted as an MS-DOS command line.

Example

To list the files in the current directory, type

```
-! DIR /W  <Enter>
```

Messages

COMMAND.COM not found!
SYMDEB could not find COMMAND.COM because it was not present in the directory location specified in the environment block's COMSPEC variable.

Not enough memory!
Free memory in the transient program area (TPA) is insufficient to execute the requested command or program. This is a common occurrence when debugging a large program with symbol files.

SYMDEB: *

Enter Comment

Purpose

Allows insertion of a comment that will be ignored by SYMDEB's command interpreter.

Syntax

**text*

where:

text is any ASCII text up to and including a carriage return.

Description

The Enter Comment (*) command causes the remainder of the text on that line to be ignored, thereby providing a means of commenting a SYMDEB debugging session. SYMDEB echoes any text following the asterisk to the screen or redirected output device, providing the user with a convenient way to comment program output redirected to a file or a printer. A maximum of 78 characters can be included on each comment line. Comment lines are also useful for documenting lines within a text file that SYMDEB will use as redirected input for the program being debugged.

Example

To echo the reminder *Errors in program output start here:* to the screen or redirected output device, type

```
-*Errors in program output start here:  <Enter>
```

A line in a text file that will be used by SYMDEB for redirected input to the program being debugged may be "commented out" by inserting an asterisk at the beginning of the line. For example:

```
*EB CS:1200 90
```

CodeView

Window-Oriented Debugger

Purpose

Allows the controlled execution of an assembly-language program or high-level-language program for debugging purposes. Both source code and the corresponding unassembled machine code can be displayed as program execution is traced. In addition, watch variables, CPU registers and flags, and program output can be examined in separate debugging windows. CodeView is supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. This documentation describes CodeView version 2.0.

Syntax

CV [*options*] *exe_file* [*parameters*]

where:

exe_file is the name of the executable file containing the program to be debugged (default extension = .EXE).

parameters is one or more filenames or switches required by the program being debugged.

options is one or more switches from the following list. Switches can be either uppercase or lowercase and can be preceded by a dash (–) instead of a forward slash (/).

/2 Allows the use of two video displays for debugging.

/43 Enables 43-line display mode. (An IBM-compatible computer with an enhanced graphics adapter [EGA] and an enhanced color display is required for this option.)

/B Forces the attached monitor to use two shades of color when displaying information.

/C*commands* Executes the specified list of startup commands when CodeView is invoked. If the list of startup commands contains any spaces, the entire list must be enclosed in double quotation marks ("). Commands in the list must be separated by a semicolon character (;).

/D Turns off nonmaskable interrupt trapping and Intel 8259 interrupt trapping. (This switch prevents system crashes on some IBM-compatible machines that do not support certain IBM-specific interrupt trapping functions.)

(more)

/E Stores the symbolic information of the program in expanded memory.

/F Enables the screen-flipping method of switching between the debugging display and the virtual output display. Screen flipping is the default method for IBM-compatible computers with color/graphics adapters.

/I Enables nonmaskable interrupt trapping and Intel 8259 interrupt trapping on computers that are not IBM-compatible.

/M Disables mouse support within CodeView.

/P Enables palette register restore mode, which allows non-IBM EGAs to restore the proper colors upon return from the virtual output screen.

/R Enables Intel 80386 debugging registers.

/S Enables the screen-swapping method of switching between the debugging display and the virtual output display. Screen swapping is the default method for IBM-compatible computers with monochrome adapters.

/T Disables window mode. This switch is necessary for some non-IBM computers or when a sequential debugging session is desired.

/W Enables window mode. This switch allows CodeView to operate in multiple windows on the same screen. (This option is not the default for some computers.)

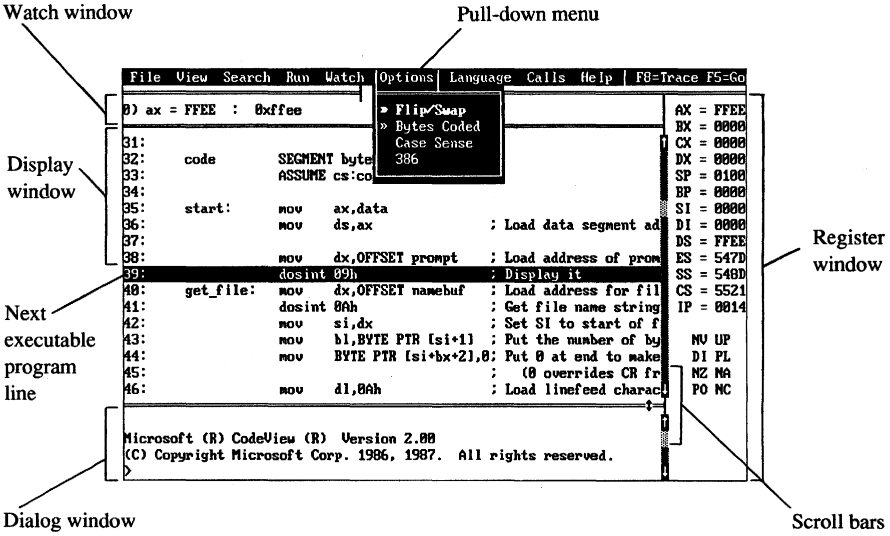
Description

CodeView is a window-oriented menu-driven debugger that allows tracing and debugging of high-level-language programs and assembly-language programs. In general, any valid C, FORTRAN, BASIC, Pascal, or MASM source code can be debugged with CodeView.

To prepare a program for debugging under CodeView, the program must be compiled and linked so that the resulting executable file has the extension .EXE and contains line-number information, a symbol table, and executable code. (To a limited extent, text files and .COM files can also be examined under CodeView.) During the debugging session, the program source file must remain in the current directory if source-code display is desired.

The CodeView screen contains four windows that display information about the program being debugged: the display window, which contains program source code and (if requested) the unassembled machine code corresponding to the source code; the dialog window, where line-oriented commands similar (and in some cases identical) to SYMDEB can be entered and viewed (*see* PROGRAMMING UTILITIES: SYMDEB); the register window (optional), which contains the current status of the microprocessor's registers and flags; and the watch window (optional), which contains program variables or memory locations to be examined during program execution. CodeView also provides a virtual output screen (stored internally) that contains all display output generated during the CodeView session.

A typical CodeView debugging screen looks like this:



```
0) ax = FFEE  :  0xffee                                       AX = FFEE
                                                              BX = 0000
                                                              CX = 0000
        code        SEGMENT byte                              DX = 0000
                    ASSUME cs:co                              SP = 0100
                                                              BP = 0000
        start:      mov    ax,data                            SI = 0000
                    mov    ds,ax                ; Load data segment ad DI = 0000
                                                              DS = FFEE
                    mov    dx,OFFSET prompt     ; Load address of prom ES = 547D
                    dosint 09h                  ; Display it  SS = 548D
        get_file:   mov    dx,OFFSET namebuf    ; Load address for fil CS = 5521
                    dosint 0Ah                  ; Get file name string IP = 0014
                    mov    si,dx                ; Set SI to start of f
                    mov    bl,BYTE PTR [si+1]   ; Put the number of by  NV UP
                    mov    BYTE PTR [si+bx+2],0; Put 0 at end to make   DI PL
                                                ;   (0 overrides CR fr  NZ NA
                    mov    dl,0Ah               ; Load linefeed charac  PO NC

Microsoft (R) CodeView (R)  Version 2.00
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
>
```

The CodeView display.

Display window commands

Commands that control the display window are available in nine pull-down menus whose names appear in a menu bar near the top of the screen. Commands can be selected with the keyboard or the mouse. Commands are selected with the keyboard by pressing the Alt key, pressing the first letter in the menu name, and then pressing the first letter of the command. Commands are selected with the mouse by pulling down the menu with the mouse pointer, highlighting the command, and then releasing the mouse button. Commands with small double arrows to the left of the command name are currently active. The CodeView menus and commands are described below.

File menu

The File menu includes commands that manipulate the current source or program file. To select the File menu with the keyboard, press Alt-F.

Command	Action
Open...	Opens the specified source file, *include* file, or text file in the display window.
DOS Shell	Exits to the shell temporarily. Type *exit* to return to CodeView.
Exit	Ends the current CodeView session.

View menu

The View menu includes commands that select source or assembly modes and commands that select the debugging screen or the virtual output screen. To select the View menu with the keyboard, press Alt-V.

Command	Action
Source	Displays only the high-level-language or assembly-language source code corresponding to the program being debugged.
Mixed	Displays both the unassembled machine code and the source code corresponding to the program being debugged.
Assembly	Displays only the unassembled machine code corresponding to the program being debugged.
Registers	Displays or removes the optional register window.
Output	Replaces the debugging screen with the virtual output screen. Press any key to return to the debugging screen.

Search menu

The Search menu includes commands that search through text files for text strings and through executable code for labels. To select the Search menu with the keyboard, press Alt-S.

Command	Action
Find...	Searches the current source file or other text file for the specified expression.
Next	Searches forward through the file for the next match of the last expression specified with the Find... command.
Previous	Searches backward through the file for the next match of the last expression specified with the Find... command.
Label...	Searches the executable code for the specified procedure name or program label.

Run menu

The Run menu includes commands that run the program being debugged. To select the Run menu with the keyboard, press Alt-R.

Command	Action
Start	Runs the program at full speed from the first instruction.
Restart	Reloads the program and moves to the first instruction.
Execute	Runs the program at reduced speed from the current instruction.
Clear Breakpoints	Clears all breakpoints.

Watch menu

The Watch menu includes commands that add watch statements to and delete watch statements from the watch window. Watch statements describe expressions or areas of memory to be examined during program execution. To select the Watch menu with the keyboard, press Alt-W.

Command	Action
Add Watch...	Adds the specified watch-expression statement to the watch window.
Watchpoint...	Adds the specified watchpoint statement to the watch window. A watchpoint is a conditional breakpoint that is taken when the expression becomes nonzero (true).
Tracepoint...	Adds the specified tracepoint statement to the watch window. A tracepoint is a conditional breakpoint that is taken when a given expression or range of memory changes.
Delete Watch...	Deletes the specified statement from the watch window.
Delete All Watch	Deletes all statements from the watch window.

Options menu

The Options menu contains commands that affect the general behavior of CodeView. To select the Options menu with the keyboard, press Alt-O.

Command	Action
Flip/Swap	When on (the default), enables screen swapping or screen flipping (whichever option CodeView was started with); when off, disables swapping or flipping. Either method can be used to display the CodeView virtual output screen.
Bytes Coded	When on (the default), displays the instructions, instruction addresses, and the bytes for each instruction; when off, displays only the instructions.
Case Sense	When on, causes CodeView to assume that symbol names are case sensitive; when off, causes CodeView to assume that symbol names are not case sensitive. This option is on by default for C programs and off by default for FORTRAN, BASIC, and assembly programs.
386	When on, allows instructions that reference 32-bit instructions to be assembled and executed and the register window to display 32-bit values. When off, does not allow Intel 80386 instructions and registers to be supported.

Language menu

The Language menu contains commands that select the language-dependent expression evaluator or instruct CodeView to select it for you. To select the Language menu with the keyboard, press Alt-L.

Command	Action
Auto	Forces CodeView to select the expression evaluator of the source file being loaded, based on the extension of the source file.
Basic	Uses a BASIC expression evaluator to determine the value of source-level expressions.
C	Uses a C expression evaluator to determine the value of source-level expressions.
Fortran	Uses a FORTRAN expression evaluator to determine the value of source-level expressions.

Calls menu

The Calls menu is different from other menus in that its contents vary depending on the status of the program. The Calls menu lists the names of specific routines that will be displayed on the screen when that routine name is selected. Routine names in the Calls menu can be selected by typing the number displayed immediately to the left of a routine name. The cursor will move to the line at which the selected routine was last executing.

The current value of each parameter, if any, is shown in parentheses following the name of the routine in the Calls menu. The menu expands to accommodate the parameters of the widest line. Parameters are shown in the current radix (default = decimal). If the program contains more active routines than will fit on the screen or if the routine parameters are too wide, the menu expands to the left and right.

To select the Calls menu with the keyboard, press Alt-C.

Help menu

The Help menu lists the major topics in the CodeView "linked-list" help system. For help, pull down the Help menu and then select the topic of interest. To select the Help menu with the keyboard, press Alt-H.

Command	Action
Intro to Help	Displays information about the "linked-list" help system.
Keyboard/Mouse	Displays information about keyboard and mouse commands.
Run commands	Displays information about Run commands.
Display cmds.	Displays information about Display commands.
Watch/Break	Displays information about setting, listing, and deleting watchpoints and breakpoints.
Memory Ops	Displays information about viewing and modifying memory.
System cmds.	Displays information about system and environment commands.
About CodeView	Displays information about the current CodeView version, time, and date.

Key commands

CodeView supports a variety of function keys and key combinations that modify the active window.

Key	Action
F1	Displays the introductory help screen.
F2	Displays or removes the register window.
F3	Changes the display in the display window to source, mixed, or assembly mode.
F4	Displays the virtual output screen (press any key to return).
F5	Executes to the next breakpoint or to the end of the program if no breakpoint is encountered.
F6	Toggles between the display window and the dialog window.
F7	Sets a temporary breakpoint on the line containing the cursor and executes to that line (or the next breakpoint).
F8	Executes a trace command, stepping through program calls if present.
F9	Sets or clears a breakpoint on the line containing the cursor.
F10	Executes the next source line (in source mode) or the next instruction (in assembly mode), stepping over program calls if present.
Ctrl+G	Increases the size of the display window or the dialog window, whichever is active.
Ctrl+T	Decreases the size of the display window or the dialog window, whichever is active.

Dialog window commands

After CodeView and the specified executable file are loaded, CodeView displays its special prompt character (>) at the bottom of the dialog window and awaits a dialog command. CodeView dialog commands consist of one, two, or three characters, usually followed by one or more parameters. CodeView treats uppercase and lowercase characters the same except when they are contained in strings enclosed within single or double quotation marks. The default radix for dialog command parameters is 10 (decimal). Dialog commands are executed when the Enter key is pressed.

A detailed explanation of CodeView dialog commands and parameters is not presented in this entry. CodeView dialog commands and parameters are similar to SYMDEB commands and parameters. *See* PROGRAMMING UTILITIES: SYMDEB. Additional information about using CodeView dialog commands and parameters can be found in the CodeView documentation supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. A sample debugging session using CodeView dialog commands and window commands is documented in this book. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Debugging in the MS-DOS Environment.

The dialog commands available with CodeView are as follows:

Command	Syntax	Action
!	! [*command*]	Escape to shell.
"	"	Pause redirected file execution.
#	#*number*	Set display window tabs.
*	*comment	Echo comment to output device.
.	.	Display current source line.
/	/[*searchtext*]	Search for regular expression.
7	7	Display 8087 registers.
:	:[:]...[:]	Delay redirected file execution.
<	< *device*	Redirect dialog window input.
=	= *device*	Redirect dialog window input and output.
>	[T] > [>] device	Redirect dialog window output.
?	? *expression*[,*format*]	Evaluate expression.
@	@	Redraw screen.
A	A [*address*]	Assemble machine instructions.
BC	BC [*] [*list*]	Clear breakpoints.
BD	BD [*] [*list*]	Disable breakpoints.
BE	BE [*] [*list*]	Enable breakpoints.
BL	BL	List breakpoints.
BP	BP [*address* [*passcount*] ["*cmds*"]]	Set breakpoints.
C	C *range address*	Compare memory areas.
D	D [*range*]	Display (dump) memory.
DA	DA [*range*]	Display ASCII.
DB	DB [*range*]	Display bytes.
DD	DD [*range*]	Display doublewords.
DI	DI [*range*]	Display integers.
DL	DL [*range*]	Display long reals.
DS	DS [*range*]	Display short reals.
DT	DT [*range*]	Display 10-byte reals.
DU	DU [*range*]	Display unsigned integers.
DW	DW [*range*]	Display words.
E	E *address* [*list*]	Enter data.
EA	EA *address* [*list*]	Enter ASCII string.
EB	EB *address* [*list*]	Enter bytes.
ED	ED *address* [*value*]	Enter doublewords.
EI	EI *address* [*list*]	Enter integers.
EL	EL *address* [*value*]	Enter long reals.
ES	ES *address* [*value*]	Enter short reals.
ET	ET *address* [*value*]	Enter 10-byte reals.

(more)

Command	Syntax	Action
EU	EU *address* [*value*]	Enter unsigned integers.
EW	EW *address* [*value*]	Enter words.
F	F *range list*	Fill memory.
G	G [*breakpoint*]	Go execute program.
H	H	Display help screen.
I	I *port*	Input from port.
K	K [*number*]	Perform stack trace.
L	L [*parameters*]	Reload program.
M	M *range address*	Move (copy) data.
N	N [*radix*]	Change current radix.
O	O *port byte*	Output to port.
O	O	Display all options.
O3	O3[+¦-]	Toggle Intel 80386 option.
OB	OB[+¦-]	Toggle bytes coded option.
OC	OC[+¦-]	Toggle case-sense option.
OF	OF[+¦-]	Toggle flip/swap option.
P	P [*count*]	Step through program (over calls).
Q	Q	Quit debugger.
R	R [*register* [*value*]]	Display or modify registers.
RF	RF [*flags*]	Display or modify flags.
S	S *range list*	Search memory.
S	S	Display current display mode.
S+	S+	Display source code.
S–	S–	Display assembly language.
S&	S&	Display source code and assembly language.
T	T [*count*]	Trace program execution (through calls).
TP	TP [*type*] *range*	Set memory-tracepoint statement.
TP?	TP? *expression*[,*format*]	Set tracepoint-expression statement.
U	U [*range*]	Disassemble (unassemble) program.
USE	USE [*language*]	Switch expression evaluators.
V	V [.[*filename*:]*linenumber*]	View source code.
W	W	List watchpoints and tracepoints.
W	W [*type*] *range*	Set memory-watch statement.
W?	W? *expression*[,*format*]	Set watch-expression statement.
WP?	WP? *expression*[,*format*]	Set watchpoint.
X	X[?[*module*!] [*routine*.]*symbol*¦*]	Examine program symbols.
Y	Y [*] [*list*]	Delete watch statements.
\	\	Display virtual output screen.

Examples

To prepare the source file SHELL.C for debugging with CodeView, first compile the source file with the switches that disable optimization and cause symbol-table and line-number information to be written to the relocatable object module:

```
C>MSC /Zi /Od SHELL;  <Enter>
```

Next, to convert the object module to an executable program and prepare it for CodeView, type

```
C>LINK /CO SHELL;  <Enter>
```

To begin debugging, type

```
C>CV SHELL  <Enter>
```

To start CodeView in 43-line mode with TEST.EXE as the executable file and INFO.DAT as the command-tail parameter, type

```
C>CV /43 TEST INFO.DAT  <Enter>
```

In both examples the source file corresponding to the specified executable file must be in the current directory if source-code display is desired.

Messages

Argument to IMAG/DIMAG must be simple type
An invalid parameter to an IMAG or DIMAG function, such as an array with no subscripts, was specified.

Array must have subscript
An array without any subscripts was specified in an expression, such as *IARRAY+2.* A correct example is *IARRAY[1]+2.*

Bad address
An invalid address was specified. For example, an address containing hexadecimal characters might have been specified when the radix is decimal.

Bad breakpoint command
An invalid breakpoint number was specified with the BC, BD, or BE dialog command. The breakpoint number must be in the range 0 through 19.

Bad flag
An invalid flag mnemonic was specified with the RF dialog command.

Bad format string
An invalid format specifier was used following an expression. Expressions used with the ?, W?, WP?, and TP? dialog commands can have format specifiers set off from the expression by a comma. The valid format specifiers are c, d, e, E, f, g, G, i, o, s, u, x, and X. Some format specifiers can be preceded by the prefix h (to specify a 2-byte integer) or l (to specify a 4-byte integer).

Bad integer or real constant
An invalid numeric constant was specified in an expression.

Bad intrinsic function
An invalid intrinsic function name was specified in an expression.

Badly formed type
The type information in the symbol table of the file being debugged is incorrect. This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Bad radix (use 8, 10, or 16)
An invalid radix was specified with the N dialog command. Use an octal, decimal, or hexadecimal radix.

Bad register
An invalid register name was specified with the R dialog command. Use AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, or IP. If your machine is equipped with an Intel 80386 microprocessor, use EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, DS, ES, FS, GS, SS, CS, or IP.

Bad subscript
An invalid subscript expression was specified for an array, such as *IARRAY(3.3)* or *IARRAY((3,3))*. The correct expression for this example (in BASIC or FORTRAN) is *IARRAY (3,3)*.

Bad type cast
Incompatible types of operands were specified in an expression.

Bad type (use one of 'ABDILSTUW')
An invalid type was used in a Display (D, DA, DB, DF, DU, DW, DD, DS, DL, or DT) dialog command. The valid types are ASCII (A), byte (B), integer (I), unsigned (U), word (W), doubleword (D), short real (S), long real (L), and 10-byte real (T).

Breakpoint # or '*' expected
The BC, BD, or BE dialog command was entered without a parameter.

Cannot cast complex constant component into REAL
An incompatible real or imaginary component was specified in a COMPLEX constant. Both real and imaginary components must be compatible with type REAL.

Cannot cast IMAG/DIMAG argument to COMPLEX
An invalid parameter was specified with an IMAG or DIMAG function. IMAG and DIMAG parameters must be simple numeric types.

Cannot use struct or union as scalar
A struct or union variable was used as a scalar value in a C expression. Such variables must be followed by a file specifier or preceded by the address-of (&) operator.

Can't find *filename*
CodeView could not find the executable file specified in the command line.

Character constant too long
A character constant that is too long for the FORTRAN expression evaluator was specified. The limit is 126 bytes.

Character too big for current radix
A radix that is larger than the current CodeView radix was specified in a constant. Use the N dialog command to change the radix.

Constant too big
An unsigned constant number larger than 4,294,967,295 (FFFFFFFFH) was specified.

CPU not an 80386
The 386 option was selected but a machine without an Intel 80386 microprocessor is being used.

Divide by zero
An expression in a parameter of a dialog command attempted to divide by zero.

EMM error
CodeView failed to use the Expanded Memory Manager (EMM) correctly. This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

EMM hardware error
The Expanded Memory Manager (EMM) routines reported a hardware error. Check your expanded memory board for defects.

EMM memory not found
The /E option was used but expanded memory has not been installed. Install software that accesses the memory according to the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS).

EMM software error
The Expanded Memory Manager (EMM) routines reported a software error. Reinstall the EMM software.

Expression too complex
An expression given as a dialog-command parameter is too complex.

Extra input ignored
Too many parameters were specified with a command. CodeView evaluates the valid parameters and ignores the rest. In this situation, CodeView often does not evaluate the parameters as intended.

Flip/Swap option off— application output lost
The program being debugged is writing to the screen, but the output cannot be displayed because the flip/swap option has been disabled.

Floating point error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Illegal instruction
This message usually indicates that a machine instruction attempted to divide by zero.

Index out of bound
A subscript value was specified that is outside the bounds declared for the array.

Insufficient EMM memory
Expanded memory is insufficient to hold the program's symbol table.

Internal debugger error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Invalid argument
An invalid CodeView expression was specified as a parameter.

Invalid executable file format — please relink
The executable file was not linked with the version of LINK released with this version of the CodeView debugger. Relink with the appropriate version of LINK.

Invalid option
An invalid switch was specified with the O command.

Missing ' " '
A string specified as a parameter to a dialog command did not have a closing double quotation mark.

Missing '('
A parameter to a dialog command was specified as an expression containing a right parenthesis but no left parenthesis.

Missing ')'
A parameter to a dialog command was specified as an expression containing a left parenthesis but no right parenthesis.

Missing ']'
A parameter to a dialog command was specified as an expression containing a left bracket but no right bracket, or a regular expression was specified with a right bracket but no left bracket.

Missing '(' in complex constant
An opening parenthesis of a complex constant in an expression was expected but was not found.

Missing ')' in complex constant
A closing parenthesis of a complex constant in an expression was expected but was not found.

Missing ')' in substring
A closing parenthesis of a substring expression was expected but was not found.

Missing '(' to intrinsic
An opening parenthesis for an intrinsic function was expected but was not found.

Missing ')' to intrinsic
A closing parenthesis for an intrinsic function was expected but was not found.

No closing single quote
A character was specified in an expression used as a dialog-command parameter, but the closing single quotation mark is missing.

No code at this line number
A breakpoint was set on a source line that does not correspond to machine code. (In other words, the source line does not contain an executable statement.) For example, the line might be a data declaration or a comment.

No free EMM memory handles
CodeView could not find an available EMM handle. Expanded Memory Manager (EMM) software allocates a fixed number of memory handles (usually 256) to be used for specific tasks.

No match of regular expression
No match was found for the regular expression specified with the Search (S) dialog command or with the Find... command from the Search menu.

No previous regular expression
The Previous command was selected from the Search menu, but CodeView found no previous match for the last regular expression specified.

No source lines at this address
The address specified as a parameter for the V dialog command does not have any source lines. For example, it could be an address in a library routine or an assembly-language module.

No such file/directory
The specified file or directory does not exist.

No symbolic information
The executable file specified is not in the CodeView format. The program cannot be debugged in source mode unless the file is created in the CodeView format. The program can be debugged in assembly mode.

Not an executable file
The file specified to be debugged when CodeView started is not an executable file with a .EXE or .COM extension.

Not a text file
An attempt was made to load a file with the Open... command from the File menu or with the V dialog command, but the file is not a text file. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9 through 13 and 20 through 126.

Not enough space
The ! dialog command or the DOS Shell command from the File menu was chosen, but free memory is insufficient to execute COMMAND.COM. Because memory is released by code in the FORTRAN startup routines, this error always occurs if the ! command is used before executing any code. Use any of the code-execution dialog commands (T, P, or G) to execute the FORTRAN startup code; then try the ! command again. This message also occurs with assembly-language programs that do not specifically release memory.

Object too big
A TP? dialog command was entered with a data object (such as an array) that is larger than 128 bytes.

Operand types incorrect for this operation
An operand in a FORTRAN expression had a type incompatible with the operation applied to it. For example, if P is declared as *CHARACTER P (10)*, then *?P+5* would produce this error, because a character array cannot be an operand of an arithmetic operator.

Operator must have a struct/union type
One of the C member-selection operators (–, >, or .) was used in an expression that does not reference an element of a structure or union.

Operator needs lvalue
An expression was specified that does not evaluate to a memory location in an operation that requires one. (An lvalue is an expression that refers to a memory location.) For example, *buffer (count)* is correct; it represents a symbol in memory. However, *I .EQV. 10* is invalid because it evaluates to TRUE or FALSE instead of to a single memory location.

Overlay not resident
An attempt was made to unassemble machine code from a function that is currently not in memory.

Program terminated normally (*exitcode*)
The program terminated execution normally. The number displayed in parentheses is the exit code returned to MS-DOS by the program.

Radix must be between 2 and 36 inclusive
A radix that is outside the allowable range was specified.

Register variable out of scope
An attempt was made to specify a register variable by using the period (.) operator and a routine name.

Regular expression too complex
The regular expression specified is too complex for CodeView to evaluate.

Regular expression too long
The regular expression specified is too long for CodeView to evaluate.

Restart program to debug
The program being debugged has executed to the end.

Simple variable cannot have argument
A parameter to a simple variable was specified in an expression. For example, given the declaration *INTEGER NUM*, the expression *NUM(I)* is not allowed.

Substring range out of bound
A character expression exceeded the length specified in the CHARACTER statement.

Syntax error
An invalid command line was specified for a dialog command, or an invalid assembly-language instruction was entered with the A dialog command.

Too few array bounds given
The bounds specified in an array subscript do not match the array declaration. For example, given the array declaration *INTEGER IARRAY(3,4)*, the expression *IARRAY(I)* would produce this error message.

Too many array bounds given
The bounds specified in an array subscript do not match the array declaration. For example, given the array declaration *INTEGER IARRAY(3,4)*, the expression *IARRAY (I,3,J)* would produce this error message.

Too many breakpoints
An attempt was made to specify more than 20 breakpoints; CodeView permits only 20.

Too many files
Too few file handles were specified for CodeView to operate correctly. Specify more files in your CONFIG.SYS file.

Type clash in function argument
The type of an actual parameter does not match the corresponding formal parameter, or a subroutine that uses alternate returns was called and the values of the return labels in the actual parameter list are not 0.

Type conversion too complex
An attempt was made to typecast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type would be typecasting to a struct or union type. An example of two levels of indirection is *char ***.

Unable to open file
A file specified in a command parameter or in response to a prompt cannot be opened.

Unknown symbol
An identifier that is not in CodeView's symbol table was specified, or a local variable was used in a parameter when not in the routine where the variable is defined, or a subroutine that uses alternate returns was called and the values of the return labels in the parameter list are not 0.

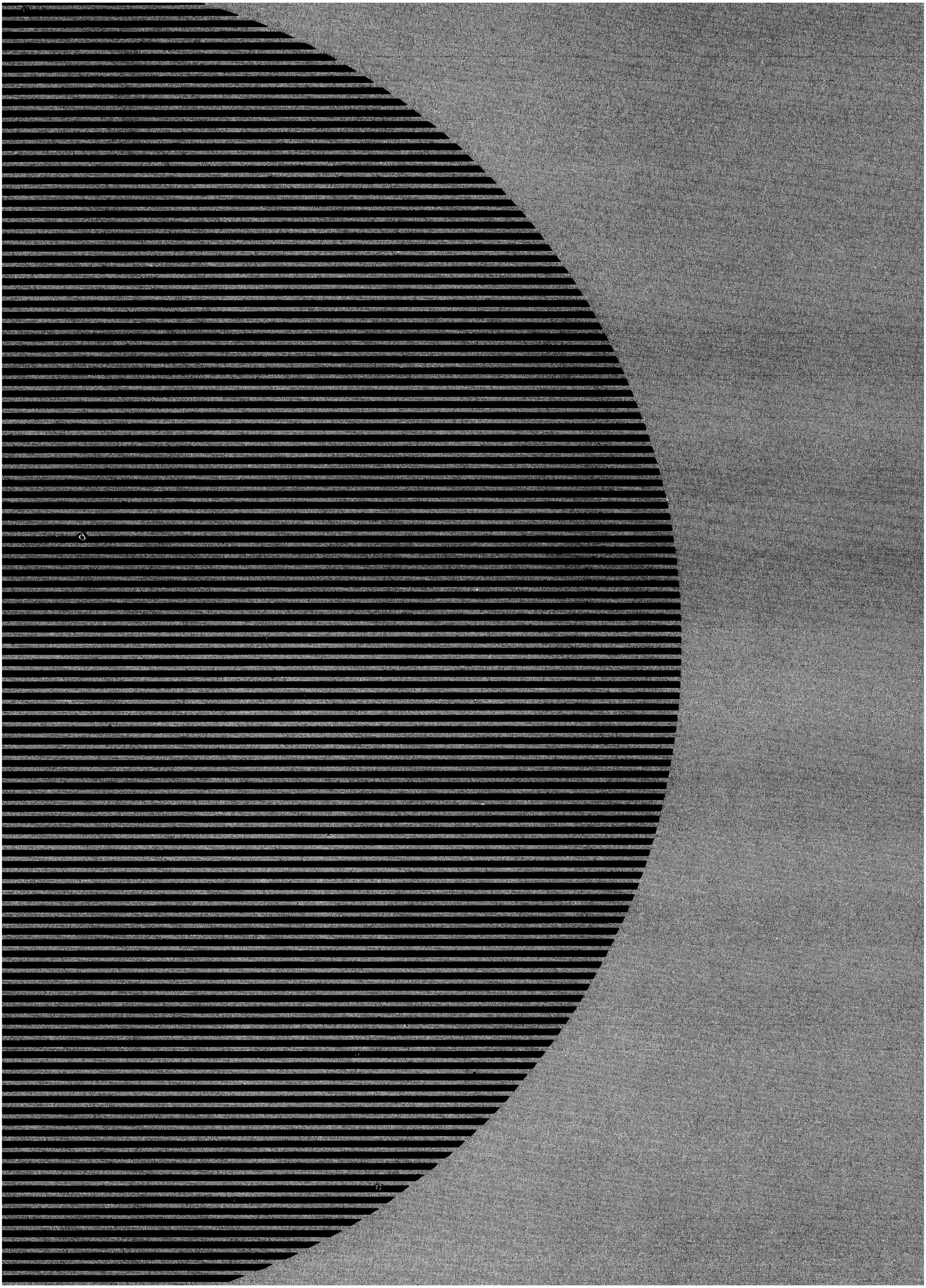
Unrecognized option *option*
Valid options: /B /C<command> /D /E /F /I /M /P /R /S /T /W /43 /2
An invalid switch was entered when starting CodeView.

Usage: cv [options] file [arguments]
An executable file was not specified when starting CodeView.

Video mode changed without /S option
The program changed video modes (either to or from graphics modes) when screen swapping was not specified. Use the /S option to specify screen swapping when debugging graphics programs. Debugging can be continued after receiving this message, but the output screen of the debugged program may be damaged.

Warning: packed file
CodeView was started with a packed file as the executable file. The program cannot be debugged in source mode because all symbolic information is stripped from a file when it is packed with LINK's /EXEPACK option or the EXEPACK utility. Try to debug the program in assembly mode. (The packing routines at the start of the program might make this difficult.)

Wrong number of function arguments
An incorrect number of parameters was specified when evaluating a function in a CodeView expression.



Section V
System Calls

Introduction

All versions of MS-DOS include operating-system services that provide the programmer with hardware-independent tools for handling such tasks as file management, device input and output, memory allocation, and getting and setting system-management information such as the date and time. The majority of these services, collectively called the MS-DOS system calls, are invoked through Interrupt 21H. A few others are called using Interrupts 20H through 27H and 2FH. This section includes descriptions of these system-management services, with details relevant to all releases of MS-DOS through version 3.2.

Use of the Interrupt 21H system calls, rather than hardware-specific routines, helps ensure that a program will run on any computer running an appropriate version of MS-DOS. Likewise, because new releases of MS-DOS attempt to maintain compatibility with earlier versions, use of the calls increases the likelihood that a program will remain usable for more than a single major or minor release of the operating system.

The MS-DOS Interrupt 21H system calls are invoked as follows:

AH	= function number
AL	= subfunction code (if required)
Other registers	= additional function-specific information
Execute Interrupt 21H	

Version Differences

With MS-DOS versions 2.0 and later, considerable overlap occurs in the way in which many system services, such as file and character device I/O, can be carried out. This overlap is a result of the manner in which MS-DOS has developed since it was first released.

The earliest version of MS-DOS, 1.0, included a relatively small set of Interrupt 21H system calls designed primarily for CP/M compatibility. These calls, numbered 00H through 2DH, relied on the use of file control blocks (FCBs) in an application's memory space for information on open files. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management; Appendix G: File Control Block (FCB) Structure. The FCB-based system calls in MS-DOS do not support hierarchical file structures, nor do they support redirection of input and output. As a result, many of these system calls have been superseded in later releases of MS-DOS. The CP/M-style calls are no longer recommended and should not be used unless program compatibility with versions 1.x is required.

Beginning with version 2.0, MS-DOS introduced the concept of handles — 16-bit numbers returned by the operating system after a successful open or create call. The handles can subsequently be used by an application program to reference an open file or device, eliminating redundancy and unnecessary overhead. These handles are also used internally by MS-DOS to keep track of open files and devices. The operating system keeps all such handle-related information in its own memory space. Handles offer full support for the hierarchical file system introduced in version 2.0 of MS-DOS and thus allow the programmer to access any file stored in any directory or subdirectory on a block device. Because of the increased flexibility offered by the handle-related system function calls, these services are recommended over the earlier FCB-based calls, which perform similar tasks but for the current directory only. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

Another advantage of using the system calls introduced in versions 2.0 and later is that these calls set the carry flag when an operational error occurs and return an error code in AX that indicates the nature of the error; the error can then be investigated further by calling Function 59H (Get Extended Error Information). The earlier system calls (00H through 2DH) generally simply return 0FFH (255) in AL to indicate an error or 00H to indicate that the call was completed successfully.

Format of Entries

Entries in this section are arranged in hexadecimal order, with decimal equivalents in parentheses. Each entry is organized as follows:

- Hexadecimal interrupt and/or function number (decimal equivalent in parentheses)
- Interrupt or function name (similar to, but not always the same as, the name used in MS-DOS documentation)
- Version dependencies
- Interrupt or function purpose
- Register contents needed to call
- Register contents on return
- Notes for programmers
- Related functions
- Program example

The format of these entries is designed to give programmers ready reference to specific information, such as register contents, as well as more detailed notes on the use and application of each system call. For further information on the use of the system calls, *see* PROGRAMMING IN THE MS-DOS ENVIRONMENT.

The assembly-language examples in this section use the Cmacros capability introduced with the Windows Software Development Kit. Cmacros, a set of assembly-language macros defined in the file CMACROS.INC, are useful because they provide a simplified interface to the function and segment conventions of high-level languages such as Microsoft C and Microsoft Pascal.

Advantages to using Cmacros for assembly-language programming include transparent support for memory models and symbolic names for function arguments and local variables. Cmacros exist for code and data segment declarations (*sBegin* and *sEnd*), storage allocation (*static*X, *global*X, *extern*X, and *label*X), function declarations (*cProc*, *parm*X, *local*X, *cBegin* and *cEnd*), function calls (*cCall*, *Save*, and *Arg*), special definitions (*Def*X, *RegPtr*, and *FarPtr*), and error control (*errnz* and *errn$*). Of these, only *sBegin*, *sEnd*, *cProc*, *parm*X, *local*X, *cBegin*, and *cEnd* are used in the examples in this section.

Two additional macros that support functions not found in CMACROS.INC are *loadCP* and *loadDP*. These macros, included in the file CMACROSX.INC listed below, allow pointers previously declared with *static*X, *global*X, *parm*X, *Def*X and *local*X to be loaded into registers without regard to the memory model in use — *loadCP* and *loadDP* generate code to load either the offset portion or the full segment:offset of the address, depending on the memory model.

```
;       CMACROSX.INC
;
;       This file includes supplemental macros for two macros included
;       in CMACROS.INC: parmCP and parmDP. When these macros are used,
;       CMACROS.INC allocates either 1 or 2 words to the variables
;       associated with these macros, depending on the memory model in
;       use. However, parmCP and parmDP provide no support for automatically
;       adjusting for different memory models—additional program code
;       needs to be written to compensate for this. The loadCP and loadDP
;       macros included in this file can be used to provide additional
;       flexibility for overcoming this limit.

;       For example, "parmDP pointer" will make space (1 word in small
;       and middle models and 2 words in compact, large, and huge models)
;       for the data pointer named "pointer". The statement
;       "loadDP ds,bx,pointer" can then be used to dynamically place the
;       value of "pointer" into DS:BX, depending on the memory model.
;       In small-model programs, this macro would generate the instruction
;       "mov dx,pointer" (it is assumed that DS already has the right
;       segment value); in large-model programs, this macro would generate
;       the statements "mov ds,SEG_pointer" and "mov dx,OFF_pointer".


checkDS macro        segmt
           diffcount = 0
           irp d,<ds,DS,Ds,dS>                ; Allow for all spellings
              ifdif <segmt>,<d>               ; of "ds".
                 diffcount = diffcount+1
              endif
           endm
           if diffcount EQ 4
              it_is_DS = 0
           else
              it_is_DS = 1
           endif
        endm
```

(more)

```
checkES macro        segmt
           diffcount = 0
           irp d,<es,ES,Es,eS>                  ; Allow for all spellings
              ifdif <segmt>,<d>                 ; of "es".
                 diffcount = diffcount+1
              endif
           endm
           if diffcount EQ 4
              it_is_ES = 0
           else
              it_is_ES = 1
           endif
        endm

loadDP  macro        segmt,offst,dptr
           checkDS segmt
           if sizeD                             ; <-- Large data model
              if it_is_DS
                 lds  offst,dptr
              else
                 checkES segmt
                 if it_is_ES
                    les  offst,dptr
                 else
                    mov  offst,OFF_&dptr
                    mov  segmt,SEG_&dptr
                 endif
              endif
           else
              mov  offst,dptr                   ; <-- Small data model
              if it_is_DS EQ 0
                 push ds                        ; If "segmt" is not DS,
                 pop  segmt                     ; move ds to segmt.
              endif
           endif
        endm

loadCP  macro        segmt,offst,cptr
           if sizeC                             ; <-- Large code model
              checkDS segmt
              if it_is_DS
                 lds offst,cptr
              else
                 checkES
                 if it_is_ES
                    les  offst,cptr
                 else
                    mov  segmt,SEG_&cptr
                    mov  offst,OFF_&cptr
                 endif
              endif
           else
```

(more)

```
                push cs                          ; <-- Small code model
                pop  segmt
                mov  offst,cptr
             endif
          endm
```

The following example program demonstrates the use of Cmacros in an assembly-language program:

```
memS    =       0               ;Small memory model
?PLM    =       0               ;C calling conventions
?WIN    =       0               ;Disable Windows support

include cmacros.inc
include cmacrosx.inc

sBegin  CODE                    ;Start of code segment
assumes CS,CODE                 ;Required by MASM

        ;Microsoft C function syntax:
        ;
        ;     int addnums(firstnum, secondnum)
        ;         int firstnum, secondnum;
        ;
        ;Returns firstnum + secondnum

cProc   addnums,PUBLIC          ;Start of addnums functions
parmW   firstnum                ;Declare parameters
parmW   secondnum
cBegin
        mov     ax,firstnum
        add     ax,secondnum
cEnd
sEnd    CODE
        end
```

A simple C program to call this function would be

```
main()
{
        printf("The sum is %d",addnums(12,33));
}
```

Contents by Functional Group

Although distinguishing between FCB-based and handle-based system calls provides a broad and very generalized means of categorizing these services, the more common and useful approach is to group the calls by the type of task they perform. The following list groups the Interrupt 21H system calls and Interrupts 20H, 22H through 27H, and 2FH by type of service.

Function	Purpose
Character Input	
01H	Character Input with Echo
03H	Auxiliary Input
06H	Direct Console I/O
07H	Unfiltered Character Input Without Echo
08H	Character Input Without Echo
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
Character Output	
02H	Character Output
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
09H	Display String
Disk Management	
0DH	Disk Reset
0EH	Select Disk
19H	Get Current Disk
1BH	Get Default Drive Data
1CH	Get Drive Data
2EH	Set/Reset Verify Flag
36H	Get Disk Free Space
54H	Get Verify Flag
File Management	
0FH	Open File with FCB
10H	Close File with FCB
11H	Find First File
12H	Find Next File
13H	Delete File
16H	Create File with FCB
17H	Rename File
1AH	Set DTA Address
23H	Get File Size
2FH	Get DTA Address
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File

(more)

Function	**Purpose**
File Management *(continued)*	
41H	Delete File
43H	Get/Set File Attributes
45H	Duplicate File Handle
46H	Force Duplicate File Handle
4EH	Find First File
4FH	Find Next File
56H	Rename File
57H	Get/Set Date/Time of File
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region
Information Management	
14H	Sequential Read
15H	Sequential Write
21H	Random Read
22H	Random Write
24H	Set Relative Record
27H	Random Block Read
28H	Random Block Write
3FH	Read File or Device
40H	Write File or Device
42H	Move File Pointer
Interrupt 25H	Absolute Disk Read
Interrupt 26H	Absolute Disk Write
Directory Management	
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
47H	Get Current Directory
Process Management	
00H	Terminate Process
31H	Terminate and Stay Resident
4BH	Load and Execute Program (EXEC)
4CH	Terminate Process with Return Code
4DH	Get Return Code of Child Process
59H	Get Extended Error Information
Interrupt 20H	Terminate Program
Interrupt 27H	Terminate and Stay Resident

(more)

Function	Purpose
Memory Management	
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
58H	Get/Set Allocation Strategy
Miscellaneous System Management	
25H	Set Interrupt Vector
26H	Create New Program Segment Prefix
29H	Parse Filename
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
30H	Get MS-DOS Version Number
33H	Get/Set Control-C Check Flag
34H	Return Address of InDOS Flag
35H	Get Interrupt Vector
38H	Get/Set Current Country
44H	IOCTL
5EH	Network Machine Name/Printer Setup
5FH	Get/Make Assign List Entry
62H	Get Program Segment Prefix Address
63H	Get Lead Byte Table (version 2.25 only)
Interrupt 22H	Terminate Routine Address
Interrupt 23H	Control-C Handler Address
Interrupt 24H	Critical Error Handler Address
Interrupt 2FH	Multiplex Interrupt

Interrupt 20H (32)

1.0 and later

Terminate Program

Interrupt 20H is one of several methods that a program can use to perform a final exit. It informs the operating system that the program is completely finished and that the memory the program occupied can be released.

To Call

CS = segment address of program segment prefix (PSP)

Returns

Nothing

Programmer's Notes

- In response to an Interrupt 20H call, MS-DOS takes the following actions:
  - Restores the termination handler vector (Interrupt 22H) from PSP:000AH.
  - Restores the Control-C vector (Interrupt 23H) from PSP:000EH.
  - With MS-DOS versions 2.0 and later, restores the critical error handler vector (Interrupt 24H) from PSP:0012H.
  - Flushes the file buffers.
  - Transfers to the termination handler address.

  The termination handler releases all memory blocks allocated to the program, including its environment block and any dynamically allocated blocks that were not previously explicitly released; closes any files opened with handles that were not previously closed; and returns control to the parent process (usually COMMAND.COM).
- If the program is returning to COMMAND.COM, control transfers first to COMMAND.COM's resident portion, which reloads COMMAND.COM's transient portion (if necessary) and passes control to it. If a batch file is in progress, the next line of the batch file is then fetched and interpreted; otherwise, a prompt is issued for the next user command.
- Any files that have been written by the program using FCBs should be closed before using Interrupt 20H; otherwise, data may be lost.
- For those programmers who have been with MS-DOS since its earliest incarnations, Interrupt 20H is the traditional way to exit from an application program. However, under versions 2.0 and later, the preferred methods of termination are Interrupt 21H Function 31H (Terminate and Stay Resident) and Interrupt 21H Function 4CH (Terminate Process with Return Code).

Example

```
;**************************************************************;
;                                                              ;
;                     Perform a final exit.                    ;
;                                                              ;
;**************************************************************;
        int     20H     ; Transfer to MS-DOS.
```

Interrupt 21H (33) Function 00H (0)

1.0 and later

Terminate Process

Function 00H flushes all file buffers to disk, terminates the current process, and releases the memory used by the process.

To Call

AH = 00H
CS = segment of program's program segment prefix (PSP)

Returns

Nothing

Programmer's Notes

- The following interrupt vectors are restored from the PSP of the terminated program:

PSP Offset	Vector for Interrupt
0AH	Interrupt 22H (terminate routine)
0EH	Interrupt 23H (Control-C handler)
12H	Interrupt 24H (critical error handler) (versions 2.0 and later.)

- All file buffers are written to disk and all handles are closed. Control is then transferred to Interrupt 22H (Terminate Routine Address).
- Any file that has changed in length and was opened with an FCB should be closed before Function 00H is called. If such a file is not closed, its length, date, and time are not recorded correctly in the directory.
- With versions 3.x of MS-DOS, restoring the default memory-allocation strategy used by MS-DOS is advisable if that strategy has been changed with Function 58H (Get/Set Allocation Strategy). Any global flags, such as the break and verify flags, that affect system behavior and that have been changed by the process should also be restored to their original values.
- Function 00H performs exactly the same processing as Interrupt 20H (Terminate Program).
- Function 00H is obsolete with MS-DOS versions 2.0 and later. Function 31H (Terminate and Stay Resident) and Function 4CH (Terminate Process with Return Code) are preferred; both enable the terminating process to pass a return code to the calling process and do not require that CS contain the PSP address.

Related Functions

31H (Terminate and Stay Resident)
4CH (Terminate Process with Return Code)

Example

None

Interrupt 21H (33) Function 01H (1)

1.0 and later

Character Input with Echo

Function 01H waits for a character from standard input, echoes it to standard output, and returns the character in the AL register.

To Call

AH = 01H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 01H reads input from the keyboard. With versions 2.0 and later, Function 01H reads a character from standard input, which defaults to the keyboard but can be redirected to another device or to a file. Whether or not input has been redirected, the character is echoed to standard output.
- Function 01H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available from standard input, and then calling Function 01H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 01H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- The carriage-return character (0DH) echoes a carriage return but not a linefeed. Likewise, the linefeed character (0AH) does not echo a carriage return.
- With MS-DOS versions 2.0 and later, Function 01H cannot detect an end-of-file condition if input has been redirected.
- Interrupt 23H (Control-C Handler Address) is called if Control-C (03H) is the input character and (with versions 2.0 and later) input is not redirected.
- With MS-DOS version 2.0 and later, if standard input has been redirected to come from a file, Break must be enabled for Interrupt 23H to be called when Control-C (03H) is the input character.
- Alternative character input functions are 06H (Direct Console I/O), 07H (Unfiltered Character Input Without Echo), and 08H (Character Input Without Echo). The four functions are related as follows:

Function	Waits for Input	Echoes to Std Output	Acts on Control-C
01H	yes	yes	yes
06H	no	no	no
07H	yes	no	no
08H	yes	no	yes

Depending on whether Control-C needs to be filtered, Function 06H, 07H, or 08H can be used to handle character display separately from character input.

- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 01H.

Related Functions

06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 01H: Character Input with Echo          ;
        ;                                                            ;
        ;           int read_kbd_echo()                              ;
        ;                                                            ;
        ;           Returns a character from standard input          ;
        ;           after sending it to standard output.             ;
        ;                                                            ;
        ;************************************************************;

cProc   read_kbd_echo,PUBLIC
cBegin
        mov     ah,01h          ; Set function code.
        int     21h             ; Wait for character.
        mov     ah,0            ; Character is in AL, so clear high
                                ; byte.
cEnd
```

Interrupt 21H (33)
Function 02H (2)

1.0 and later

Character Output

Function 02H sends a character to standard output.

To Call

AH = 02H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 02H sends a character to the active display. With MS-DOS versions 2.0 and later, Function 02H sends the character to standard output. By default, the output is sent to the active display, but it can be redirected to another device or to a file.
- With all versions of MS-DOS, displaying a backspace (08H) moves the cursor back one position but does not erase the character at the new position.
- If a Control-C is detected after the character is sent, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 02H.

Related Functions

06H (Direct Console I/O)
09H (Display String)
40H (Write File or Device)

Example

```
;*************************************************************;
;                                                             ;
;                Function 02H: Character Output               ;
;                                                             ;
;                int disp_ch(c)                               ;
;                    char c;                                  ;
;                                                             ;
;                Returns 0.                                   ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   disp_ch,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,02h          ; Set function code.
        int     21h             ; Send character.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 03H (3)

1.0 and later

Auxiliary Input

Function 03H waits for a character from the standard auxiliary device and returns the character in the AL register.

To Call

AH = 03H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 03H reads a character from the first serial port. With versions 2.0 and later, Function 03H reads from the standard auxiliary device (AUX), which defaults to COM1.
- Function 03H waits for input until a character is available from the standard auxiliary device.
- Function 03H is not interrupt driven and does not buffer characters received from the standard auxiliary device. As a result, it may not be fast enough for some telecommunications applications and data may be lost.
- A program cannot perform error detection using Function 03H. On IBM PCs and compatibles, error detection is available through the ROM BIOS Interrupt 14H. Another option is to drive the communications controller directly.
- Function 03H does not ensure that auxiliary input is connected and working, nor does it perform any error checking or set up the auxiliary input device. On IBM PCs and compatibles, the standard auxiliary device, normally COM1, is set to 2400 baud, no parity, 1 stop bit, and 8 databits at startup. These parameters can be changed with the MS-DOS MODE command.
- Some auxiliary input devices do not support 8-bit data transmission. This transmission parameter is a characteristic of the device and the communication parameters to which it is set; it is independent of Function 03H.
- If a Control-C is detected at the console, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device), which handles strings as well as single characters, should be used in preference to Function 03H.

Related Functions

04H (Auxiliary Output)
3FH (Read File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;           Function 03H: Auxiliary Input                     ;
        ;                                                             ;
        ;           int aux_in()                                      ;
        ;                                                             ;
        ;           Returns next character from AUX device.           ;
        ;                                                             ;
        ;*************************************************************;

cProc   aux_in,PUBLIC
cBegin
        mov     ah,03h          ; Set function code.
        int     21h             ; Wait for character from AUX.
        mov     ah,0            ; Character is in AL
                                ; so clear high byte.
cEnd
```

Interrupt 21H (33) Function 04H (4)

1.0 and later

Auxiliary Output

Function 04H sends a character to the standard auxiliary device.

To Call

AH = 04H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 04H sends a character to the first serial port. With versions 2.0 and later, Function 04H sends the character to the standard auxiliary device (AUX), which defaults to COM1.
- Function 04H does not ensure that auxiliary output is connected and working, nor does it perform any error checking or set up the auxiliary output device. On IBM PCs and compatibles, the standard auxiliary device, normally COM1, is set to 2400 baud, no parity, 1 stop bit, and 8 databits at startup. These parameters can be changed with the MS-DOS MODE command.
- Function 04H does not return the status of auxiliary output, nor does it return an error code if the auxiliary output device is not ready for data. If the device is busy, Function 04H waits until it is available.
- Interrupt 23H (Control-C Handler Address) is called if a Control-C is detected at the console.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device), which manages strings as well as single characters, should be used in preference to Function 04H.

Related Functions

03H (Auxiliary Input)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;                Function 04H: Auxiliary Output              ;
        ;                                                            ;
        ;                int aux_out(c)                              ;
        ;                    char c;                                 ;
        ;                                                            ;
        ;                Returns 0.                                  ;
        ;                                                            ;
        ;************************************************************;

cProc   aux_out,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,04h          ; Set function code.
        int     21h             ; Write character to AUX.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33)
Function 05H (5)

1.0 and later

Print Character

Function 05H sends a character to the standard printer.

To Call

AH = 05H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 05H sends a character to the first parallel port (LPT1). With versions 2.0 and later, Function 05H sends the character to the standard printer (PRN), which defaults to LPT1 unless LPT1 has been reassigned with the MS-DOS MODE command. If redirection is in effect, calls to this function send output to the device currently assigned to LPT1.
- Function 05H does not return the status of the standard printer, nor does it return an error code if the standard printer is not ready for characters. If the printer is busy or off line, Function 05H waits until it is available. MS-DOS does, however, perform error checking during the print operation and send any error messages to the standard error device (normally the display).
- If a Control-C is detected at the console, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 05H.

Related Function

40H (Write File or Device)

Example

```
;************************************************************;
;                                                            ;
;               Function 05H: Print Character                ;
;                                                            ;
;               int print_ch(c)                              ;
;                   char c;                                  ;
;                                                            ;
;               Returns 0.                                   ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   print_ch,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,05h          ; Set function code.
        int     21h             ; Write character to standard printer.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 06H (6)

1.0 and later

Direct Console I/O

Function 06H reads a character from standard input or writes a character to standard output.

To Call

AH = 06H

For character input:

DL = FFH

For character output:

DL = 00–FEH (8-bit character code)

Returns

If DL was 0FFH on call and a character was ready:

Zero flag is clear.

AL = 8-bit character code

If DL was 0FFH on call and no character was ready:

Zero flag is set.

Programmer's Notes

- With MS-DOS versions 1.x, Function 06H reads a character from the keyboard or sends a character to the display. With versions 2.0 and later, input and output can be redirected; Function 06H reads from the device currently assigned to standard input or sends to the device currently assigned to standard output.
- Function 06H allows all possible characters and control codes with values between 00H and 0FEH to be read or written with standard input and output and with no filtering by the operating system. The rubout character (0FFH, 255 decimal), however, cannot be output with Function 06H; Function 02H (Character Output) should be used instead.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 06H must be called twice.

With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.

- If Function 06H is an input request and a Control-C is read, the character is returned as any other character would be. Interrupt 23H (Control-C Handler Address) is not called.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) and Function 40H (Write File or Device) should be used in preference to Function 06H.

Related Functions

01H (Character Input with Echo)
02H (Character Output)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
09H (Display String)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)
40H (Write File or Device)

Example

```
        ;***************************************************************;
        ;                                                               ;
        ;          Function 06H: Direct Console I/O                     ;
        ;                                                               ;
        ;          int con_io(c)                                        ;
        ;              char c;                                          ;
        ;                                                               ;
        ;          Returns meaningless data if c is not 0FFH,           ;
        ;          otherwise returns next character from                ;
        ;          standard input.                                      ;
        ;                                                               ;
        ;***************************************************************;

cProc   con_io,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,06h          ; Set function code.
        int     21h             ; This function does NOT wait in
                                ; input case (c = 0FFH)!
        mov     ah,0            ; Return the contents of AL.
cEnd
```

Interrupt 21H (33) Function 07H (7)

1.0 and later

Unfiltered Character Input Without Echo

Function 07H waits for a character from standard input. It does not echo the character to standard output, and it ignores Control-C characters.

To Call

AH = 07H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 07H reads input from the keyboard. With versions 2.0 and later, Function 07H reads a character from standard input. Standard input defaults to the keyboard but can be redirected to another device or to a file.
- Function 07H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available from standard input, and then calling Function 07H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 07H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- Interrupt 23H (Control-C Handler Address) is not called if a Control-C is read. Function 07H simply passes the character back through the AL register. If Control-C checking is required, Function 08H (Character Input Without Echo) should be used instead.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 07H.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;         Function 07H: Unfiltered Character Input           ;
        ;                       Without Echo                         ;
        ;                                                            ;
        ;         int con_in()                                       ;
        ;                                                            ;
        ;         Returns next character from standard input.        ;
        ;                                                            ;
        ;************************************************************;

cProc   con_in,PUBLIC
cBegin
        mov     ah,07h          ; Set function code.
        int     21h             ; Wait for character, no echo.
        mov     ah,0            ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 08H (8)

1.0 and later

Character Input Without Echo

Function 08H waits for a character from standard input. The character is not echoed to standard output.

To Call

AH = 08H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 08H reads input from the keyboard. With versions 2.0 and later, Function 08H reads a character from standard input. Standard input defaults to the keyboard but can be redirected to another device or to a file.
- Function 08H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available, and then calling Function 08H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 08H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A process can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- If a Control-C is read and (with versions 2.0 and later) input has not been redirected, Interrupt 23H (Control-C Handler Address) is called. To read the Control-C character as data, Function 07H (Unfiltered Character Input Without Echo) should be used.
- Interrupt 23H (Control-C Handler Address) is called if Control-C is the input character, Break is enabled, and (with versions 2.0 and later) standard input has been redirected to come from a file.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 08H.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;    Function 08H:  Unfiltered Character Input Without Echo  ;
        ;                                                            ;
        ;    int read_kbd()                                          ;
        ;                                                            ;
        ;    Returns next character from standard input.             ;
        ;                                                            ;
        ;************************************************************;

cProc   read_kbd,PUBLIC
cBegin
        mov     ah,08h          ; Set function code.
        int     21h             ; Wait for character, no echo.
        mov     ah,0            ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 09H (9)

1.0 and later

Display String

Function 09H sends a string of characters to standard output. The string must end with the dollar-sign character ($). All characters up to, but not including, the $ are displayed.

To Call

AH	= 09H
DS:DX	= segment:offset of string to display

Returns

Nothing

Programmer's Notes

- With MS-DOS versions 1.x, Function 09H sends the string to the display. With versions 2.0 and later, the string is written to standard output. By default, standard output is sent to the display, but it can be redirected to another device or to a file.
- The string can include any valid ASCII characters, including control codes. Sending a dollar sign with this function, however, is not possible.
- Depending on the device currently serving as standard output, characters other than the normally displayable ASCII characters (20H to 7FH) may or may not be displayed. On IBM PCs and most compatibles, extensions to the displayable ASCII character set (character codes 80H to FFH) appear as foreign or graphics characters.
- Display begins at the current cursor position on standard output. After the string is completely displayed, the cursor position is updated to the location immediately following the string.

  On IBM PCs and compatibles, if the end of a line is reached before the string is completely displayed, a carriage return and linefeed are issued and the next character is displayed in the first position of the following line. If the cursor reaches the bottom right corner of the display before the complete string has been sent, the display is scrolled up one line.
- Control characters are often included in the string to be sent. The following sample fragment of code contains carriage returns and linefeeds:

```
msg     db      'Resident part of TSR.COM installed'
        db      0dh, 0ah
        db      'Copyright (c) 19xx Foo Software, Inc.'
        db      0dh, 0ah, 0ah, 0ah
        db      '$'
```

- If a Control-C is detected, Interrupt 23H (Control-C Handler Address) is called.

- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 09H.

Related Functions

02H (Character Output)
06H (Direct Console I/O)
40H (Write File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;                 Function 09H: Display String                ;
        ;                                                             ;
        ;                 int disp_str(pstr)                          ;
        ;                     char *pstr;                             ;
        ;                                                             ;
        ;                 Returns 0.                                  ;
        ;                                                             ;
        ;*************************************************************;

cProc   disp_str,PUBLIC,<ds,di>
parmDP  pstr
cBegin
        loadDP  ds,dx,pstr      ; DS:DX = pointer to string.
        mov     ax,0900h        ; Prepare to write dollar-terminated
                                ; string to standard output, but
                                ; first replace the 0 at the end of
                                ; the string with '$'.
        push    ds              ; Set ES equal to DS.
        pop     es              ; (MS-C does not require ES to be
                                ; saved.)
        mov     di,dx           ; ES:DI points at string.
        mov     cx,0ffffh       ; Allow string to be 64KB long.
        repne   scasb           ; Look for 0 at end of string.
        dec     di              ; Scasb search always goes 1 byte too
                                ; far.
        mov     byte ptr [di],'$' ; Replace 0 with dollar sign.
        int     21h             ; Have MS-DOS print string.
        mov     [di],al         ; Restore 0 terminator.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0AH (10)

1.0 and later

Buffered Keyboard Input

Function 0AH collects characters from standard input and places them in a user-specified memory buffer. Input is accepted until either a carriage return (0DH) is encountered or the buffer is filled to one character less than its capacity. The characters are echoed to standard output.

To Call

AH	= 0AH
DS:DX	= segment:offset of input buffer

Returns

Nothing

Programmer's Notes

- With MS-DOS versions 1.x, Function 0AH reads a string from the keyboard. With versions 2.0 and later, calls to this function read a string from standard input, which defaults to the keyboard but can be redirected to another device or to a file. The MS-DOS editing keys are active during input with this function.
- The buffer pointed to by DS:DX must have the following format:

Byte	**Contents**
0	Maximum number of characters to read (1–255); this value must be set by the process before Function 0AH is called.
1	Count of characters read (does not include the carriage return); this value is set by Function 0AH before returning to the process.
2–(*n*+2)	Actual string of characters read, including the carriage return; *n* = number of bytes read.

- The first byte of the buffer must contain the maximum number of characters the program will accept, including the carriage return at the end. Because the last byte must be a carriage return, the maximum number of bytes this function will actually read is 254. The carriage return is not included in the character count returned by MS-DOS in the second byte of the buffer.
- If the buffer fills to 1 byte less than its capacity, succeeding characters are ignored and a beep is sounded for each keypress until a carriage return is received.
- If a Control-C is detected and (with versions 2.0 and later) input has not been redirected, Interrupt 23H (Control-C Handler Address) is called.
- With versions 2.0 and later, if standard input has been redirected to come from a file, Break must be enabled for Interrupt 23H (Control-C Handler Address) to be called when Control-C is the input character.

- With MS-DOS versions 2.0 and later, if input is redirected, an end-of-file condition goes undetected by Function 0AH.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;        Function 0AH: Buffered Keyboard Input                ;
        ;                                                             ;
        ;        int read_str(pbuf,len)                               ;
        ;            char *pbuf;                                      ;
        ;            int len;                                         ;
        ;                                                             ;
        ;        Returns number of bytes read into buffer.            ;
        ;                                                             ;
        ;        Note: pbuf must be at least len+3 bytes long.        ;
        ;                                                             ;
        ;*************************************************************;

cProc   read_str,PUBLIC,<ds,di>
parmDP  pbuf
parmB   len
cBegin
        loadDP  ds,dx,pbuf      ; DS:DX = pointer to buffer.
        mov     al,len          ; AL = len.
        inc     al              ; Add 1 to allow for CR in buf.
        mov     di,dx
        mov     [di],al         ; Store max length into buffer.
        mov     ah,0ah          ; Set function code.
        int     21h             ; Ask MS-DOS to read string.
        mov     al,[di+1]       ; Return number of characters read.
        mov     ah,0
        mov     bx,ax
        mov     [bx+di+2],ah    ; Store 0 at end of buffer.
cEnd
```

Interrupt 21H (33) Function 0BH (11)

1.0 and later

Check Keyboard Status

Function 0BH returns a value in AL that indicates whether a character is available from standard input.

To Call

AH = 0BH

Returns

AL	= 00H	no character available
	FFH	one or more characters available

Programmer's Notes

- With MS-DOS versions 1.x, Function 0BH checks the type-ahead buffer for a character. With versions 2.0 and later, if input has been redirected, Function 0BH checks standard input for a character. If input has not been redirected, the function checks the type-ahead buffer.
- Function 0BH does not indicate how many characters are available; it merely indicates whether at least one character is available.
- If the available character is Control-C, Interrupt 23H (Control-C Handler Address) is called.
- Function 0BH does not remove characters from standard input. Thus, if a character is present, repeated calls return 0FFH in AL until all characters in the buffer are read, either with one of the character-input functions (01H, 06H, 07H, 08H, or 0AH) or with Function 3FH (Read File or Device) using the handle for standard input (0).

Related Functions

06H (Direct Console I/O)
44H Subfunction 06H (IOCTL: Check Input Status)

Example

```
;************************************************************;
;                                                            ;
;             Function 0BH: Check Keyboard Status            ;
;                                                            ;
;             int key_ready()                                ;
;                                                            ;
;             Returns 1 if key is ready, 0 if not.           ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   key_ready,PUBLIC
cBegin
        mov     ah,0bh          ; Set function code.
        int     21h             ; Ask MS-DOS if key is available.
        and     ax,0001h        ; Keep least significant bit only.
cEnd
```

Interrupt 21H (33) Function 0CH (12)

1.0 and later

Flush Buffer, Read Keyboard

Function 0CH clears the standard-input buffer and then performs one of the other keyboard input functions (01H, 06H, 07H, 08H, 0AH).

To Call

AH	= 0CH
AL	= input function number to execute

If AL is 06H:

DL	= FFH

If AL is 0AH:

DS:DX	= segment:offset of buffer to receive input

Returns

If AL was 01H, 06H, 07H, or 08H on call:

AL	= 8-bit ASCII character from standard input

If AL was 0AH on call:

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 0CH empties the type-ahead buffer before executing the input function specified in AL. With versions 2.0 and later, if input has been redirected to a file, Function 0CH does nothing before carrying out the input function specified in AL; if input was not redirected, the type-ahead buffer is flushed.
- A function number other than 01H, 06H, 07H, 08H, or 0AH in AL simply flushes the standard-input buffer and returns control to the calling program.
- If AL contains 0AH, DS:DX must point to the buffer in which MS-DOS is to place the string read from the keyboard.
- Because the buffer is flushed before the input function is carried out, any Control-C characters pending in the buffer are discarded. If subsequent input is a Control-C, however, Interrupt 23H (Control-C Handler Address) is called if (in versions 2.0 and later) standard input has not been redirected to come from a file.
- With versions 2.0 and later, if standard input has been redirected to come from a file and, after the buffer is flushed, subsequent input is a Control-C character, Interrupt 23H (Control-C handler address) is called only if Break is enabled.
- This function exists to defeat the type-ahead feature if necessary—for example, to obtain input at a critical prompt the user may not have anticipated.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 0CH: Flush Buffer, Read Keyboard         ;
        ;                                                            ;
        ;          int flush_kbd()                                   ;
        ;                                                            ;
        ;          Returns 0.                                        ;
        ;                                                            ;
        ;************************************************************;

cProc   flush_kbd,PUBLIC
cBegin
        mov     ax,0c00h        ; Just flush type-ahead buffer.
        int     21h             ; Call MS-DOS.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0DH (13)

1.0 and later

Disk Reset

Function 0DH writes to disk all internal MS-DOS file buffers in memory that have been modified since the last write. All buffers are then marked as "free."

To Call

AH = 0DH

Returns

Nothing

Programmer's Notes

- Function 0DH ensures that the information stored on disk matches changes made by write requests to file buffers in memory.
- Function 0DH does not update the disk directory. The application must issue Function 10H (Close File with FCB) or Function 3EH (Close File) to update directory information correctly.
- Function 0DH should be part of Control-C interrupt-handling routines so that the system is left in a known state when an application is terminated.
- Disk Reset calls can be issued after particularly important disk write calls, such as transactions in an accounting application. Repeated use of this function, however, degrades system performance by defeating the MS-DOS buffering scheme.

Related Functions

10H (Close File with FCB)
3EH (Close File)

Example

```
;************************************************************;
;                                                            ;
;                   Function 0DH: Disk Reset                 ;
;                                                            ;
;                   int reset_disk()                         ;
;                                                            ;
;                   Returns 0.                               ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   reset_disk,PUBLIC
cBegin
        mov     ah,0dh          ; Set function code.
        int     21h             ; Ask MS-DOS to write all dirty file
                                ; buffers to the disk.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0EH (14)

1.0 and later

Select Disk

Function 0EH sets the default disk drive to the drive specified in the DL register. The default is the disk drive MS-DOS chooses for file access when a filename is specified without a drive designator. A successful call to this function returns the number of logical (not physical) drives in the system.

To Call

AH = 0EH
DL = drive number (0 = drive A, 1 = drive B, 2 = drive C, and so on)

Returns

AL = number of logical drives in the system

Programmer's Notes

- The value used as a drive number is the ASCII value of the uppercase drive letter minus the ASCII value of the uppercase letter A (41H); thus, 0 = drive A, 1 = drive B, and so on.
- A logical drive is defined as any block-oriented device; this category includes floppy-disk drives, RAMdisks, tape devices, fixed disks (which can be partitioned into more than one logical drive), and network drives.
- The maximum numbers of drive designators available for each MS-DOS version are as follows:

MS-DOS Version	Number of Designators	Values
1.x	16	0 through 0FH
2.x	63	0 through 3FH
3.x	26	0 through 19H

  Drive letters should be limited to A through P (0 through 0FH) to ensure that an application runs on all versions of MS-DOS.
- With versions of MS-DOS earlier than 3.0 running on IBM PCs and compatibles with one floppy-disk drive, Function 0EH returns 02H as the drive count, because the single physical drive is equivalent to the two logical drives A and B. MS-DOS versions 3.0 and later return a minimum value of 05H in AL.
- On IBM PCs and compatibles, the number of physical floppy-disk drives in a system can be obtained from the ROM BIOS with Interrupt 11H (Equipment Determination).

Related Function

19H (Get Current Disk)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;     Function 0EH: Select Disk                               ;
        ;                                                             ;
        ;     int select_drive(drive_ltr)                             ;
        ;         char drive_ltr;                                     ;
        ;                                                             ;
        ;     Returns number of logical drives present in system.     ;
        ;                                                             ;
        ;*************************************************************;

cProc   select_drive,PUBLIC
parmB   drive_ltr
cBegin
        mov     dl,drive_ltr     ; Get new drive letter.
        and     dl,not 20h      ; Make sure letter is uppercase.
        sub     dl,'A'          ; Convert drive letter to number,
                                ; 'A' = 0, 'B' = 1, etc.
        mov     ah,0eh          ; Set function code.
        int     21h             ; Ask MS-DOS to set default drive.
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 0FH (15)

1.0 and later

Open File with FCB

Function 0FH opens the file named in the file control block (FCB) pointed to by DS:DX.

To Call

AH	= 0FH
DS:DX	= segment:offset of an unopened FCB

Returns

If function is successful:

AL	= 00H

If function is not successful:

AL	= FFH

Programmer's Notes

- MS-DOS provides several types of file services: FCB file services, which are relatively compatible with the CP/M methods of file handling; extended FCB file services, which take advantage of both CP/M compatibility and MS-DOS extensions; and handle, or "stream-oriented," file services, which are more compatible with UNIX/XENIX and support pathnames (MS-DOS versions 2.0 and later).
- Function 0FH does not support pathnames and so is capable of opening files only in the current directory of the specified drive.
- Function 0FH does not create a new file if the specified file does not already exist. Function 16H (Create File with FCB) is used to create new files with FCBs.
- Function 0FH must use an unopened FCB — that is, one in which all but the drive-designator, filename, and extension fields are zero. If the call is successful, the function fills in the file size and date fields from the file's directory entry. In MS-DOS versions 2.0 and later, the function also fills in the time field.
- If the file is opened on the default drive (the drive number in the FCB is set to 0), MS-DOS fills in the actual drive code. Thus, at some later point in processing, the default drive can be changed and MS-DOS will still have the drive number in the FCB for use in accessing the file. It will therefore continue to use the correct drive.
- If Function 0FH is successful, MS-DOS sets the current-block field to 0; that is, the file pointer is at the beginning of the file. It also sets the record size to 128 bytes (the system default).
- If a record size other than 128 is needed, the record size field of the FCB should be changed after the file is successfully opened and before attempting any I/O.

- In a network running under MS-DOS version 3.1 or later, files are opened by Function 0FH with the share code set to compatibility mode and the access code set to read/write.
- If Function 0FH returns an error code (0FFH) in the AL register, the attempt to open the file was not successful. Possible causes for the failure are
  - File was not found.
  - File has the hidden or system attribute and a properly formatted extended FCB was not used.
  - Filename was improperly specified in the FCB.
  - SHARE is loaded and the file is already open by another process in a mode other than compatibility mode.
- With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to determine why the attempt to open the file failed.
- MS-DOS passes the first two command-tail parameters into default FCBs located at offsets 5CH and 6CH in the program segment prefix (PSP). Many applications designed to run as .COM files take advantage of one or both of these default FCBs.
- With MS-DOS versions 2.0 and later, Function 3DH (Open File with Handle) should be used in preference to Function 0FH.

Related Functions

10H (Close File with FCB)
16H (Create File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
3EH (Close File)
59H (Get Extended Error Information)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
;************************************************************;
;                                                            ;
;       Function 0FH: Open File, FCB-based                   ;
;                                                            ;
;       int FCB_open(uXFCB,recsize)                          ;
;           char *uXFCB;                                     ;
;           int recsize;                                     ;
;                                                            ;
;       Returns 0 if file opened OK, otherwise returns -1.   ;
;                                                            ;
;       Note: uXFCB must have the drive and filename         ;
;       fields (bytes 07H through 12H) and the extension     ;
;       flag (byte 00H) set before the call to FCB_open      ;
;       (see Function 29H).                                  ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   FCB_open,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB     ; Pointer to unopened extended FCB.
        mov     ah,0fh           ; Ask MS-DOS to open an existing file.
        int     21h
        add     dx,7             ; Advance pointer to start of regular
                                 ; FCB.
        mov     bx,dx            ; BX = FCB pointer.
        mov     dx,recsize       ; Get record size parameter.
        mov     [bx+0eh],dx      ; Store record size in FCB.
        xor     dx,dx
        mov     [bx+20h],dl      ; Set current-record
        mov     [bx+21h],dx      ; and relative-record
        mov     [bx+23h],dx      ; fields to 0.
        cbw                      ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 10H (16)

1.0 and later

Close File with FCB

Function 10H flushes file-related information to disk, closes the file named in the file control block (FCB) pointed to by DS:DX, and updates the file's directory entry.

To Call

AH	= 10H
DS:DX	= segment:offset of previously opened FCB

Returns

If function is successful:

AL	= 00H

If function is not successful:

AL	= FFH

Programmer's Notes

- A successful call to Function 10H flushes to disk all MS-DOS internal buffers associated with the file and updates the directory entry and file allocation table (FAT). The function thus ensures that correct information is contained in the copy of the file on disk.
- Because MS-DOS versions 1.x and 2.x do not always detect a disk change, an error can occur if the user changes disks between the time the file is opened and the time it is closed. In the worst case, the FAT and the directory of the newly inserted disk may be damaged.
- With MS-DOS versions 2.0 and later, Function 3EH (Close File) should be used in preference to Function 10H.

Related Functions

0FH (Open File with FCB)
3EH (Close File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 10H: Close file, FCB-based              ;
        ;                                                            ;
        ;           int FCB_close(oXFCB)                             ;
        ;               char *oXFCB;                                 ;
        ;                                                            ;
        ;           Returns 0 if file closed OK, otherwise           ;
        ;           returns -1.                                      ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_close,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,10h          ; Ask MS-DOS to close file.
        int     21h
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 11H (17)

1.0 and later

Find First File

Function 11H searches the current directory for the first file that matches a specified name and extension.

To Call

AH	= 11H
DS:DX	= segment:offset of unopened file control block (FCB)

Returns

If function is successful:

AL	= 00H

Disk transfer area (DTA) contains unopened FCB of same type (normal or extended) as search FCB.

If function is not successful:

AL	= FFH

Programmer's Notes

- If necessary, Function 1AH (Set DTA Address) should be used before Function 11H is called, to set the location of the DTA in which the results of the search will be placed.
- With MS-DOS versions 1.0 and later, the wildcard character ? is allowed in the filename. With MS-DOS versions 3.0 and later, both wildcard characters (? and *) are allowed in filenames. Pathnames are not supported.
- With MS-DOS versions 2.0 and later, the attribute field of an extended FCB can be used to search for files with the hidden, system, subdirectory, or volume-label attributes. In such a search, specifying either the normal (00H) or volume-label (08H) attribute restricts MS-DOS to files with the given attribute. Specifying any combination of the hidden (02H), system (04H), and subdirectory (10H) attributes, however, causes MS-DOS to search both for normal files and for those that match the specified attributes.
- For a normal FCB, Function 11H places the drive number in the first byte of the DTA and fills the succeeding 32 bytes with the directory entry.

  For an extended FCB, Function 11H fills in the first 7 bytes of the DTA as follows: the first byte contains 0FFH, indicating an extended FCB; the second through sixth bytes contain 00H, as required by MS-DOS; the seventh byte contains the value of the attribute byte in the search FCB. The next 33 bytes contain the drive number and directory information, as for a normal FCB.

- As with other FCB functions, the number 0 can be used to indicate the default drive. MS-DOS fills in the actual drive number and continues to use that drive for calls to Function 12H (Find Next File) that use the same FCB, regardless of any subsequent selection of a different default drive.
- The FCB with the initial file specifications must remain unmodified if Function 12H is used to continue the search.
- Error reporting in Function 11H is incomplete. An error return (0FFH in the AL register) does not always mean that the file does not exist. Other possibilities include
  - Filename in the FCB was improperly specified.
  - If an extended FCB was used, no files match the attributes given.

  With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about the error.
- With MS-DOS versions 2.0 and later, Functions 4EH (Find First File) and 4FH (Find Next File) should be used in preference to Functions 11H and 12H.

Related Functions

12H (Find Next File)
1AH (Set DTA Address)
4EH (Find First File)
4FH (Find Next File)

Example

```
;**************************************************************;
;                                                              ;
;       Function 11H: Find First File, FCB-based               ;
;                                                              ;
;       int FCB_first(puXFCB,attrib)                           ;
;           char *puXFCB;                                      ;
;           char  attrib;                                      ;
;                                                              ;
;       Returns 0 if match found, otherwise returns -1.        ;
;                                                              ;
;       Note: The FCB must have the drive and                  ;
;       filename fields (bytes 07H through 12H) and            ;
;       the extension flag (byte 00H) set before               ;
;       the call to FCB_first (see Function 29H).              ;
;                                                              ;
;**************************************************************;
```

(more)

```
cProc   FCB_first,PUBLIC,ds
parmDP  puXFCB
parmB   attrib
cBegin
        loadDP  ds,dx,puXFCB     ; Pointer to unopened extended FCB.
        mov     bx,dx            ; BX points at FCB, too.
        mov     al,attrib        ; Get search attribute.
        mov     [bx+6],al        ; Put attribute into extended FCB
                                 ; area.
        mov     byte ptr [bx],0ffh ; Set flag for extended FCB.
        mov     ah,11h           ; Ask MS-DOS to find 1st matching
                                 ; file in current directory.
        int     21h              ; If match found, directory entry can
                                 ; be found at DTA address.
        cbw                      ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 12H (18)

1.0 and later

Find Next File

Function 12H searches the current directory for the next file that matches a specified filename and extension. The function assumes a previous successful call to Function 11H (Find First File) with the same file control block (FCB).

To Call

AH	= 12H
DS:DX	= segment:offset of search FCB

Returns

If function is successful:

AL	= 00H

Disk transfer area (DTA) contains unopened FCB of same type (normal or extended) as search FCB.

If function is not successful:

AL	= FFH

Programmer's Notes

- Function 12H assumes that a successful call to Function 11H (Find First File) has been completed with the same FCB. The FCB specifies the search pattern. This function also assumes that the wildcard character ? appears at least once in the filename or extension specified.
- An error (indicated by 0FFH returned in register AL) does not necessarily mean that a file matching the file specification does not exist in the current directory. MS-DOS relies on certain information that appears in the search FCB initialized by Function 11H, so it is important not to alter that FCB either between calls to Functions 11H and 12H or between subsequent calls to Function 12H.
- If drive code 0 (the default drive) was used in the call to Function 11H, MS-DOS has already filled in the actual drive number for the current directory. MS-DOS continues to use that drive for all calls to Function 12H that use the same FCB, regardless of the default drive in effect at the time of the call.
- With MS-DOS versions 2.0 and later, Functions 4EH (Find First File) and 4FH (Find Next File) should be used in preference to Functions 11H and 12H.

Related Functions

11H (Find First File)
1AH (Set DTA Address)
4EH (Find First File)
4FH (Find Next File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;       Function 12H: Find Next File, FCB-based              ;
        ;                                                            ;
        ;       int FCB_next(puXFCB)                                 ;
        ;           char *puXFCB;                                    ;
        ;                                                            ;
        ;       Returns 0 if match found, otherwise returns -1.      ;
        ;                                                            ;
        ;       Note: The FCB must have the drive and                ;
        ;       filename fields (bytes 07H through 12H) and          ;
        ;       the extension flag (byte 00H) set before             ;
        ;       the call to FCB_next (see Function 29H).             ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_next,PUBLIC,ds
parmDP  puXFCB
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,12h          ; Ask MS-DOS to find next matching
                                ; file in current directory.
        int     21h             ; If match found, directory entry can
                                ; be found at DTA address.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 13H (19)

1.0 and later

Delete File

Function 13H deletes all files matching a specified name and extension from the current directory.

To Call

AH	= 13H
DS:DX	= segment:offset of an unopened file control block (FCB)

Returns

If function is successful:

AL	= 00H

If function is not successful:

AL	= FFH

Programmer's Notes

- The wildcard character ? can be used to match any character or sequence of characters in specifying the filename and extension.
- Open files must not be deleted.
- Function 13H does not support pathnames.
- An error (indicated by 0FFH returned in register AL) does not necessarily mean that the filename specified does not exist in the current directory. Other possible causes for an error include
  - Filename in the FCB is improperly specified.
  - File is a read-only, hidden, or system file and an extended FCB with the appropriate attribute byte was not used.
  - Program attempted to delete a volume label and the label does not exist or a properly formatted extended FCB was not used.
  - In networking environments, file is locked or access rights are insufficient for deletion.
- MS-DOS removes file allocation table (FAT) mapping for the file or files deleted by this function and flushes the FAT to disk to ensure that the disk contains a correct table. The first character of the filename in the directory entry is replaced by the value 0E5H, indicating a deleted file.
- Because the function does not physically erase data, use of Function 13H alone is not sufficient in security-critical applications that strictly prohibit viewing the data.

- On networks running under MS-DOS versions 3.1 and later, the user must have Create access rights to the directory containing the file to be deleted.
- Because Function 13H deletes all files matching a given file specification, a conservative approach is to use a combination of Functions 11H (Find First File) and 12H (Find Next File) to build a list of files matching the file specification and then obtain confirmation from the user before deleting the files in the list.
- With MS-DOS versions 2.0 and later, Function 41H (Delete File) should be used in preference to Function 13H.

Related Function

41H (Delete File)

Example

```
                ;***********************************************************;
                ;                                                           ;
                ;       Function 13H: Delete File(s), FCB-based             ;
                ;                                                           ;
                ;       int FCB_delete(uXFCB)                               ;
                ;           char *uXFCB;                                    ;
                ;                                                           ;
                ;       Returns 0 if file(s) were deleted OK, otherwise     ;
                ;       returns -1.                                         ;
                ;                                                           ;
                ;       Note: uXFCB must have the drive and                 ;
                ;       filename fields (bytes 07H through 12H) and         ;
                ;       the extension flag (byte 00H) set before            ;
                ;       the call to FCB_delete (see Function 29H).          ;
                ;                                                           ;
                ;***********************************************************;

cProc   FCB_delete,PUBLIC,ds
parmDP  puXFCB
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,13h          ; Ask MS-DOS to delete file(s).
        int     21h
        cbw                     ; Return value of 0 or -1.
cEnd
```

Interrupt 21H (33) Function 14H (20)

1.0 and later

Sequential Read

Function 14H reads the next sequential block of data from a file and places the data in the current disk transfer area (DTA).

To Call

AH	= 14H
DS:DX	= segment:offset of a previously opened file control block (FCB)

Returns

AL	= 00H	read successful
	01H	end of file encountered; no data in record
	02H	DTA too small (segment wrap error); read canceled
	03H	end of file; partial record read

If AL = 00H or 03H:

DTA contains data read from file.

Programmer's Notes

- If necessary, Function 1AH (Set DTA Address) should be used to set the base address of the DTA before Function 14H is called. The default DTA is 128 bytes and is located at offset 80H of the program segment prefix (PSP). If record sizes larger than 128 bytes will be used, the program must change the DTA address to point to a buffer of adequate size.
- The read process begins at the current position in the file. When the read is complete, Function 14H increments the current-block and current-record fields of the FCB.
- The size of the record loaded into the DTA is specified in the record size field of the FCB. The default is 128 bytes, set by Function 0FH (Open File with FCB) or Function 16H (Create File with FCB). If the record size is not 128 bytes, the application must set the record size correctly before issuing any reads.
- Function 0FH does not fill in the current-record field of the FCB when opening a file, so this field must be explicitly set (usually to zero) before the first call to Function 14H. The record pointer, which includes the current-block and current-record fields of the FCB, is incremented when Function 14H is successfully completed.
- Function 14H deals with fixed-length records only. Buffering logic must be added to an application if variable-length records are to be manipulated.
- The block of data to be read can be chosen by changing the current-block and current-record fields of the FCB.

- Partial records read at the end of a file are padded with zeros to the requested record length.
- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 14H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
21H (Random Read)
27H (Random Block Read)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 14H: Sequential Read, FCB-based         ;
        ;                                                            ;
        ;           int FCB_sread(oXFCB)                             ;
        ;               char *oXFCB;                                 ;
        ;                                                            ;
        ;           Returns 0 if record read OK, otherwise           ;
        ;           returns error code 1, 2, or 3.                   ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_sread,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,14h          ; Ask MS-DOS to read next record,
                                ; placing it at DTA.
        int     21h
        cbw                     ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 15H (21)

1.0 and later

Sequential Write

Function 15H writes the next sequential block of data from the disk transfer area (DTA) to a specified file.

To Call

AH	= 15H
DS:DX	= segment:offset of a previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	block written successfully
	01H	disk full; write canceled
	02H	DTA too small (segment wrap error); write canceled

Programmer's Notes

- If necessary, the calling process should set the DTA address with Function 1AH (Set DTA Address) to point to the data to be written before issuing a call to Function 15H. The default address of the DTA is offset 80H in the program segment prefix (PSP).
- The FCB must already have been filled in by a call to Function 0FH (Open File with FCB) before Function 15H is called.
- The location of the block to be written is given by the current-block and current-record fields of the FCB. If the write is successful, Function 15H increments the current-block and current-record fields.
- The size of the record written by Function 15H is determined by the value in the record size field of the FCB. The default value is 128, set by Function 0FH (Open File with FCB) or Function 16H (Create File with FCB). A process must set the record size in the FCB correctly before issuing any writes.
- Function 15H deals with fixed-length records only. Buffering logic must be added to an application if variable-length records are to be manipulated.
- Function 15H performs a logical, but not necessarily physical, write operation. If less than one sector is being written, MS-DOS moves the record from the DTA to an appropriate MS-DOS internal buffer. When a full sector of data has been buffered, MS-DOS flushes the buffer to disk. Function 0DH (Disk Reset) or Function 10H (Close File with FCB) can be used to flush data to disk before a full sector is buffered.
- On networks running under MS-DOS versions 3.1 and later, the user must have Write access to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 15H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
22H (Random Write)
28H (Random Block Write)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 15H: Sequential Write, FCB-based         ;
        ;                                                            ;
        ;          int FCB_swrite(oXFCB)                             ;
        ;              char *oXFCB;                                  ;
        ;                                                            ;
        ;          Returns 0 if record read OK, otherwise            ;
        ;          returns error code 1 or 2.                        ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_swrite,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,15h          ; Ask MS-DOS to write next record
                                ; from DTA to disk file.
        int     21h
        cbw                     ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 16H (22)

1.0 and later

Create File with FCB

Function 16H creates a directory entry in the current directory for a specified file and opens the file for use. If the file already exists, it is opened and truncated to zero length.

To Call

AH	= 16H
DS:DX	= segment:offset of an unopened file control block (FCB)

Returns

If function is successful:

AL	= 00H

If function is not successful:

AL	= FFH

Programmer's Notes

- Before creating a new directory entry for the specified file, Function 16H searches the current directory for a matching filename. If a match is found, the existing file is opened, but its length is set to 0. In effect, this action erases an existing file and replaces it with a new, empty file of the same name.

  If a matching filename is not found and the directory has room for a new entry, the file is created and opened, and its length is set to 0.
- An extended file control block (FCB) can be used to create a file with a special attribute, such as hidden. Before the Create File call is issued, the attribute byte must be set appropriately.
- A value of 0FFH returned in the AL register can indicate one of several errors:
  - Filename was improperly specified in the FCB.
  - File with the same name exists but is a read-only, hidden, system, or (in MS-DOS versions 3.x and networks) locked file.
  - Disk is full.
  - Current working directory is the root directory, and it is full.
  - User does not have the appropriate access rights to create a file in this directory (in MS-DOS versions 3.x and networks).

  With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about an error.
- Upon successful completion of Function 16H, MS-DOS has
  - Created and opened the file specified in the FCB.

- Filled in the date and time fields of the FCB with the current date and time.
- Set file size to zero.

All other changes made to the FCB are similar to those made by Function 0FH (Open File with FCB).

- Pathnames and wildcard characters (? and *) are not supported by Function 16H.
- With MS-DOS versions 2.0 and later, Function 16H has been superseded by Functions 3CH (Create File with Handle), 5AH (Create Temporary File), and 5BH (Create New File).

Related Functions

0FH (Open File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;         Function 16H: Create File, FCB-based               ;
        ;                                                            ;
        ;         int FCB_create(uXFCB,recsize)                      ;
        ;             char *uXFCB;                                   ;
        ;             int recsize;                                   ;
        ;                                                            ;
        ;         Returns 0 if file created OK, otherwise            ;
        ;         returns -1.                                        ;
        ;                                                            ;
        ;         Note: uXFCB must have the drive and filename       ;
        ;         fields (bytes 07H through 12H) and the             ;
        ;         extension flag (byte 00H) set before the           ;
        ;         call to FCB_create (see Function 29H).             ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_create,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,16h          ; Ask MS-DOS to create file.
        int     21h
        add     dx,7            ; Advance pointer to start of regular
                                ; FCB.
        mov     bx,dx           ; BX = FCB pointer.
        mov     dx,recsize      ; Get record size parameter.
        mov     [bx+0eh],dx     ; Store record size in FCB.
        xor     dx,dx
        mov     [bx+20h],dl     ; Set current-record
        mov     [bx+21h],dx     ; and relative-record
        mov     [bx+23h],dx     ; fields to 0.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 17H (23)

1.0 and later

Rename File

Function 17H renames one or more files in the current directory.

To Call

AH	= 17H
DS:DX	= segment:offset of modified file control block (FCB) in the following nonstandard format:

Byte(s)	Contents
00H	Drive number
01–08H	Old filename (padded with blanks, if necessary)
09–0BH	Old file extension (padded with blanks, if necessary)
0CH–10H	Zeroed out
11H–18H	New filename (padded with blanks, if necessary)
19H–1BH	New file extension (padded with blanks, if necessary)
11CH–24H	Zeroed out

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- The wildcard character ? can be used in specifying both the old and the new filenames, but its meaning differs in each case. A wildcard character in the old filename matches any single character or sequence of characters in the directory entry. A wildcard character in the new filename, however, indicates that the corresponding character or characters in the original filename are not to change.
- With MS-DOS versions 2.0 and later, Function 17H views subdirectory entries as files. These subdirectory entries can be renamed using this function and an extended FCB with the appropriate attribute byte.
- A value of 0FFH returned in the AL register can indicate one of several errors:
  - Old filename is improperly specified in the FCB.
  - File with the new filename already exists in the current directory.

  - Old file is a read-only file.
  - With MS-DOS versions 3.1 and later in a networking environment, the user has insufficient access rights to the directory.

  With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about the cause of an error.

- With MS-DOS versions 2.0 and later, Function 56H (Rename File) should be used in preference to Function 17H.

Related Function

56H (Rename File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 17H: Rename File(s), FCB-based           ;
        ;                                                            ;
        ;          int FCB_rename(uXFCBold,uXFCBnew)                 ;
        ;              char *uXFCBold,*uXFCBnew;                     ;
        ;                                                            ;
        ;          Returns 0 if file(s) renamed OK, otherwise        ;
        ;          returns -1.                                       ;
        ;                                                            ;
        ;          Note: Both uXFCB's must have the drive and        ;
        ;          filename fields (bytes 07H through 12H) and       ;
        ;          the extension flag (byte 00H) set before          ;
        ;          the call to FCB_rename (see Function 29H).        ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_rename,PUBLIC,<ds,si,di>
parmDP  puXFCBold
parmDP  puXFCBnew
cBegin
        loadDP  es,di,puXFCBold ; ES:DI = Pointer to uXFCBold.
        mov     dx,di           ; Save offset in DX.
        add     di,7            ; Advance pointer to start of regular
                                ; FCBold.
        loadDP  ds,si,puXFCBnew ; DS:SI = Pointer to uXFCBnew.
        add     si,8            ; Advance pointer to filename field
                                ; FCBnew.
                                ; Copy name from FCBnew into FCBold
                                ; at offset 11H:
        add     di,11h          ; DI points 11H bytes into old FCB.
        mov     cx,0bh          ; Copy 0BH bytes, moving new
        rep     movsb           ; name into old FCB.
        push    es              ; Set DS to segment of FCBold.
        pop     ds
        mov     ah,17h          ; Ask MS-DOS to rename old
        int     21h             ; file(s) to new name(s).
        cbw                     ; Set return flag to 0 or -1.
cEnd
```

Interrupt 21H (33)
Function 19H (25)

1.0 and later

Get Current Disk

Function 19H returns the code for the current disk drive.

To Call

AH = 19H

Returns

AL = drive code (0 = drive A, 1 = drive B, 2 = drive C, and so on)

Programmer's Note

- The drive code returned by Function 19H is zero-based, meaning that drive A = 0, drive B = 1, and so on. This value is unlike the drive code used in file control blocks (FCBs) and in some other MS-DOS functions, such as 1CH (Get Drive Data) and 36H (Get Disk Free Space), in which 0 indicates the default rather than the current drive.

Related Function

0EH (Select Disk)

Example

```
              ;************************************************************;
              ;                                                            ;
              ;          Function 19H: Get Current Disk                    ;
              ;                                                            ;
              ;          int cur_drive()                                   ;
              ;                                                            ;
              ;          Returns letter of current "logged" disk.          ;
              ;                                                            ;
              ;************************************************************;

cProc   cur_drive,PUBLIC
cBegin
        mov     ah,19h          ; Set function code.
        int     21h             ; Get number of logged disk.
        add     al,'A'          ; Convert number to letter.
        cbw                     ; Clear the high byte of return value.
cEnd
```

Interrupt 21H (33) Function 1AH (26)

1.0 and later

Set DTA Address

Function 1AH specifies the location of the disk transfer area (DTA) to be used for file control block (FCB) disk I/O operations.

To Call

AH	= 1AH
DS:DX	= segment:offset of DTA

Returns

Nothing

Programmer's Notes

- If an application does not specify a disk transfer area, MS-DOS uses a default buffer at offset 80H in the program segment prefix (PSP).
- The DTA specified must be large enough to accommodate the amount of data to be transferred in a single block. The default record size for FCB file operations is 128 bytes; this value can be changed after a file is successfully opened or created by altering the record size field in the FCB. If the DTA is too small for the record size used by the program, other code or data may be damaged.
- The location of the DTA must be far enough from the top of the segment that contains it to avoid errors caused by segment wrap (data wrapping from the end of the segment to the beginning), which will cause the disk transfer to be terminated. Thus, for example, if records of 128 bytes are to be read, the highest location acceptable for the DTA is DS:FF80H.
- The DTA is used by all FCB-based read and write functions. In addition, any application using the following functions must also set up a DTA for use as a scratch area in directory searches:
  - 11H (Find First File)
  - 12H (Find Next File)
  - 4EH (Find First File)
  - 4FH (Find Next File)

Related Function

2FH (Get DTA Address)

Example

```
                ;************************************************************;
                ;                                                            ;
                ;                Function 1AH: Set DTA Address               ;
                ;                                                            ;
                ;                int set_DTA(pDTAbuffer)                     ;
                ;                    char far *pDTAbuffer;                   ;
                ;                                                            ;
                ;                Returns 0.                                  ;
                ;                                                            ;
                ;************************************************************;

        cProc   set_DTA,PUBLIC,ds
        parmD   pDTAbuffer
        cBegin
                lds     dx,pDTAbuffer   ; DS:DX = pointer to buffer.
                mov     ah,1ah          ; Set function code.
                int     21h             ; Ask MS-DOS to change DTA address.
                xor     ax,ax           ; Return 0.
        cEnd
```

Interrupt 21H (33) Function 1BH (27)

1.0 and later

Get Default Drive Data

Function 1BH returns information about the disk in the default drive.

To Call

AH	= 1BH

Returns

If function is successful:

AL	= number of sectors per cluster (allocation unit)
CX	= number of bytes per sector
DX	= number of clusters
DS:BX	= segment:offset of the file allocation table (FAT) identification byte

If function is not successful:

AL	= FFH

Programmer's Notes

- If Function 1BH returns 0FFH in the AL register, the current drive was invalid or a disk error occurred. The most likely causes of the latter are
  - Drive door was open.
  - Disk was not ready.
  - Medium was bad.
  - Disk was unformatted.

  If any of these situations arises, MS-DOS issues Interrupt 24H (critical error). If Interrupt 24H has not been revectored to a critical error handler controlled by the program and the user responds *Ignore* to the MS-DOS *Abort, Retry, Ignore?* message, the error code 0FFH is returned to the program. An application should check the AL register for a value of 0FFH before assuming it has information on the default drive.
- Possible values of the FAT ID byte (for IBM-compatible media) are the following:

Value	Medium
0FFH	Double-sided, 8 sectors/track, 40 tracks/side
0FEH	Single-sided, 8 sectors/track, 40 tracks/side
0FDH	Double-sided, 9 sectors/track, 40 tracks/side
0FCH	Single-sided, 9 sectors/track, 40 tracks/side

(more)

Value	Medium
0F9H	Double-sided, 15 sectors/track, 40 tracks/side or double-sided, 9 sectors/ track, 80 tracks/side
0F8H	Fixed disk
0F0H	Others

- With MS-DOS versions 1.x, Function 1BH returns a pointer in DS:BX for the actual memory image of the FAT. In MS-DOS versions 2.0 and later, the function returns a pointer in DS:BX for a copy of the FAT identification byte; the contents of memory beyond the identification byte are not necessarily the FAT memory image. If access to the FAT is necessary, Interrupt 25H (Absolute Disk Read) can be used to read it into memory.
- The FAT ID byte is not enough to identify a drive completely in MS-DOS versions 2.0 and later. In these versions of MS-DOS, Function 36H (Get Disk Free Space) should be used in preference to Function 1BH to avoid the ambiguity caused by the FAT identification byte.
- With MS-DOS versions 3.2 and later, additional drive information can be obtained by inspecting the BIOS parameter block (BPB) obtained with Function 44H (IOCTL) Subfunction 0DH (Generic I/O Control for Block Devices) minor code 60H (Get Device Parameters).
- With MS-DOS versions 2.0 and later, Function 1CH (Get Drive Data) provides the same types of information as Function 1BH, but for a disk in a drive other than the default drive.

Related Functions

1CH (Get Drive Data)
36H (Get Disk Free Space)
44H (IOCTL)

Example

See SYSTEM CALLS: INTERRUPT 21H: Function ICH.

Interrupt 21H (33)
Function 1CH (28)

2.0 and later

Get Drive Data

Function 1CH returns information about the disk in a specified drive.

To Call

AH	= 1CH
DL	= drive code (0 = default drive, 1 = drive A, 2 = drive B, 3 = drive C, and so on)

Returns

If function is successful:

AL	= number of sectors per cluster (allocation unit)
CX	= number of bytes per sector
DX	= number of clusters
DS:BX	= segment:offset of the file allocation table (FAT) identification byte

If function is not successful:

AL	= FFH

Programmer's Notes

- Function 1CH is not available with MS-DOS versions 1.x.
- If the function returns 0FFH in the AL register, the drive code was invalid or a disk error occurred. The most likely causes of the latter are
  - Drive door was open.
  - Disk was not ready.
  - Medium was bad.
  - Disk was unformatted.

  If any of these situations arises, MS-DOS issues Interrupt 24H (critical error). If Interrupt 24H has not been revectored to a critical error handler controlled by the program and the user responds *Ignore* to the MS-DOS *Abort, Retry, Ignore?* message, the error code 0FFH is returned to the program. An application should check the AL register for a value of 0FFH before assuming it has information on the specified drive.
- Possible values of the FAT ID byte (for IBM-compatible media) are the following:

Value	**Medium**
0FFH	Double-sided, 8 sectors/track, 40 tracks/side
0FEH	Single-sided, 8 sectors/track, 40 tracks/side

(more)

Value	Medium
0FDH	Double-sided, 9 sectors/track, 40 tracks/side
0FCH	Single-sided, 9 sectors/track, 40 tracks/side
0F9H	Double-sided, 15 sectors/track, 40 tracks/side or double-sided, 9 sectors/track, 80 tracks/side
0F8H	Fixed disk
0F0H	Others

- The contents of memory beyond the identification byte pointed to by DS:BX are not necessarily the FAT memory image. If access to the FAT is necessary, Interrupt 25H (Absolute Disk Read) can be used to read it into memory.
- The FAT ID byte is not enough to identify a drive completely. To avoid the ambiguity caused by the FAT identification byte, Function 36H (Get Disk Free Space) should be used in preference to Function 1CH.
- With MS-DOS versions 3.2 and later, additional drive information can be obtained by inspecting the BIOS parameter block (BPB) obtained with Function 44H (IOCTL) Subfunction 0DH (Generic I/O Control for Block Devices) minor code 60H (Get Device Parameters).

Related Functions

1BH (Get Default Drive Data)
36H (Get Disk Free Space)
44H (IOCTL)

Example

```
;************************************************************;
;                                                            ;
;   Function 1CH: Get Drive Data                             ;
;                                                            ;
;   Get information about the disk in the specified          ;
;   drive.  Set drive_ltr to binary 0 for default drive info. ;
;                                                            ;
;   int get_drive_data(drive_ltr,                            ;
;          pbytes_per_sector,                                ;
;          psectors_per_cluster,                             ;
;          pclusters_per_drive)                              ;
;        int  drive_ltr;                                     ;
;        int *pbytes_per_sector;                             ;
;        int *psectors_per_cluster;                          ;
;        int *pclusters_per_drive;                           ;
;                                                            ;
;   Returns -1 for invalid drive, otherwise returns          ;
;   the disk's type (from the 1st byte of the FAT).          ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   get_drive_data,PUBLIC,<ds,si>
parmB   drive_ltr
parmDP  pbytes_per_sector
parmDP  psectors_per_cluster
parmDP  pclusters_per_drive
cBegin
        mov     si,ds           ; Save DS in SI to use later.
        mov     dl,drive_ltr    ; Get drive letter.
        or      dl,dl           ; Leave 0 alone.
        jz      gdd
        and     dl,not 20h      ; Convert letter to uppercase.
        sub     dl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
gdd:
        mov     ah,1ch          ; Set function code.
        int     21h             ; Ask MS-DOS for data.
        cbw                     ; Extend AL into AH.
        cmp     al,0ffh         ; Bad drive letter?
        je      gddx            ; If so, exit with error code -1.
        mov     bl,[bx]         ; Get FAT ID byte from DS:BX.
        mov     ds,si           ; Get back original DS.
        loadDP  ds,si,pbytes_per_sector
        mov     [si],cx         ; Return bytes per sector.
        loadDP  ds,si,psectors_per_cluster
        mov     ah,0
        mov     [si],ax         ; Return sectors per cluster.
        loadDP  ds,si,pclusters_per_drive
        mov     [si],dx         ; Return clusters per drive.
        mov     al,bl           ; Return FAT ID byte.
gddx:
cEnd
```

Interrupt 21H (33)
Function 21H (33)

1.0 and later

Random Read

Function 21H reads a selected record from disk into memory.

To Call

AH	= 21H
DS:DX	= segment:offset of previously opened file control block (FCB)

Returns

AL	= 00H	record read successfully
	01H	end of file; no record read
	02H	DTA too small (segment wrap error); read canceled
	03H	end of file; partial record transferred

If AL = 00H or 03H:

DTA contains data read from file.

Programmer's Notes

- Function 21H reads the record into the current disk transfer area (DTA). Unless the 128-byte default DTA (at offset 80H in the program segment prefix) is adequate, Function 1AH (Set DTA Address) should be used to set the DTA address before Function 21H is called. The program must ensure that the buffer pointed to by the DTA address is large enough to hold the records to be transferred.
- The relative-record field in the FCB must be set to the record number to be read. Numbering begins with record 00H; thus, the value 06H in the relative-record field would indicate the seventh record, not the sixth.
- Function 21H sets the current-block and current-record fields to match the relative-record field before transferring the data to the DTA.
- Unlike Function 27H (Random Block Read), Function 21H does not increment the current-block, current-record, or relative-record fields.
- The record length read is determined by the record size field of the FCB.
- If a partial record is read and the end of file is encountered, the remainder of the record is filled out to the requested length with zero bytes.
- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 21H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
22H (Random Write)
24H (Set Relative Record)
27H (Random Block Read)
3FH (Read File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;          Function 21H: Random File Read, FCB-based          ;
        ;                                                             ;
        ;          int FCB_rread(oXFCB,recnum)                        ;
        ;              char *oXFCB;                                   ;
        ;              long recnum;                                   ;
        ;                                                             ;
        ;          Returns 0 if record read OK, otherwise             ;
        ;          returns error code 1, 2, or 3.                     ;
        ;                                                             ;
        ;*************************************************************;

cProc   FCB_rread,PUBLIC,ds
parmDP  poXFCB
parmD   recnum
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     bx,dx           ; BX points at FCB, too.
        mov     ax,word ptr (recnum)    ; Get low 16 bits of record
        mov     [bx+28h],ax             ; number and store in FCB.
        mov     ax,word ptr (recnum+2)  ; Get high 16 bits of record
        mov     [bx+2ah],ax             ; number and store in FCB.
        mov     ah,21h          ; Ask MS-DOS to read recnum'th
                                ; record, placing it at DTA.
        int     21h
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 22H (34)

1.0 and later

Random Write

Function 22H writes data from the current disk transfer area (DTA) to a specified record location in a file.

To Call

AH	= 22H
DS:DX	= segment:offset of previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	record written successfully
	01H	disk full
	02H	DTA too small (segment wrap error); write canceled

Programmer's Notes

- Before calling Function 22H, the program must set the disk transfer area (DTA) address appropriately with a call to Function 1AH (Set DTA Address), if necessary, and place the data to be written in the DTA.
- The relative-record field in the FCB must be set to the record number that is to be written. Numbering begins with record 00H; thus, the value 06H in the relative-record field would indicate the seventh record, not the sixth.
- Function 22H sets the current-block and current-record fields to match the relative-record field before writing the data from the DTA.
- Unlike Function 28H (Random Block Write), Function 22H does not increment the current-block, current-record, or relative-record fields.
- The record size field determines the record length written by the function.
- If a record is written beyond the current end of file, the data between the old end of file and the beginning of the new record is uninitialized.
- The file that is written to cannot have the read-only attribute.
- Information is written logically, but not always physically, to disk at the time Function 22H is called. The contents of the DTA are written immediately to disk only if they constitute a sector's worth of information. If less than a sector is written, it is transferred from the DTA to an MS-DOS buffer and is not physically written to disk until one of the following occurs:
  - A full sector of information is ready.
  - The file is closed.
  - Function 0DH (Disk Reset) is issued.

- On networks running under MS-DOS version 3.1 or later, the user must have Write access rights to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 22H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
21H (Random Read)
24H (Set Relative Record)
28H (Random Block Write)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 22H: Random File Write, FCB-based        ;
        ;                                                            ;
        ;          int FCB_rwrite(oXFCB,recnum)                      ;
        ;              char *oXFCB;                                  ;
        ;              long recnum;                                  ;
        ;                                                            ;
        ;          Returns 0 if record read OK, otherwise            ;
        ;          returns error code 1 or 2.                        ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_rwrite,PUBLIC,ds
parmDP  poXFCB
parmD   recnum
cBegin
        loadDP  ds,dx,poXFCB      ; Pointer to opened extended FCB.
        mov     bx,dx             ; BX points at FCB, too.
        mov     ax,word ptr (recnum)    ; Get low 16 bits of record
        mov     [bx+28h],ax             ; number and store in FCB.
        mov     ax,word ptr (recnum+2)  ; Get high 16 bits of record
        mov     [bx+2ah],ax             ; number and store in FCB.
        mov     ah,22h            ; Ask MS-DOS to write DTA to
        int     21h               ; recnum'th record of file.
        cbw                       ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 23H (35)

1.0 and later

Get File Size

Function 23H searches the current directory for a specified file and returns the size of the file in records.

To Call

AH	= 23H
DS:DX	= segment:offset of unopened file control block (FCB) with record size field set appropriately

Returns

If function is successful:

AL	= 00H

FCB relative-record field contains number of records, rounded upward if necessary.

If function is not successful:

AL	= FFH

Programmer's Notes

- The record size field in the FCB can be set to 1 to find the number of bytes in the file.
- The number of records is the file size divided by the record size. If there is a remainder, the record count is rounded upward. The result stored in the relative-record field may, therefore, contain a value that is 1 larger than the number of complete records in the file.
- Because record numbers are zero based and this function returns the number of records in a file in the relative-record field of the FCB, Function 23H can be used to position the file pointer to the end of file.
- With MS-DOS versions 2.0 and later, Function 42H (Move File Pointer) should be used in preference to Function 23H.

Related Function

42H (Move File Pointer)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;      Function 23H: Get File Size, FCB-based                ;
        ;                                                            ;
        ;      long FCB_nrecs(uXFCB,recsize)                         ;
        ;           char *uXFCB;                                     ;
        ;           int recsize;                                     ;
        ;                                                            ;
        ;      Returns a long -1 if file not found, otherwise        ;
        ;      returns the number of records of size recsize.        ;
        ;                                                            ;
        ;      Note: uXFCB must have the drive and                   ;
        ;      filename fields (bytes 07H through 12H) and           ;
        ;      the extension flag (byte 00H) set before              ;
        ;      the call to FCB_nrecs (see Function 29H).             ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_nrecs,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     bx,dx           ; Copy FCB pointer into BX.
        mov     ax,recsize      ; Get record size
        mov     [bx+15h],ax     ; and store it in FCB.
        mov     ah,23h          ; Ask MS-DOS for file size (in
                                ; records).
        int     21h
        cbw                     ; If AL = 0FFH, set AX to -1.
        cwd                     ; Extend to long.
        or      dx,dx           ; Is DX negative?
        js      nr_exit         ; If so, exit with error flag.
        mov     [bx+2bh],al     ; Only low 24 bits of the relative-
                                ; record field are used, so clear the
                                ; top 8 bits.
        mov     ax,[bx+28h]     ; Return file length in DX:AX.
        mov     dx,[bx+2ah]
nr_exit:
cEnd
```

Interrupt 21H (33)
Function 24H (36)

1.0 and later

Set Relative Record

Function 24H sets the relative-record field of a file control block (FCB) to match the file position indicated by the current-block and current-record fields of the same FCB.

To Call

AH	= 24H
DS:DX	= segment:offset of previously opened FCB

Returns

AL	= 00H

Relative-record field is modified in FCB.

Programmer's Notes

- The AL register is always set to 00H by Function 24H. Thus, any preexisting information in the AL register is lost.
- Before Function 24H is called, the program must open the FCB with Function 0FH (Open File with FCB) or with Function 16H (Create File with FCB).
- The entire relative-record field (4 bytes) of the FCB must be initialized to zeros before calling Function 24H. If this is not done, any value in the high-order byte of the high-order word remaining from previous reads or writes might not be overwritten and the resulting relative-record number will be invalid.
- Function 24H is normally used in changing from sequential to random I/O. Sequential I/O, performed by Functions 14H (Sequential Read) and 15H (Sequential Write), sets the current-block and current-record fields of the FCB. Random I/O uses the relative-record field, which is set by Function 24H to match the current file position as recorded in the current-block and current-record fields.

  After the file pointer is set, any of the following functions can be used to access data at the record pointed to by the relative-record field:
  - 21H (Random Read)
  - 22H (Random Write)
  - 27H (Random Block Read)
  - 28H (Random Block Write)
- With MS-DOS versions 2.0 and later, Function 42H (Move File Pointer) should be used in preference to Function 24H.

Related Function

42H (Move File Pointer)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;             Function 24H: Set Relative Record              ;
        ;                                                            ;
        ;             int FCB_set_rrec(oXFCB)                        ;
        ;                 char *oXFCB;                               ;
        ;                                                            ;
        ;             Returns 0.                                     ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_set_rrec,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB     ; Pointer to opened extended FCB.
        mov     bx,dx            ; BX points at FCB, too.
        mov     byte ptr [bx+2bh],0 ; Zero high byte of high word of
                                    ; relative-record field.
        mov     ah,24h           ; Ask MS-DOS to set relative record
                                 ; to current record.
        int     21h
        xor     ax,ax            ; Return 0.
cEnd
```

Interrupt 21H (33) Function 25H (37)

1.0 and later

Set Interrupt Vector

Function 25H sets an address in the interrupt vector table to point to a specified interrupt handler.

To Call

AH	= 25H
AL	= interrupt number
DS:DX	= segment:offset of interrupt handler

Returns

Nothing

Programmer's Notes

- When Function 25H is called, the 4-byte address in DS:DX is placed in the correct position in the interrupt vector table.
- Function 25H is the recommended method for initializing or changing an interrupt vector. A vector in the interrupt vector table should never be changed directly.
- Before Function 25H is used to change an interrupt vector, the address of the current interrupt handler should be read with Function 35H (Get Interrupt Vector) and then saved for restoration before the program terminates.

Related Function

35H (Get Interrupt Vector)

Example

```
;*************************************************************;
;                                                             ;
;              Function 25H: Set Interrupt Vector             ;
;                                                             ;
;              typedef void (far *FCP)();                     ;
;              int set_vector(intnum,vector)                  ;
;                  int intnum;                                ;
;                  FCP vector;                                ;
;                                                             ;
;              Returns 0.                                     ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   set_vector,PUBLIC,ds
parmB   intnum
parmD   vector
cBegin
        lds     dx,vector       ; Get vector segment:offset into
                                ; DS:DX.
        mov     al,intnum       ; Get interrupt number into AL.
        mov     ah,25h          ; Select "set vector" function.
        int     21h             ; Ask MS-DOS to change vector.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 26H (38)

1.0 and later

Create New Program Segment Prefix

Function 26H creates a new program segment prefix (PSP) at a specified segment address.

To Call

AH = 26H
DX = segment address of the PSP to create

Returns

Nothing

Programmer's Notes

- Function 26H copies the current PSP to the address indicated by DX. Note that DX contains a segment address, not an absolute address.
- After the copy is made, the memory size information located at offset 06H in the new PSP is adjusted to match the amount of memory available to the new PSP. In addition, the current contents of the interrupt vectors for Interrupt 22H (Terminate Routine Address), Interrupt 23H (Control-C Handler Address), and Interrupt 24H (Critical Error Handler Address) are saved starting at offset 0AH of the new PSP.
- A .COM file can be loaded into memory immediately after the new PSP and execution can begin at that location. A .EXE file cannot be loaded and executed in this manner.
- With MS-DOS versions 2.0 and later, Function 4BH (Load and Execute Program) should be used in preference to Function 26H. Function 4BH can be used to load .COM files, .EXE files, or overlays.

Related Function

4BH (Load and Execute Program)

Example

```
;************************************************************;
;                                                            ;
;       Function 26H: Create New Program Segment Prefix      ;
;                                                            ;
;       int create_psp(pspseg)                               ;
;           int  pspseg;                                     ;
;                                                            ;
;       Returns 0.                                           ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   create_psp,PUBLIC
parmW   pspseg
cBegin
        mov     dx,pspseg       ; Get segment address of new PSP.
        mov     ah,26h          ; Set function code.
        int     21h             ; Ask MS-DOS to create new PSP.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 27H (39)

1.0 and later

Random Block Read

Function 27H reads one or more records into memory, placing the records in the current disk transfer area (DTA).

To Call

AH	= 27H
CX	= number of records to read
DS:DX	= segment:offset of previously opened file control block (FCB)

Returns

AL	= 00H	read successful
	01H	end of file; no record read
	02H	DTA too small (segment wrap error); no record read
	03H	end of file; partial record read

If AL is 00H or 03H:

CX = number of records read

DTA contains data read from file.

Programmer's Notes

- The DTA address should be set with Function 1AH (Set DTA Address) before Function 27H is called. If the DTA address has not been set, MS-DOS uses a default 128-byte DTA at offset 80H in the program segment prefix (PSP).
- Function 27H reads the number of records specified in CX sequentially, starting at the file location indicated by the relative-record and record size fields in the FCB. If CX = 0, no records are read.
- The record length used by Function 27H is the value in the record size field of the FCB. Unless a new value is placed in this field after a file is opened or created, MS-DOS uses a default record length of 128 bytes.
- Function 27H is similar to Function 21H (Random Read); however, Function 27H can read more than one record at a time and updates the relative-record field of the FCB after each call. Successive calls to this function thus read sequential groups of records from a file, whereas successive calls to Function 21H repeatedly read the same record.
- Possible alternative causes for end-of-file (01H) errors include
  - Disk removed from drive since file was opened.
  - Previous open failed.

  With MS-DOS versions 3.0 and later, more detailed information on the error can be obtained by calling Function 59H (Get Extended Error Information).

- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 27H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
21H (Random Read)
24H (Set Relative Record)
28H (Random Block Write)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;       Function 27H: Random File Block Read, FCB-based      ;
        ;                                                            ;
        ;       int FCB_rblock(oXFCB,nrequest,nactual,start)         ;
        ;           char *oXFCB;                                     ;
        ;           int   nrequest;                                  ;
        ;           int  *nactual;                                   ;
        ;           long  start;                                     ;
        ;                                                            ;
        ;       Returns read status 0, 1, 2, or 3 and sets           ;
        ;       nactual to number of records actually read.          ;
        ;                                                            ;
        ;       If start is -1, the relative-record field is         ;
        ;       not changed, causing the block to be read starting   ;
        ;       at the current record.                               ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_rblock,PUBLIC,<ds,di>
parmDP  poXFCB
parmW   nrequest
parmDP  pnactual
parmD   start
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     di,dx           ; DI points at FCB, too.
        mov     ax,word ptr (start) ; Get long value of start.
        mov     bx,word ptr (start+2)
        mov     cx,ax           ; Is start = -1?
        and     cx,bx
        inc     cx
        jcxz    rb_skip         ; If so, don't change relative-record
                                ; field.
        mov     [di+28h],ax     ; Otherwise, seek to start record.
```

(more)

```
        mov     [di+2ah],bx
rb_skip:
        mov     cx,nrequest     ; CX = number of records to read.
        mov     ah,27h          ; Get MS-DOS to read CX records,
        int     21h             ; placing them at DTA.
        loadDP  ds,bx,pnactual  ; DS:BX = address of nactual.
        mov     [bx],cx         ; Return number of records read.
        cbw                     ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 28H (40)

1.0 and later

Random Block Write

Function 28H writes one or more records from the current disk transfer area (DTA) to a file.

To Call

AH	= 28H
CX	= number of records to write
DS:DX	= segment:offset of previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	write successful
	01H	disk full
	02H	DTA too small (segment wrap error); write canceled

If AL is 00H or 01H:

CX	= number of records written

Programmer's Notes

- Data to be written must be placed in the DTA before Function 28H is called. Unless the DTA address has been set with Function 1AH (Set DTA Address), MS-DOS uses a default 128-byte DTA at offset 80H in the program segment prefix (PSP).
- Function 28H writes the number of records indicated in CX, beginning at the location specified in the relative-record field of the file control block (FCB). If Function 28H is called with CX = 0, the file is truncated or extended to the size indicated by the record-size and relative-record fields of the FCB.
- The record length used by Function 28H is the value in the record size field of the FCB. Unless a new value is assigned after a file is opened or created, MS-DOS uses a default record length of 128 bytes.
- Function 28H is similar to Function 22H (Random Write); however, Function 28H can write more than one record at a time and updates the relative-record field of the FCB after each call. Successive calls to this function thus write sequential groups of records to a file, whereas successive calls to Function 22H repeatedly write the same record.

- Possible alternative causes for disk full (01H) errors include
  – Disk removed from drive since file was opened.
  – Previous open failed.

  In MS-DOS versions 3.0 and later, more detailed information on the error can be obtained by calling Function 59H (Get Extended Error Information).
- Information is written logically, but not always physically, to disk at the time Function 28H is called. The contents of the DTA are written immediately to disk only if they constitute a full sector of information. If less than a sector is written, it is transferred from the DTA to an MS-DOS buffer and is not physically written to disk until one of the following occurs:
  – A full sector of information is ready.
  – The file is closed.
  – Function 0DH (Disk Reset) is issued.
- On networks running under MS-DOS version 3.1 or later, the user must have Write access rights to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 28H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
22H (Random Write)
24H (Set Relative Record)
27H (Random Block Read)
40H (Write File or Device)

Example

```
;**************************************************************;
;                                                              ;
;       Function 28H: Random File Block Write, FCB-based       ;
;                                                              ;
;       int FCB_wblock(oXFCB,nrequest,nactual,start)           ;
;           char *oXFCB;                                       ;
;           int   nrequest;                                    ;
;           int  *nactual;                                     ;
;           long  start;                                       ;
;                                                              ;
;       Returns write status of 0, 1, or 2 and sets            ;
;       nactual to number of records actually written.         ;
;                                                              ;
;       If start is -1, the relative-record field is           ;
;       not changed, causing the block to be written           ;
;       starting at the current record.                        ;
;                                                              ;
;**************************************************************;
```

(more)

```
cProc   FCB_wblock,PUBLIC,<ds,di>
parmDP  poXFCB
parmW   nrequest
parmDP  pnactual
parmD   start
cBegin
        loadDP  ds,dx,poXFCB     ; Pointer to opened extended FCB.
        mov     di,dx            ; DI points at FCB, too.
        mov     ax,word ptr (start) ; Get long value of start.
        mov     bx,word ptr (start+2)
        mov     cx,ax            ; Is start = -1?
        and     cx,bx
        inc     cx
        jcxz    wb_skip          ; If so, don't change relative-record
                                 ; field.
        mov     [di+28h],ax      ; Otherwise, seek to start record.
        mov     [di+2ah],bx
wb_skip:
        mov     cx,nrequest      ; CX = number of records to write.
        mov     ah,28h           ; Get MS-DOS to write CX records
        int     21h              ; from DTA to file.
        loadDP  ds,bx,pnactual   ; DS:BX = address of nactual.
        mov     ds:[bx],cx       ; Return number of records written.
        cbw                      ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 29H (41)

1.0 and later

Parse Filename

Function 29H examines a string for a valid filename in the form *drive:filename.ext.* If the string represents a valid filename, the function creates an unopened file control block (FCB) for it.

To Call

AH = 29H
AL = code to control parsing, as follows (bits 0–3 only):

Bit	Value	Meaning
0	0	Stop parsing if file separator is found.
	1	Ignore leading separators (parse off white space).
1	0	Set drive number field in FCB to 0 (current drive) if string does not include a drive identifier.
	1	Set drive as specified in the string; leave unaltered if string does not include a drive identifier.
2	0	Set filename field in the FCB to blanks (20H) if string does not include a filename.
	1	Leave filename field unaltered if string does not include a filename.
3	0	Set extension field in FCB to blanks (20H) if string does not include a filename extension.
	1	Leave extension field unaltered if string does not include a filename extension.

DS:SI = segment:offset of string to parse
ES:DI = segment:offset of buffer for unopened FCB

Returns

AL	= 00H	string does not contain wildcard characters
	01H	string contains wildcard characters
	FFH	drive specifier invalid
DS:SI	= segment:offset of first byte following the parsed string	
ES:DI	= segment:offset of unopened FCB	

Programmer's Notes

- Bits 0 through 3 of the byte in the AL register control the way the text string is parsed; bits 4 through 7 are not used and must be 0.
- After MS-DOS parses the string, DS:SI points to the first byte following the parsed string. If DS:SI points to an earlier byte, MS-DOS did not parse the entire string.
- If Function 29H encounters the MS-DOS wildcard character * (match all remaining characters) in a filename or extension, the remaining bytes in the corresponding FCB field are set to the wildcard character ? (match one character). For example, the string DOS*.D* would be converted to DOS????? in the filename field and D?? in the extension field of the FCB.
- With MS-DOS versions 1.x, the following characters are filename separators:

  : . ; , = + space tab / " [ ]

  With MS-DOS versions 2.0 and later, the following characters are filename separators:

  : . ; , = + space tab
- The following characters are filename terminators:

  / " [ ] < > ¦
  All filename separators
  Any control character
- If the string does not contain a valid filename, ES:DI+1 points to an ASCII blank character (20H).
- Function 29H cannot parse pathnames.

Related Functions

None

Example

```
;************************************************************;
;                                                            ;
;             Function 29H: Parse Filename into FCB          ;
;                                                            ;
;             int FCB_parse(uXFCB,name,ctrl)                 ;
;                 char *uXFCB;                               ;
;                 char *name;                                ;
;                 int ctrl;                                  ;
;                                                            ;
;             Returns -1 if error,                           ;
;                      0 if no wildcards found,              ;
;                      1 if wildcards found.                 ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   FCB_parse,PUBLIC,<ds,si,di>
parmDP  puXFCB
parmDP  pname
parmB   ctrl
cBegin
        loadDP  es,di,puXFCB    ; Pointer to unopened extended FCB.
        push    di              ; Save DI.
        xor     ax,ax           ; Fill all 22 (decimal) words of the
                                ; extended FCB with zeros.
        cld                     ; Make sure direction flag says UP.
        mov     cx,22d
        rep     stosw
        pop     di              ; Recover DI.
        mov     byte ptr [di],0ffh ; Set flag byte to mark this as an
                                   ; extended FCB.
        add     di,7            ; Advance pointer to start of regular
                                ; FCB.
        loadDP  ds,si,pname     ; Get pointer to filename into DS:SI.
        mov     al,ctrl         ; Get parse control byte.
        mov     ah,29h          ; Parse filename, please.
        int     21h
        cbw                     ; Set return parameter.
cEnd
```

Interrupt 21H (33) Function 2AH (42)

1.0 and later

Get Date

Function 2AH returns the current system date—year, month, day, and day of the week—in binary form.

To Call

AH = 2AH

Returns

AL = day of the week (0 = Sunday, 1 = Monday, 2 = Tuesday, and so on; MS-DOS versions 1.10 and later)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Programmer's Note

- Years outside the range 1980–2099 cannot be returned by Function 2AH.

Related Functions

2BH (Set Date)
2CH (Get Time)
2DH (Set Time)

Example

```
;************************************************************;
;                                                            ;
;           Function 2AH: Get Date                           ;
;                                                            ;
;           long get_date(pdow,pmonth,pday,pyear)            ;
;                char *pdow,*pmonth,*pday;                   ;
;                int *pyear;                                 ;
;                                                            ;
;           Returns the date packed into a long:             ;
;                low byte  = day of month                    ;
;                next byte = month                           ;
;                next word = year.                           ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   get_date,PUBLIC,ds
parmDP  pdow
parmDP  pmonth
parmDP  pday
parmDP  pyear
cBegin
        mov     ah,2ah          ; Set function code.
        int     21h             ; Get date info from MS-DOS.
        loadDP  ds,bx,pdow      ; DS:BX = pointer to dow.
        mov     [bx],al         ; Return dow.
        loadDP  ds,bx,pmonth    ; DS:BX = pointer to month.
        mov     [bx],dh         ; Return month.
        loadDP  ds,bx,pday      ; DS:BX = pointer to day.
        mov     [bx],dl         ; Return day.
        loadDP  ds,bx,pyear     ; DS:BX = pointer to year.
        mov     [bx],cx         ; Return year.
        mov     ax,dx           ; Pack day, month, ...
        mov     dx,cx           ; ... and year into return value.
cEnd
```

Interrupt 21H (33) Function 2BH (43)

1.0 and later

Set Date

Function 2BH accepts binary values for the year, month, and day of the month and stores them in the system's date counter as the number of days since January 1, 1980.

To Call

AH = 2BH
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Returns

AL =	00H	system date updated
	FFH	invalid date specified

Programmer's Note

- The year must be a 16-bit value in the range 1980 through 2099. Values outside this range are not accepted. In addition, supplying only the last two digits of the year causes an error.

Related Functions

2AH (Get Date)
2CH (Get Time)
2DH (Set Time)

Example

```
;*************************************************************;
;                                                             ;
;             Function 2BH: Set Date                          ;
;                                                             ;
;             int set_date(month,day,year)                    ;
;                 char month,day;                             ;
;                 int year;                                   ;
;                                                             ;
;             Returns 0 if date was OK, -1 if not.            ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   set_date,PUBLIC
parmB   month
parmB   day
parmW   year
cBegin
        mov     dh,month        ; Get new month.
        mov     dl,day          ; Get new day.
        mov     cx,year         ; Get new year.
        mov     ah,2bh          ; Set function code.
        int     21h             ; Ask MS-DOS to change date.
        cbw                     ; Return 0 or -1.
cEnd
```

Interrupt 21H (33) Function 2CH (44)

1.0 and later

Get Time

Function 2CH reports the current system time — hours (based on a 24-hour clock), minutes, seconds, and hundredths of a second — in binary form.

To Call

AH = 2CH

Returns

CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of second (0 through 99)

Programmer's Note

- The accuracy of the time returned by Function 2CH depends on the accuracy of the system's timekeeping hardware. On systems unable to resolve time to the hundredth of a second, the DL register may contain either 00H or an approximate value calculated by an MS-DOS algorithm.

Related Functions

2AH (Get Date)
2BH (Set Date)
2DH (Set Time)

Example

```
;*************************************************************;
;                                                             ;
;             Function 2CH: Get Time                          ;
;                                                             ;
;             long get_time(phour,pmin,psec,phund)            ;
;                  char *phour,*pmin,*psec,*phund;            ;
;                                                             ;
;             Returns the time packed into a long:            ;
;                  low byte  = hundredths                     ;
;                  next byte = seconds                        ;
;                  next byte = minutes                        ;
;                  next byte = hours.                         ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   get_time,PUBLIC,ds
parmDP  phour
parmDP  pmin
parmDP  psec
parmDP  phund
cBegin
        mov     ah,2ch          ; Set function code.
        int     21h             ; Get time from MS-DOS.
        loadDP  ds,bx,phour     ; DS:BX = pointer to hour.
        mov     [bx],ch         ; Return hour.
        loadDP  ds,bx,pmin      ; DS:BX = pointer to min.
        mov     [bx],cl         ; Return min.
        loadDP  ds,bx,psec      ; DS:BX = pointer to sec.
        mov     [bx],dh         ; Return sec.
        loadDP  ds,bx,phund     ; DS:BX = pointer to hund.
        mov     [bx],dl         ; Return hund.
        mov     ax,dx           ; Pack seconds, hundredths, ...
        mov     dx,cx           ; ... minutes, and hour into
                                ; return value.
cEnd
```

Interrupt 21H (33) Function 2DH (45)

1.0 and later

Set Time

Function 2DH accepts binary values for the hour (based on a 24-hour clock), minute, second, and hundredths of a second and stores them in the operating system's time counter.

To Call

AH = 2DH
CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of second (0 through 99)

Returns

AL =	00H	time successfully updated
	FFH	invalid time specified

Programmer's Note

- On systems that are unable to resolve the time to the hundredth of a second, the DL register should be set to 00H before Function 2DH is called.

Related Functions

2AH (Get Date)
2BH (Set Date)
2CH (Get Time)

Example

```
;************************************************************;
;                                                            ;
;             Function 2DH: Set Time                         ;
;                                                            ;
;             int set_time(hour,min,sec,hund)                ;
;                 char hour,min,sec,hund;                    ;
;                                                            ;
;             Returns 0 if time was OK, -1 if not.           ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   set_time,PUBLIC
parmB   hour
parmB   min
parmB   sec
parmB   hund
cBegin
        mov     ch,hour         ; Get new hour.
        mov     cl,min          ; Get new minutes.
        mov     dh,sec          ; Get new seconds.
        mov     dl,hund         ; Get new hundredths.
        mov     ah,2dh          ; Set function code.
        int     21h             ; Ask MS-DOS to change time.
        cbw                     ; Return 0 or -1.
cEnd
```

Interrupt 21H (33) Function 2EH (46)

1.0 and later

Set/Reset Verify Flag

Function 2EH turns the internal MS-DOS verify flag on or off, thus determining whether MS-DOS verifies disk write operations.

To Call

AH = 2EH
AL = 00H turn verify off
01H turn verify on
DL = 00H (MS-DOS versions 1.x and 2.x only)

Returns

Nothing

Programmer's Notes

- If the verify flag is on, MS-DOS requests any block-device driver to verify each sector written. If the driver does not support read-after-write verification, the verify flag has no effect.
- Function 54H (Get Verify Flag) can be used to check the current setting of the verify flag.
- Verifying data slows disk access during write operations. Because disk errors are rare, the default setting of the verify flag is off.
- Verification can be controlled at the user level with the MS-DOS VERIFY command.

Related Function

54H (Get Verify Flag)

Example

```
;************************************************************;
;                                                            ;
;             Function 2EH: Set/Reset Verify Flag            ;
;                                                            ;
;             int set_verify(newvflag)                       ;
;                 char newvflag;                             ;
;                                                            ;
;             Returns 0.                                     ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   set_verify,PUBLIC
parmB   newvflag
cBegin
        mov     al,newvflag       ; Get new value of verify flag.
        mov     ah,2eh            ; Set function code.
        int     21h               ; Ask MS-DOS to store flag.
        xor     ax,ax             ; Return 0.
cEnd
```

Interrupt 21H (33)
Function 2FH (47)

2.0 and later

Get DTA Address

Function 2FH returns the current disk transfer area (DTA) address.

To Call

AH	= 2FH

Returns

ES:BX	= segment:offset of current DTA address

Programmer's Notes

- Function 2FH returns the base address of the current DTA. MS-DOS has no way of knowing the size of the buffer at that address; the program must ensure that the buffer pointed to by the DTA address is large enough to hold any records transferred to it.
- The current DTA address can be set with Function 1AH (Set DTA Address). If the DTA address is not set, MS-DOS uses a default buffer of 128 bytes located at offset 80H in the program segment prefix (PSP).

Related Function

1AH (Set DTA Address)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 2FH: Get DTA Address                    ;
        ;                                                            ;
        ;           char far *get_DTA()                              ;
        ;                                                            ;
        ;           Returns a far pointer to the DTA buffer.         ;
        ;                                                            ;
        ;************************************************************;

cProc   get_DTA,PUBLIC
cBegin
        mov     ah,2fh          ; Set function code.
        int     21h             ; Ask MS-DOS for current DTA address.
        mov     ax,bx           ; Return offset in AX.
        mov     dx,es           ; Return segment in DX.
cEnd
```

Interrupt 21H (33) Function 30H (48)

2.0 and later

Get MS-DOS Version Number

Function 30H returns the major and minor version numbers for MS-DOS versions 2.0 and later.

To Call

AH	= 30H
AL	= 00H

Returns

AL	= major version number (for example, 3 for MS-DOS version 3.x)
AH	= minor version number (for example, 0AH for MS-DOS version x.10)
BH	= original equipment manufacturer's (OEM's) serial number (OEM dependent — usually 00H for PC-DOS, 0FFH or other values for MS-DOS)
BL:CX	= 24-bit user serial number (optional; OEM dependent)

Programmer's Notes

- With MS-DOS versions 1.x, Function 30H returns 00H in the AL register; the value returned in AH is variable and not representative of the actual 1.x minor version number.
- Function 30H supplies the MS-DOS version number to an application program that might require features of the operating system that are not available in all versions. If an application attempts to use such features with the wrong version of MS-DOS, the results are unpredictable.

  Applications requiring MS-DOS version 2.0 or later should use Function 30H to check for versions 1.x. Because versions 1.x do not contain predefined handles for displaying error messages, Function 02H (Character Output) or Function 09H (Display String) must be used with those versions. Similarly, applications running under versions 1.x cannot terminate through a call to Function 4CH (Terminate Process with Return Code).

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 30H: Get MS-DOS Version Number          ;
        ;                                                            ;
        ;           int DOS_version()                                ;
        ;                                                            ;
        ;           Returns number of MS-DOS version, with           ;
        ;              major version in high byte,                   ;
        ;              minor version in low byte.                    ;
        ;                                                            ;
        ;************************************************************;

cProc   DOS_version,PUBLIC
cBegin
        mov     ax,3000H        ; Set function code and clear AL.
        int     21h             ; Ask MS-DOS for version number.
        xchg    al,ah           ; Swap major and minor numbers.
cEnd
```

Interrupt 21H (33) Function 31H (49)

2.0 and later

Terminate and Stay Resident

Function 31H terminates a program and returns control to the parent process (usually COMMAND.COM) but keeps the terminated program resident in memory.

To Call

AH = 31H
AL = return code
DX = number of paragraphs of memory to be reserved for current process

Returns

Nothing

Programmer's Notes

- The following interrupt vectors are restored from the program segment prefix (PSP) of the terminated program:

PSP Offset	Vector for Interrupt
0AH	Interrupt 22H (terminate routine)
0EH	Interrupt 23H (Control-C handler)
12H	Interrupt 24H (critical error handler) (versions 2.0 and later.)

- The minimum amount of memory a process can reserve is 6 paragraphs (60H bytes), which constitutes the initial portion of the process's PSP (including the reserved areas).
- The amount of memory required by the program is not necessarily the same as the size of the file that holds the program on disk. The program must allow for its PSP and stack in the amount of memory reserved; on the other hand,the memory occupied by code and data used only during program initialization frequently can be discarded as a side effect of the Function 31H call.

  Before Function 31H is called, memory allocated to the terminating process's environment block should be released by loading ES with the segment value at offset 2CH in the PSP (the segment address of the environment) and calling Function 49H (Free Memory Block).
- The terminating process should return a completion code in the AL register. If the program terminates normally, the return code should be 00H. A return code of 01H or greater usually indicates that termination was caused by an error encountered by the process.

The parent process can retrieve the return code with Function 4DH (Get Return Code of Child Process). If control returns to COMMAND.COM, the return code can be tested with an ERRORLEVEL statement in a batch file.

- After terminating the current process, MS-DOS attempts to set the program's memory allocation to the amount specified in DX.
- Function 31H is most often used for memory-resident utilities and subroutine libraries that can be accessed using interrupts.
- This function is preferable to Interrupt 27H (Terminate and Stay Resident) because it allows programs that are larger than 64 KB to remain resident, allows the terminating program to pass a return code to the parent process, and does not require that the CS register contain the PSP address.

Related Functions

48H (Allocate Memory Block)
49H (Free Memory Block)
4AH (Resize Memory Block)
4BH (Load and Execute Program)
4CH (Terminate Process with Return Code)
4DH (Get Return Code of Child Process)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;          Function 31H: Terminate and Stay Resident          ;
        ;                                                             ;
        ;          void keep_process(exit_code,nparas)                ;
        ;               int exit_code,nparas;                         ;
        ;                                                             ;
        ;          Does NOT return!                                   ;
        ;                                                             ;
        ;*************************************************************;

cProc   keep_process,PUBLIC
parmB   exit_code
parmW   nparas
cBegin
        mov     al,exit_code    ; Get return code.
        mov     dx,nparas       ; Set DX to number of paragraphs the
                                ; program wants to keep.
        mov     ah,31h          ; Set function code.
        int     21h             ; Ask MS-DOS to keep process.
cEnd
```

Interrupt 21H (33) Function 33H (51)

2.0 and later

Get/Set Control-C Check Flag

Function 33H gets or sets the status of the Control-C check flag.

To Call

AH	= 33H	
AL	= 00H	get current Control-C check flag
	01H	set Control-C check flag to value in DL

If AL is 01H:

DL	= 00H	set Control-C check flag to off
	01H	set Control-C check flag to on

Returns

AL	= 00H	flag set successfully
	FFH	code in AL on call not 00H or 01H

If AL was 00H on call:

DL	= 00H	Control-C check flag off
	01H	Control-C check flag on

Programmer's Notes

- If the Control-C check flag is off, MS-DOS checks for a Control-C entered at the keyboard only during servicing of the character I/O functions, 01H through 0CH. If the Control-C check flag is on, MS-DOS also checks for user entry of a Control-C during servicing of other functions, such as file and record operations.
- The state of the Control-C check flag affects all programs. If a program needs to change the state of Control-C checking, it should save the original flag and restore it before terminating.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 33H: Get/Set Control-C Check Flag        ;
        ;                                                            ;
        ;          int controlC(func,state)                          ;
        ;              int func,state;                               ;
        ;                                                            ;
        ;          Returns current state of Control-C flag.          ;
        ;                                                            ;
        ;************************************************************;

cProc   controlC,PUBLIC
parmB   func
parmB   state
cBegin
        mov     al,func         ; Get set/reset function.
        mov     dl,state        ; Get new value if present.
        mov     ah,33h          ; MS-DOS ^C check function.
        int     21h             ; Call MS-DOS.
        mov     al,dl           ; Return current state.
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 34H (52)

2.0 and later

Return Address of InDOS Flag

Function 34H returns the address of the InDOS flag, which reflects the current state of Interrupt 21H function processing.

Note: Microsoft cannot guarantee that the information in this entry will be valid for future versions of MS-DOS.

To Call

AH = 34H

Returns

ES:BX = segment:offset of InDOS flag

Programmer's Notes

- The InDOS flag is a byte within the MS-DOS kernel. The value in InDOS is incremented when MS-DOS begins execution of an Interrupt 21H function and decremented when MS-DOS's processing of that function is completed. Thus, the value of InDOS is zero only when no Interrupt 21H processing is occurring.
- The InDOS flag is one of the elements used in terminate-and-stay-resident (TSR) programs to determine when the TSR can be executed safely.

Related Functions

None

Example

```
                ;************************************************************;
                ;                                                            ;
                ;       Function 34H: Get Return Address of InDOS Flag       ;
                ;                                                            ;
                ;       char far *inDOS_ptr()                                ;
                ;                                                            ;
                ;       Returns a far pointer to the MS-DOS inDOS flag.      ;
                ;                                                            ;
                ;************************************************************;

cProc   inDOS_ptr,PUBLIC
cBegin
        mov     ah,34h          ; InDOS flag function.
        int     21h             ; Call MS-DOS.
        mov     ax,bx           ; Return offset in AX.
        mov     dx,es           ; Return segment in DX.
cEnd
```

Interrupt 21H (33) Function 35H (53)

2.0 and later

Get Interrupt Vector

Function 35H returns the address stored in the interrupt vector table for the handler associated with the specified interrupt.

To Call

AH	= 35H
AL	= interrupt number

Returns

ES:BX	= segment:offset of handler for interrupt specified in AL

Programmer's Note

- Interrupt vectors should always be read with Function 35H and set with Function 25H (Set Interrupt Vector). Programs should never attempt to read or change interrupt vectors directly in memory.

Related Function

25H (Set Interrupt Vector)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 35H: Get Interrupt Vector               ;
        ;                                                            ;
        ;           typedef void (far *FCP)();                       ;
        ;           FCP get_vector(intnum)                           ;
        ;               int intnum;                                  ;
        ;                                                            ;
        ;           Returns a far code pointer that is the           ;
        ;           segment:offset of the interrupt vector.          ;
        ;                                                            ;
        ;************************************************************;

cProc   get_vector,PUBLIC
parmB   intnum
cBegin
        mov     al,intnum       ; Get interrupt number into AL.
        mov     ah,35h          ; Select "get vector" function.
        int     21h             ; Call MS-DOS.
        mov     ax,bx           ; Return vector offset.
        mov     dx,es           ; Return vector segment.
cEnd
```

Interrupt 21H (33) Function 36H (54)

2.0 and later

Get Disk Free Space

Function 36H returns disk-storage information for the specified drive.

To Call

AH = 36H
DL = drive specification (0 = default drive, 1 = drive A, 2 = drive B, and so on)

Returns

If function is successful:

AX = number of sectors per cluster
BX = number of clusters available
CX = number of bytes per sector
DX = number of clusters on drive

If function is not successful:

AX = FFFFH	invalid drive number in DL

Programmer's Notes

- The AX register should be checked for a value of FFFFH (error) before information returned by this function is used.
- The number of bytes of free storage remaining on the disk can be calculated by multiplying available clusters times sectors per cluster times bytes per sector (BX * AX * CX).
- Function 36H regards "lost" clusters (clusters that are allocated in the file allocation table [FAT] but do not belong to a file) as being in use and subtracts them from the amount of available storage, exactly as if they were allocated to a file.
- With MS-DOS versions 2.0 and later, Function 36H should be used in preference to the FCB Functions 1BH (Get Default Drive Data) and 1CH (Get Drive Data).

Related Functions

1BH (Get Default Drive Data)
1CH (Get Drive Data)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;             Function 36H: Get Disk Free Space              ;
        ;                                                            ;
        ;             long free_space(drive_ltr)                     ;
        ;                 char drive_ltr;                            ;
        ;                                                            ;
        ;             Returns the number of bytes free as            ;
        ;             a long integer.                                ;
        ;                                                            ;
        ;************************************************************;

cProc   free_space,PUBLIC
parmB   drive_ltr
cBegin
        mov     dl,drive_ltr    ; Get drive letter.
        or      dl,dl           ; Leave 0 alone.
        jz      fsp
        and     dl,not 20h      ; Convert letter to uppercase.
        sub     dl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
fsp:
        mov     ah,36h          ; Set function code.
        int     21h             ; Ask MS-DOS to get disk information.
        mul     cx              ; Bytes/sector * sectors/cluster
        mul     bx              ; * free clusters.
cEnd
```

Interrupt 21H (33)
Function 38H (56)

2.0 and later

Get/Set Current Country: Get Current Country

Function 38H includes two subfunctions that either get or set country data, depending on the value in the DX register when the function is called.

With MS-DOS versions 2.0 and later, if DX contains any value other than FFFFH, the Get Current Country subfunction is invoked. Information on date, currency, and other country-specific formats is then returned in a buffer specified by the calling program. The country code is usually the same as the country's international telephone prefix.

To Call

AH	= 38H	

With MS-DOS versions 2.x:

AL	= 00H	current country
DS:DX	= segment:offset of 32-byte buffer	

With MS-DOS versions 3.x:

AL	= 00H	current country
	01–FEH	country code between 1 and 254
	FFH	country code of 255 or greater, specified in BX
BX	= country code if AL = FFH	
DS:DX	= segment:offset of 34-byte buffer	

Returns

If function is successful:

Carry flag is clear.

BX	= country code (MS-DOS version 3.x only)
DS:DX	= segment:offset of buffer containing country information

If function is not successful:

Carry flag is set.

AX	= error code:	
	02H	invalid country code

Programmer's Notes

- With MS-DOS versions 2.x, the Get Current Country subfunction returns the following information for the current country in the 32-byte country-data buffer (ASCIIZ format is an ASCII character string ending in a zero byte):

Offset	Type	Description
00H	Word	Date format: 0 = United States (m/d/y) 1 = Europe (d/m/y) 2 = Japan (y/m/d)
02H	ASCIIZ	Currency symbol
04H	ASCIIZ	Character used as thousands separator
06H	ASCIIZ	Character used as decimal separator
08H	24 bytes	Reserved

- With MS-DOS versions 3.x, the Get Current Country subfunction returns the following information for the specified country in the 34-byte country-data buffer:

Offset	Type	Description
00H	Word	Date format: 0 = United States (m/d/y) 1 = Europe (d/m/y) 2 = Japan (y/m/d)
02H	ASCIIZ	Currency symbol (5 bytes, as opposed to 2 in versions 2.x of MS-DOS)
07H	ASCIIZ	Character used as thousands separator
09H	ASCIIZ	Character used as decimal separator
0BH	ASCIIZ	Character used as date separator
0DH	ASCIIZ	Character used as time separator
0FH	Byte	Position of currency symbol; possible values are 00H Currency symbol precedes value with no space 01H Currency symbol follows value with no space 02H Currency symbol precedes value with one space 03H Currency symbol follows value with one space
10H	Byte	Number of decimal places in currency

(more)

Offset	Type	Description
11H	Byte	Time format (00H = 12-hour clock; 01H = 24-hour clock)
12H	Dword	Case-mapping call address (*See* Programmer's Notes below.)
16H	ASCIIZ	Character used as separator in data lists
18H	10 bytes	Reserved

- The case-mapping call address (MS-DOS versions 3.x only) is the segment:offset of a FAR procedure that performs country-specific mapping on ASCII characters in the range 80H through 0FFH. The character to be mapped must be placed in the AL register before the call is made. If the character has an uppercase value, that value is returned in AL. If the character has no such value, AL is unchanged.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

38H (Set Current Country subfunction)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;         Function 38H: Get/Set Current Country Data          ;
        ;                                                             ;
        ;         int country_info(country,pbuffer)                   ;
        ;             char country,*pbuffer;                          ;
        ;                                                             ;
        ;         Returns -1 if the "country" code is invalid.        ;
        ;                                                             ;
        ;*************************************************************;

cProc   country_info,PUBLIC,ds
parmB   country
parmDP  pbuffer
cBegin
        mov     al,country      ; Get country code.
        loadDP  ds,dx,pbuffer   ; Get buffer pointer (or -1).
        mov     ah,38h          ; Set function code.
        int     21h             ; Ask MS-DOS to get country
                                ; information.
        jnb     cc_ok           ; Branch if country code OK.
        mov     ax,-1           ; Else return -1.
cc_ok:
cEnd
```

Interrupt 21H (33) Function 38H (56)

3.0 and later

Get/Set Current Country: Set Current Country

Function 38H includes two subfunctions that either get or set country data, depending on the value in the DX register when the function is called.

With MS-DOS versions 3.0 and later, the Set Current Country subfunction is invoked if Function 38H is called with DX = FFFFH (–1). This subfunction selects the country for which subsequent calls to Get Current Country will return information. The country code used with this function is usually the same as the country's international telephone prefix.

To Call

AH	= 38H	
AL	= country code	for a code less than 255
	FFH	for country code of 255 or greater, specified in BX
BX	= country code if AL = FFH	
DX	= FFFFH (–1)	

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	02H	invalid country code

Programmer's Notes

- MS-DOS normally uses the country code associated with the current KEYBxx keyboard driver file, if any. Otherwise, the default country code is OEM dependent.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

38H (Get Current Country subfunction)

Example

See Function 38H Subfunction Get Current Country for example.

Interrupt 21H (33) Function 39H (57)

2.0 and later

Create Directory

Function 39H creates a subdirectory using the specified path.

To Call

AH	= 39H
DS:DX	= segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	03H	path not found
	05H	access denied

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- MS-DOS places the current directory (.) and parent directory (..) entries in all new directories.
- Function 39H returns error code 05H (access denied) in the following cases:
  - File or directory with the same name already exists in the specified path.
  - Parent directory is the root directory and the root directory is full.
  - Path specifies a device.
  - Program is running on a network under MS-DOS version 3.1 or later and the user does not have Create access to the parent directory.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

3AH (Remove Directory)
3BH (Change Current Directory)
47H (Get Current Directory)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;              Function 39H: Create Directory                 ;
        ;                                                             ;
        ;              int make_dir(pdirpath)                         ;
        ;                  char *pdirpath;                            ;
        ;                                                             ;
        ;              Returns 0 if directory created OK,             ;
        ;              otherwise returns error code.                  ;
        ;                                                             ;
        ;*************************************************************;

cProc   make_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,39h          ; Set function code.
        int     21h             ; Ask MS-DOS to make new subdirectory.
        jb      md_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
md_err:
cEnd
```

Interrupt 21H (33) Function 3AH (58)

2.0 and later

Remove Directory

Function 3AH removes (deletes) the specified subdirectory.

To Call

AH	= 3AH
DS:DX	= segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

03H	path not found
05H	access denied
10H	current directory was specified

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- Function 3AH returns error code 05H (access denied) in the following cases:
  - Directory is not empty.
  - Root directory was specified.
  - Current directory was specified.
  - Path does not specify a valid directory.
  - Directory is malformed (. and .. not first two entries).
  - User has insufficient access rights on a network running under MS-DOS version 3.1 or later.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

39H (Create Directory)
3BH (Change Current Directory)
47H (Get Current Directory)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;             Function 3AH: Remove Directory                  ;
        ;                                                             ;
        ;             int remove_dir(pdirpath)                        ;
        ;                 char *pdirpath;                             ;
        ;                                                             ;
        ;             Returns 0 if directory was removed,             ;
        ;             otherwise returns error code.                   ;
        ;                                                             ;
        ;*************************************************************;

cProc   remove_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,3ah          ; Set function code.
        int     21h             ; Ask MS-DOS to delete subdirectory.
        jb      rd_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
rd_err:
cEnd
```

Interrupt 21H (33) Function 3BH (59)

2.0 and later

Change Current Directory

Function 3BH changes the current directory to the specified path.

To Call

AH	= 3BH
DS:DX	= segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	03H	path not found

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- Before a call to Function 3BH, Function 47H (Get Current Directory) can be used to determine the current directory so that the original directory can be restored later (for example, on termination of the program).
- Function 3BH can be used with programs that rely on either FCB-based or handle-based calls. It is the only method of changing the current directory that is supported by MS-DOS.
- The path string is limited to a total of 64 characters, including separators.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

39H (Create Directory)
3AH (Remove Directory)
47H (Get Current Directory)

Example

```
        ;**************************************************************;
        ;                                                              ;
        ;            Function 3BH: Change Current Directory           ;
        ;                                                              ;
        ;            int change_dir(pdirpath)                          ;
        ;                char *pdirpath;                               ;
        ;                                                              ;
        ;            Returns 0 if directory was changed,               ;
        ;            otherwise returns error code.                     ;
        ;                                                              ;
        ;**************************************************************;

cProc   change_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,3bh          ; Ask MS-DOS to move to
        int     21h             ; different directory.
        jb      cd_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
cd_err:
cEnd
```

Interrupt 21H (33) Function 3CH (60)

2.0 and later

Create File with Handle

Function 3CH creates a file, assigns it the attributes specified, and returns a 16-bit handle for the file. If the named file already exists, Function 3CH opens it and truncates it to zero length.

To Call

AH	= 3CH
CX	= attribute
DS:DX	= segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

AX	= handle number

If function is not successful:

Carry flag is set.

AX = error code:

03H	path not found
04H	too many open files
05H	access denied

Programmer's Notes

- Function 3CH is preferable to Function 16H (Create File with FCB) for creating a file because it supports full pathnames. Function 16H should be used only if compatibility with versions 1.x of MS-DOS is required.
- The pathname must be a null-terminated ASCII string (ASCIIZ).
- Bits 0 through 2 of the 2-byte file attribute in CX determine whether the file is normal, read-only, hidden, or system. The attribute codes are
  - 00H normal file
  - 01H read-only file
  - 02H hidden file
  - 04H system file

  Bits 3 through 5 are associated with volume labels, subdirectories, and archive files. The volume and subdirectory bits are invalid for Function 3CH and must be set to 0. Bits 6 through 15 should be set to 0 to ensure future compatibility.

Values can be combined to set several file attributes. For example, if Function 3CH is called with CX = 0003H, the file created is a read-only hidden file.

- Because Function 3CH truncates an existing file to zero length, any information previously in the file is lost. Alternative functions that protect against such loss include the following:
  - Function 3DH (Open File with Handle) or Function 4EH (Find First File), which can be used to check for the previous existence of the file before Function 3CH is called
  - Function 5AH (Create Temporary File), which creates a file in the specified subdirectory and gives it a unique name assigned by MS-DOS
  - Function 5BH (Create New File), which is similar to Function 3CH but fails if it finds a file that matches the specified pathname
- After creating a file, Function 3CH sets the position of the file pointer to 0. Thus, the next read or write operation takes place at the beginning of the file.
- Function 3CH returns error code 04H (too many open files) if no handle is currently available. With MS-DOS versions 3.2 and earlier, a single process can have no more than 20 files open at one time, 5 of which are normally assigned to the standard devices.

  Error code 05H (access denied) is returned if the file is to be created in the root directory and the root is full or if a read-only file with the same name already exists in the specified subdirectory.
- On networks running under MS-DOS version 3.1 or later, the user must have Create access to the directory containing the file specified.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

16H (Create File with FCB)
43H (Get/Set File Attributes)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
;************************************************************;
;                                                            ;
;           Function 3CH: Create File with Handle            ;
;                                                            ;
;           int create(pfilepath,attr)                       ;
;              char *pfilepath;                              ;
;           int attr;                                        ;
;                                                            ;
;           Returns -1 if file was not created,              ;
;           otherwise returns file handle.                   ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   create,PUBLIC,ds
parmDP  pfilepath
parmW   attr
cBegin
        loadDP  ds,dx,pfilepath ; Get pointer to pathname.
        mov     cx,attr         ; Get new file's attribute.
        mov     ah,3ch          ; Ask MS-DOS to make a new file.
        int     21h
        jnb     cr_ok           ; Branch if MS-DOS returned handle.
        mov     ax,-1           ; Else return -1.
cr_ok:
cEnd
```

Interrupt 21H (33) Function 3DH (61)

2.0 and later

Open File with Handle

Function 3DH opens the specified file and returns a 16-bit handle number for subsequent access to the file.

To Call

AH = 3DH

With versions 2.x of MS-DOS:

AL = file-access code:

Bits	Value	Meaning
3–7	00000	Reserved
0–2	000	Read-only access
	001	Write-only access
	010	Read/write access

DS:DX = segment:offset of ASCIIZ pathname

With versions 3.x of MS-DOS:

AL = file-access, file-sharing, and inheritance codes:

Bits	Value	Meaning
7 (inherit bit)	0	Child process inherits file
	1	Child process does not inherit file
4–6 (sharing mode; file access granted to other processes)	000	Compatibility mode
	001	Deny read/write access
	010	Deny write access
	011	Deny read access
	100	Deny none
3	0	Reserved
0–2 (access code; file usage)	000	Read-only access
	001	Write-only access
	010	Read/write access

DS:DX = segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

AX = handle number

If function is not successful:

Carry flag is set.

AX = error code:

02H	file not found
03H	path not found
04H	too many open files
05H	access denied
0CH	invalid access code

Programmer's Notes

- Function 3DH is preferable to Function 0FH (Open File with FCB) because it allows the use of pathnames. Function 0FH should be used only if compatibility with versions 1.x of MS-DOS is required.
- Function 3DH opens any file matching the pathname in DS:DX, including hidden and system files.
- The pathname must be a null-terminated ASCII string (ASCIIZ).
- Function 3DH returns error code 04H (too many open files) if no handle is currently available. With MS-DOS versions 3.2 and earlier, a single process can have no more than 20 files open at one time, 5 of which are normally assigned to the standard devices.

  Function 3DH returns error code 05H (access denied) if the pathname specifies a directory or volume label or if read/write access was requested for a read-only file.

  Function 3DH returns error code 0CH (invalid access code) if bits 0–2 in AL contain any value other than 000, 001, or 010.
- With MS-DOS versions 2.x, only bits 0–2 of the byte in AL are meaningful; they should contain the type of access allowed for the file. Bits 3–7 should always be zero.

  With MS-DOS versions 3.0 and later, networking capabilities require bits 4–7, as well as 0–2, to be set. (Bit 3 is reserved and should be 0.)

  Bit 7, the inherit bit, should be set to indicate whether child processes created by the current process with Function 4BH (Load and Execute Program) either can (0) or cannot (1) inherit the file. When a process inherits a file, it also inherits the access and sharing modes.

Bits 4–6 are called the "sharing code"; they indicate the type of access other users on the network can have to the file. The five sharing modes and the conditions under which they pertain are as follows:

- mode 000 (compatibility). Allows other programs running on the same machine unlimited access to the file. Programs running on other machines cannot access the file across the network unless it has the read-only attribute. An attempt to open the file in compatibility mode fails if the file has already been opened with any other sharing mode.
- 001 (deny read and write access). Provides exclusive access to the file. Any subsequent attempts by others (including the current process) to open the file fail. This mode fails if the file has already been opened in compatibility mode or for read or write access, even by the current process.
- 010 (deny write access). Allows other processes to open the file for read-only access. This mode fails if the file has already been opened in compatibility mode or for write access by any other process.
- 011 (deny read access). Allows other processes to open the file for write-only access. This mode fails if the file has already been opened in compatibility mode or for read access by any other process.
- 100 (deny none). Similar to compatibility mode, but does not allow other processes to open the file in compatibility mode. This mode fails if the file has already been opened in compatibility mode by any other process.

- When the file is opened, the position of the file pointer is set to 0. Function 42H (Move File Pointer) can be used to change its position.
- With MS-DOS versions 3.0 and later, if this function fails because of a file-sharing error, the operating system issues an Interrupt 24H (Critical Error Handler Address) with error code 02H (drive not ready). Function 59H (Get Extended Error Information) must be used to find the extended error code specifying the type of sharing violation that occurred.

Related Functions

0FH (Open File with FCB)
3EH (Close File)
3FH (Read File or Device)
40H (Write File or Device)
42H (Move File Pointer)
43H (Get/Set File Attributes)
57H (Get/Set Date/Time of File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;             Function 3DH: Open File with Handle            ;
        ;                                                            ;
        ;             int open(pfilepath,mode)                       ;
        ;                 char *pfilepath; int mode;                 ;
        ;                                                            ;
        ;             Modes:                                         ;
        ;                     0: Read                                ;
        ;                     1: Write                               ;
        ;                     2: Read/Write                          ;
        ;                                                            ;
        ;             Returns -1 if file was not opened,             ;
        ;             otherwise returns file handle.                 ;
        ;                                                            ;
        ;************************************************************;

cProc   open,PUBLIC,ds
parmDP  pfilepath
parmB   mode
cBegin
        loadDP  ds,dx,pfilepath ; Get pointer to pathname.
        mov     al,mode         ; Get read/write mode.
        mov     ah,3dh          ; Request MS-DOS to open the
        int     21h             ; existing file.
        jnb     op_ok           ; Branch if MS-DOS returned handle.
        mov     ax,-1           ; Else return -1.
op_ok:
cEnd
```

Interrupt 21H (33) Function 3EH (62)

2.0 and later

Close File

Function 3EH closes the file referenced by the specified handle.

To Call

AH = 3EH
BX = handle number

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

06H	invalid handle number

Programmer's Notes

- The handle in BX must be one that was returned by a successful call to one of the following functions:
  - 3CH (Create File with Handle)
  - 3DH (Open File with Handle)
  - 5AH (Create Temporary File)
  - 5BH (Create New File)
- If the file has been modified, truncated, or extended, Function 3EH updates the current date, time, and file size in the directory entry.
- All internal MS-DOS buffers for the file, including directory and file allocation table (FAT) buffers, are flushed to disk.
- With MS-DOS versions 3.0 and later, a program must remove all file locks in effect before it closes a file. The result of closing a file with active locks is unpredictable.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

10H (Close File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;               Function 3EH: Close File                     ;
        ;                                                            ;
        ;               int close(handle)                            ;
        ;                   int handle;                              ;
        ;                                                            ;
        ;               Returns -1 if file was not closed,           ;
        ;               otherwise returns 0.                         ;
        ;                                                            ;
        ;************************************************************;

cProc   close,PUBLIC
parmW   handle
cBegin
        mov     bx,handle       ; Get handle.
        mov     ah,3eh          ; Set function codes.
        int     21h             ; Ask MS-DOS to close handle.
        mov     al,0
        jnb     cl_ok           ; Branch if no error.
        mov     al,-1           ; Else return -1.
cl_ok:
        cbw                     ; Extend result.
cEnd
```

Interrupt 21H (33) Function 3FH (63)

2.0 and later

Read File or Device

Function 3FH reads from the file or device referenced by a handle.

To Call

AH	= 3FH
BX	= handle number
CX	= number of bytes to read
DS:DX	= segment:offset of data buffer

Returns

If function is successful:

Carry flag is clear.

AX	= number of bytes read from file
DS:DX	= segment:offset of data read from file

If function is not successful:

Carry flag is set.

AX = error code:

05H	access denied
06H	invalid handle

Programmer's Notes

- Data is read from the file beginning at the current location of the file pointer. After a successful read, the file pointer is updated to point to the byte following the last byte read.
- If Function 3FH returns 00H in the AX register, the function attempted to read when the file pointer was at the end of the file. If AX is less than CX, a partial record at the end of the file was read.
- Function 3FH can be used with all handles, including standard input (normally the keyboard). When reading from standard input, this function normally reads characters only to the first carriage-return character. Thus, the number of bytes read in AX will not necessarily match the length requested in CX.
- On networks running under MS-DOS version 3.1 or later, the user must have Read access to the directory and file containing the information to be read.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

40H (Write File or Device)
42H (Move File Pointer)
59H (Get Extended Error Information)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;           Function 3FH: Read File or Device                 ;
        ;                                                             ;
        ;           int read(handle,pbuffer,nbytes)                   ;
        ;               int handle,nbytes;                            ;
        ;               char *pbuffer;                                ;
        ;                                                             ;
        ;           Returns -1 if there was a read error,             ;
        ;           otherwise returns number of bytes read.           ;
        ;                                                             ;
        ;*************************************************************;

cProc   read,PUBLIC,ds
parmW   handle
parmDP  pbuffer
parmW   nbytes
cBegin
        mov     bx,handle       ; Get handle.
        loadDP  ds,dx,pbuffer   ; Get pointer to buffer.
        mov     cx,nbytes       ; Get number of bytes to read.
        mov     ah,3fh          ; Set function code.
        int     21h             ; Ask MS-DOS to read CX bytes.
        jnb     rd_ok           ; Branch if read worked.
        mov     ax,-1           ; Else return -1.
rd_ok:
cEnd
```

Interrupt 21H (33) Function 40H (64)

2.0 and later

Write File or Device

Function 40H writes the specified number of bytes to a file or device referenced by a handle.

To Call

AH	= 40H
BX	= handle
CX	= number of bytes to write
DS:DX	= segment:offset of data buffer

Returns

If function is successful:

Carry flag is clear.

AX	= number of bytes written to file or device

If function is not successful:

Carry flag is set.

AX = error code:

05H	access denied
06H	invalid handle

Programmer's Notes

- Data is written to the file or device beginning at the current location of the file pointer. After writing the specified data, Function 40H updates the position of the file pointer and returns the actual number of bytes written in AX.
- Function 40H returns error code 05H (access denied) if the file was opened as read-only with Function 3CH (Create File with Handle), 3DH (Open File with Handle), 5AH (Create Temporary File), or 5BH (Create New File). On networks running under MS-DOS version 3.1 or later, access is also denied if the file or record has been locked by another process.
- The handle number in BX must be one of the predefined device handles (0 through 4) or a handle obtained through a previous call to open or create a file (such as Function 3CH, 3DH, 5AH, or 5BH).
- If CX = 0, the file is truncated or extended to the current file pointer location. Clusters are allocated or released in the file allocation table (FAT) as required to fulfill the request.

- If the handle parameter for Function 40H refers to a disk file and the number of bytes written (returned in AX) is less than the number requested in CX, the destination disk is full. The carry flag is *not* set in this situation.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

3FH (Read File or Device)
42H (Move File Pointer)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;          Function 40H: Write File or Device                 ;
        ;                                                             ;
        ;          int write(handle,pbuffer,nbytes)                   ;
        ;              int handle,nbytes;                             ;
        ;              char *pbuffer;                                 ;
        ;                                                             ;
        ;          Returns -1 if there was a write error,             ;
        ;          otherwise returns number of bytes written.         ;
        ;                                                             ;
        ;*************************************************************;

cProc   write,PUBLIC,ds
parmW   handle
parmDP  pbuffer
parmW   nbytes
cBegin
        mov     bx,handle       ; Get handle.
        loadDP  ds,dx,pbuffer   ; Get pointer to buffer.
        mov     cx,nbytes       ; Get number of bytes to write.
        mov     ah,40h          ; Set function code.
        int     21h             ; Ask MS-DOS to write CX bytes.
        jnb     wr_ok           ; Branch if write successful.
        mov     ax,-1           ; Else return -1.
wr_ok:
cEnd
```

Interrupt 21H (33)
Function 41H (65)

2.0 and later

Delete File

Function 41H deletes the directory entry of the specified file.

To Call

AH	= 41H
DS:DX	= segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

02H	file not found
03H	path not found
05H	access denied

Programmer's Notes

- The pathname must be a null-terminated ASCII string (ASCIIZ). Unlike Function 13H (Delete File), Function 41H does not allow wildcard characters in the pathname.
- Because Function 41H supports the use of full pathnames, it is preferable to Function 13H.
- Function 41H returns error code 05H (access denied) and fails if the file has either a directory or volume attribute or if it is a read-only file.

  A directory can be deleted (if it is empty) with Function 3AH (Remove Directory). A read-only file can be deleted if its attribute is changed to normal with Function 43H (Get/Set File Attributes) before Function 41H is called.
- On networks running under MS-DOS version 3.1 or later, the user must have Create access to the directory containing the file to be deleted.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

3AH (Remove Directory)
43H (Get/Set File Attributes)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;                Function 41H: Delete File                   ;
        ;                                                            ;
        ;                int delete(pfilepath)                       ;
        ;                    char *pfilepath;                        ;
        ;                                                            ;
        ;                Returns 0 if file deleted,                  ;
        ;                otherwise returns error code.               ;
        ;                                                            ;
        ;************************************************************;

cProc   delete,PUBLIC,ds
parmDP  pfilepath
cBegin
        loadDP  ds,dx,pfilepath ; Get pointer to pathname.
        mov     ah,41h          ; Set function code.
        int     21h             ; Ask MS-DOS to delete file.
        jb      dl_err          ; Branch if MS-DOS could not delete
                                ; file.
        xor     ax,ax           ; Else return 0.
dl_err:
cEnd
```

Interrupt 21H (33) Function 42H (66)

2.0 and later

Move File Pointer

Function 42H sets the position of the file pointer (for the next read/write operation) for the file associated with the specified handle.

To Call

AH	= 42H	
AL	= method code:	
	00H	byte offset from beginning of file
	01H	byte offset from current location of file pointer
	02H	byte offset from end of file
BX	= handle number	
CX:DX	= offset value to move pointer:	
	CX	most significant half of a doubleword value
	DX	least significant half of a doubleword value

Returns

If function is successful:

Carry flag is clear.

DX:AX = new file pointer position (absolute byte offset from beginning of file)

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function (AL not 00H, 01H, or 02H)
	06H	invalid handle

Programmer's Notes

- The value in CX:DX is an offset specifying how far the file pointer is to be moved. With method code 00H, the value in CX:DX is always interpreted as a positive 32-bit integer, meaning the file pointer is always set relative to the beginning of the file.

  With method codes 01H and 02H, the value in CX:DX can be either a positive or negative 32-bit integer. Thus, method 1 can move the file pointer either forward or backward from its current position; method 2 can move the file pointer either forward or backward from the end of the file.

- Specifying method code 00H with an offset of 0 positions the file pointer at the beginning of the file. Similarly, specifying method code 02H with an offset of 0 conveniently positions the file pointer at the end of the file. With method code 02H offset 0, the size of the file can also be determined by examining the pointer position returned by the function.
- Depending on the offset specified in CX:DX, methods 1 and 2 may move the file pointer to a position before the start of the file. Function 42H does not return an error code if this happens, but later attempts to read from or write to the file will produce unexpected errors.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

3FH (Read File or Device)
40H (Write File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;           Function 42H: Move File Pointer                   ;
        ;                                                             ;
        ;           long seek(handle,distance,mode)                   ;
        ;                int handle,mode;                             ;
        ;                long distance;                               ;
        ;                                                             ;
        ;           Modes:                                            ;
        ;                   0: from beginning of file                 ;
        ;                   1: from the current position              ;
        ;                   2: from the end of the file               ;
        ;                                                             ;
        ;           Returns -1 if there was a seek error,             ;
        ;           otherwise returns long pointer position.          ;
        ;                                                             ;
        ;*************************************************************;

cProc   seek,PUBLIC
parmW   handle
parmD   distance
parmB   mode
cBegin
        mov     bx,handle       ; Get handle.
        les     dx,distance     ; Get distance into ES:DX.
        mov     cx,es           ; Put high word of distance into CX.
        mov     al,mode         ; Get move method code.
        mov     ah,42h          ; Set function code.
```

(more)

```
        int     21h             ; Ask MS-DOS to move file pointer.
        jnb     sk_ok           ; Branch if seek successful.
        mov     ax,-1           ; Else return -1.
        cwd
sk_ok:
cEnd
```

Interrupt 21H (33) Function 43H (67)

2.0 and later

Get/Set File Attributes

Function 43H gets or sets the attributes of the specified file.

To Call

AH = 43H

To get file attributes:

AL = 00H
DS:DX = segment:offset of ASCIIZ pathname

To set file attributes:

AL = 01H
CX = attributes to set:

Bit	Attribute
0	Read-only file
1	Hidden file
2	System file
5	Archive

DS:DX = segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

CX = attribute

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function (AL not 00H or 01H)
02H	file not found
03H	path not found
05H	access denied

Programmer's Notes

- The pathname must be a null-terminated ASCII string (ASCIIZ).
- Function 43H cannot be used to set or change either a volume-label or directory attribute (bits 3 and 4 of the attribute byte). With MS-DOS versions 3.x, Function 43H can be used to make a directory hidden or read-only.
- On networks running under MS-DOS version 3.1 or later, the user must have Create access to the directory containing the file in order to change the read-only, hidden, or system attribute. The archive bit, however, can be changed regardless of access rights.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;            Function 43H: Get/Set File Attributes           ;
        ;                                                            ;
        ;            int file_attr(pfilepath,func,attr)              ;
        ;                char *pfilepath;                            ;
        ;                int func,attr;                              ;
        ;                                                            ;
        ;            Returns -1 for all errors,                      ;
        ;            otherwise returns file attribute.               ;
        ;                                                            ;
        ;************************************************************;

cProc   file_attr,PUBLIC,ds
parmDP  pfilepath
parmB   func
parmW   attr
cBegin
        loadDP  ds,dx,pfilepath ; Get pointer to pathname.
        mov     al,func         ; Get/set flag into AL.
        mov     cx,attr         ; Get new attr (if present).
        mov     ah,43h          ; Set code function.
        int     21h             ; Call MS-DOS.
        jnb     fa_ok           ; Branch if no error.
        mov     cx,-1           ; Else return -1.
fa_ok:
        mov     ax,cx           ; Return this value.

cEnd
```

Interrupt 21H (33) Function 44H (68)

2.0 and later

IOCTL

Function 44H is a collection of subfunctions that provide a process a direct path of communication with a device driver. As such, this function is the most flexible means of gaining access to the full capabilities of an installed device.

An IOCTL subfunction is called with 44H in AH and the value for the subfunction in AL. If a subfunction has minor functions, those values are specified in CL. Otherwise, the BX, CX, and DX registers are used for such information as handles, drive identifiers, buffer addresses, and so on.

The subfunctions and the versions of MS-DOS with which they are available are

Subfunction	Name	MS-DOS Versions
00H	Get Device Data	2.0 and later
01H	Set Device Data	2.0 and later
02H	Receive Control Data from Character Device	2.0 and later
03H	Send Control Data to Character Device	2.0 and later
04H	Receive Control Data from Block Device	2.0 and later
05H	Send Control Data to Block Device	2.0 and later
06H	Check Input Status	2.0 and later
07H	Check Output Status	2.0 and later
08H	Check If Block Device Is Removable	3.0 and later
09H	Check If Block Device Is Remote	3.1 and later
0AH	Check If Handle Is Remote	3.1 and later
0BH	Change Sharing Retry Count	3.1 and later
0CH	Generic I/O Control for Handles Minor Code 45H: Set Iteration Count Minor Code 65H: Get Iteration Count	3.2
0DH	Generic I/O Control for Block Devices Minor Code 40H: Set Device Parameters Minor Code 60H: Get Device Parameters Minor Code 41H: Write Track on Logical Drive Minor Code 61H: Read Track on Logical Drive Minor Code 42H: Format and Verify Track on Logical Drive Minor Code 62H: Verify Track on Logical Drive	3.2

(more)

Subfunction	Name	MS-DOS Versions
0EH	Get Logical Drive Map	3.2
0FH	Set Logical Drive Map	3.2

These subfunctions are documented, either individually or in related pairs, in the entries that follow.

Interrupt 21H (33) Function 44H (68) Subfunction 00H

2.0 and later

IOCTL: Get Device Data

Function 44H Subfunction 00H gets information about a character device or file referenced by a handle.

To Call

AH = 44H
AL = 00H
BX = handle number

Returns

If function is successful:

Carry flag is clear.

DX contains information on file or device:

Bit	Value	Meaning
For a file (bit 7 = 0):		
8–15	0	Reserved.
7	0	Handle refers to a file.
6	0	File has been written.
0–5		Drive number (0 = A, 1 = B, 2 = C, and so on).
For a device (bit 7 = 1):		
15	0	Reserved.
14	1	Processes control strings transferred by IOCTL Subfunctions 02H (Receive Control Data from Character Device) and 03H (Send Control Data to Character Device), set by MS-DOS.
8–13	0	Reserved.
7	1	Handle refers to a device.
6	0	End of file on input.
5	0	Checks for control characters (cooked mode).
	1	Does not check for control characters (raw mode).

(more)

Bit	Value	Meaning
4	0	Reserved.
3	1	Clock device.
2	1	Null device.
1	1	Standard output device.
0	1	Standard input device.

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid IOCTL subfunction
05H	access denied
06H	invalid handle

Programmer's Notes

- Bits 8–15 of DX correspond to the upper 8 bits of the device-driver attribute word.
- The handle in BX must reference an open device or file.
- Bit 5 of the device data word for character-device handles defines whether that handle is in raw mode or cooked mode. In cooked mode, MS-DOS checks for Control-C, Control-P, Control-S, and Control-Z characters and transfers control to the Control-C exception handler (whose address is saved in the vector for Interrupt 23H) when a Control-C is detected. In raw mode, MS-DOS does not check for such characters when I/O is performed to the handle; however, it will still check for a Control-C entered at the keyboard on other function calls unless such checking has been turned off with Function 33H, the BREAK=OFF directive in CONFIG.SYS, or a BREAK OFF command at the MS-DOS prompt.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

33H (Get/Set Control-C Check Flag)
3CH (Create File with Handle)
3DH (Open File with Handle)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;        Function 44H, Subfunctions 00H,01H:                  ;
        ;                      Get/Set IOCTL Device Data              ;
        ;                                                             ;
        ;        int ioctl_char_flags(setflag,handle,newflags)        ;
        ;            int setflag;                                     ;
        ;            int handle;                                      ;
        ;            int newflags;                                    ;
        ;                                                             ;
        ;        Set setflag = 0 to get flags, 1 to set flags.        ;
        ;                                                             ;
        ;        Returns -1 for error, else returns flags.            ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_char_flags,PUBLIC
parmB   setflag
parmW   handle
parmW   newflags
cBegin
        mov     al,setflag      ; Get setflag.
        and     al,1            ; Save only lsb.
        mov     bx,handle       ; Get handle to character device.
        mov     dx,newflags     ; Get new flags (they are used only
                                ; by "set" option).
        mov     ah,44h          ; Set function code.
        int     21h             ; Call MS-DOS.
        mov     ax,dx           ; Assume success - prepare to return
                                ; flags.
        jnc     iocfx           ; Branch if no error.
        mov     ax,-1           ; Else return error flag.
iocfx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 01H

2.0 and later

IOCTL: Set Device Data

Function 44H Subfunction 01H, the complement of IOCTL Subfunction 00H, sets information about a character device — but not a file — referenced by a handle.

To Call

AH = 44H
AL = 01H
BX = handle number
DX = device data word:

Bit	Value	Meaning
8–15	0	Reserved.
7	1	Handle refers to a device.
6	0	End of file on input.
5	0	Check for control characters (cooked mode).
	1	Do not check for control characters (raw mode).
4	0	Reserved.
3	1	Clock device.
2	1	Null device.
1	1	Standard output device.
0	1	Standard input device.

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid IOCTL subfunction
05H	access denied
06H	invalid handle

Programmer's Notes

- The handle in BX must reference an open device.
- DH must be 00H. If it is not, the carry flag is set and error code 01H (invalid function) is returned.
- Bit 5 of the device data word for character-device handles selects raw mode or cooked mode for the handle. In cooked mode, MS-DOS checks for Control-C, Control-P, Control-S, and Control-Z characters and transfers control to the Control-C exception handler (whose address is saved in the vector for Interrupt 23H) when a Control-C is detected. In raw mode, MS-DOS does not check for such characters when I/O is performed to the handle; however, it will still check for a Control-C entered at the keyboard on other function calls unless such checking has been turned off with Function 33H, the BREAK=OFF directive in CONFIG.SYS, or a BREAK OFF command at the MS-DOS prompt.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

33H (Get/Set Control-C Check Flag)
3CH (Create File with Handle)
3DH (Open File with Handle)

Example

See SYSTEM CALLS: INTERRUPT 21H: Function 44H Subfunction 00H.

Interrupt 21H (33) Function 44H (68) Subfunctions 02H and 03H

2.0 and later

IOCTL: Receive Control Data from Character Device; Send Control Data to Character Device

Function 44H Subfunctions 02H and 03H respectively receive and send control strings from and to a character-oriented device driver.

To Call

AH	= 44H	
AL	= 02H	receive control strings
	03H	send control strings
BX	= handle number	
CX	= number of bytes to transfer	
DS:DX	= segment:offset of data buffer	

Returns

If function is successful:

Carry flag is clear.

AX = number of bytes transferred

If AL was 02H on call:

Buffer at DS:DX contains data read from device driver.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	05H	access denied
	06H	invalid handle
	0DH	invalid data (bad control string)

Programmer's Notes

- Subfunctions 02H and 03H provide a means of transferring control information of any type or length between an application program and a character-device driver. They do not necessarily result in any input to or output from the physical device itself.
- Subfunction 02H can be used to read control information about such features as device status, availability, and current output location. Subfunction 03H is often used to configure the driver or device for subsequent I/O; for example, it may be used to set the baud rate, word length, and parity for a serial communications adapter or to initialize a printer for a specific font, page length, and so on. The format of the control data passed by these subfunctions is driver specific and does not follow any standard.

- Character-device drivers are not required to support IOCTL Subfunctions 02H and 03H. Therefore, Subfunction 00H (Get Device Data) should be called before either Subfunction 02H or 03H to determine whether a device can process control strings. If bit 14 of the device data word returned by Subfunction 00H is set, the device driver supports IOCTL Subfunctions 02H and 03H.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

44H Subfunction 00H (Get Device Data)
44H Subfunction 04H (Receive Control Data from Block Device)
44H Subfunction 05H (Send Control Data to Block Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;     Function 44H, Subfunctions 02H,03H:                     ;
        ;                   IOCTL Character Device Control            ;
        ;                                                             ;
        ;     int ioctl_char_ctrl(recvflag,handle,pbuffer,nbytes)     ;
        ;         int   recvflag;                                     ;
        ;         int   handle;                                       ;
        ;         char *pbuffer;                                      ;
        ;         int   nbytes;                                       ;
        ;                                                             ;
        ;     Set recvflag = 0 to receive info, 1 to send.            ;
        ;                                                             ;
        ;     Returns -1 for error, otherwise returns number of       ;
        ;     bytes sent or received.                                 ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_char_ctrl,PUBLIC,<ds>
parmB   recvflag
parmW   handle
parmDP  pbuffer
parmW   nbytes
cBegin
        mov     al,recvflag     ; Get recvflag.
        and     al,1            ; Keep only lsb.
        add     al,2            ; AL = 02H for receive, 03H for send.
        mov     bx,handle       ; Get character-device handle.
        mov     cx,nbytes       ; Get number of bytes to receive/send.
        loadDP  ds,dx,pbuffer   ; Get pointer to buffer.
        mov     ah,44h          ; Set function code.
        int     21h             ; Call MS-DOS.
        jnc     iccx            ; Branch if no error.
        mov     ax,-1           ; Return -1 for all errors.
iccx:
cEnd
```

Interrupt 21H (33) — 2.0 and later
Function 44H (68) Subfunctions 04H and 05H

IOCTL: Receive Control Data from Block Device; Send Control Data to Block Device

Function 44H Subfunctions 04H and 05H respectively receive and send control strings from and to a block-oriented device driver.

To Call

AH	= 44H	
AL	= 04H	receive block-device data
	05H	send block-device data
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	
CX	= number of bytes to transfer	
DS:DX	= segment:offset of data buffer	

Returns

If function is successful:

Carry flag is clear.

AX = number of bytes transferred

If AL was 04H on call:

Buffer at DS:DX contains control data read from device driver.

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
05H	access denied
06H	invalid handle
0DH	invalid data (bad control string)

Programmer's Notes

- Subfunctions 04H and 05H provide a means of transferring control information of any type or length between an application program and a block-device driver. They do not necessarily result in any input to or output from the physical device itself.
- Control strings can be used to request driver operations that are not file oriented, such as tape rewind or disk eject (if hardware supported). The contents of such control strings are specific to individual device drivers and do not follow any standard format.

- Subfunction 04H can be used to obtain a code from the driver indicating device availability or status. Block devices that might use this subfunction include magnetic tape or tape cassette, CD ROM, and Small Computer Standard Interface (SCSI) devices.
- Block-device drivers are not required to support IOCTL Subfunctions 04H and 05H. If the driver does not support these subfunctions, error code 01H (Invalid Function) is returned.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

44H Subfunction 00H (Get Device Data)
44H Subfunction 02H (Receive Control Data from Character Device)
44H Subfunction 03H (Send Control Data to Character Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;   Function 44H, Subfunctions 04H,05H:                      ;
        ;                 IOCTL Block Device Control                 ;
        ;                                                            ;
        ;   int ioctl_block_ctrl(recvflag,drive_ltr,pbuffer,nbytes)  ;
        ;       int   recvflag;                                      ;
        ;       int   drive_ltr;                                     ;
        ;       char *pbuffer;                                       ;
        ;       int   nbytes;                                        ;
        ;                                                            ;
        ;   Set recvflag = 0 to receive info, 1 to send.             ;
        ;                                                            ;
        ;   Returns -1 for error, otherwise returns number of        ;
        ;   bytes sent or received.                                  ;
        ;                                                            ;
        ;************************************************************;

cProc   ioctl_block_ctrl,PUBLIC,<ds>
parmB   recvflag
parmB   drive_ltr
parmDP  pbuffer
parmW   nbytes
cBegin
        mov     al,recvflag     ; Get recvflag.
        and     al,1            ; Keep only lsb.
        add     al,4            ; AL = 04H for receive, 05H for send.
        mov     bl,drive_ltr    ; Get drive letter.
        or      bl,bl           ; Leave 0 alone.
        jz      ibc
        and     bl,not 20h      ; Convert letter to uppercase.
        sub     bl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
```

(more)

```
ibc:
        mov     cx,nbytes         ; Get number of bytes to receive/send.
        loadDP  ds,dx,pbuffer     ; Get pointer to buffer.
        mov     ah,44h            ; Set function code.
        int     21h               ; Call MS-DOS.
        jnc     ibcx              ; Branch if no error.
        mov     ax,-1             ; Return -1 for all errors.
ibcx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunctions 06H and 07H

2.0 and later

IOCTL: Check Input Status; Check Output Status

Function 44H Subfunctions 06H and 07H respectively determine whether a device or file associated with a handle is ready for input or output.

To Call

AH	= 44H	
AL	= 06H	get input status
	07H	get output status
BX	= handle number	

Returns

If function is successful:

Carry flag is clear.

AL = input or output status:

00H	not ready
FFH	ready

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
05H	access denied
06H	invalid handle
0DH	invalid data (bad control string)

Programmer's Notes

- The status returned in AL has the following meanings:

Status	Device	Input File	Output File
00H	Not ready	Pointer at EOF	Ready
0FFH	Ready	Ready	Ready

- Output files always return a ready condition, even if the disk is full or no disk is in the drive.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;   Function 44H, Subfunctions 06H,07H:                      ;
        ;                 IOCTL Input/Output Status                  ;
        ;                                                            ;
        ;   int ioctl_char_status(outputflag,handle)                 ;
        ;       int outputflag;                                      ;
        ;       int handle;                                          ;
        ;                                                            ;
        ;   Set outputflag = 0 for input status, 1 for output status. ;
        ;                                                            ;
        ;   Returns -1 for all errors, 0 for not ready,              ;
        ;   and 1 for ready.                                         ;
        ;                                                            ;
        ;************************************************************;

cProc   ioctl_char_status,PUBLIC
parmB   outputflag
parmW   handle
cBegin
        mov     al,outputflag   ; Get outputflag.
        and     al,1            ; Keep only lsb.
        add     al,6            ; AL = 06H for input status, 07H for output
                                ; status.
        mov     bx,handle       ; Get handle.
        mov     ah,44h          ; Set function code.
        int     21h             ; Call MS-DOS.
        jnc     isnoerr         ; Branch if no error.
        mov     ax,-1           ; Return error code.
        jmp     short isx
isnoerr:
        and     ax,1            ; Keep only lsb for return value.
isx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 08H

3.0 and later

IOCTL: Check If Block Device Is Removable

Function 44H Subfunction 08H checks whether the specified block device contains a removable storage medium, such as a floppy disk.

To Call

AH = 44H
AL = 08H
BL = drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)

Returns

If function is successful:

Carry flag is clear.

AX =	00H	storage medium removable
	01H	storage medium not removable

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
0FH	invalid drive

Programmer's Notes

- This subfunction exists to allow an application to check for a removable disk so that the user can be prompted to change disks if a required file is not found.
- When the carry flag is set, error code 01H normally means that MS-DOS did not recognize the function call. However, this error can also mean that the device driver does not support Subfunction 08H. In this case, MS-DOS assumes that the storage medium is not removable.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;      Function 44H, Subfunction 08H:                         ;
        ;                    IOCTL Removable Block Device Query       ;
        ;                                                             ;
        ;      int ioctl_block_fixed(drive_ltr)                       ;
        ;          int drive_ltr;                                     ;
        ;                                                             ;
        ;      Returns -1 for all errors, 1 if disk is fixed (not     ;
        ;      removable), 0 if disk is not fixed.                    ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_block_fixed,PUBLIC
parmB   drive_ltr
cBegin
        mov     bl,drive_ltr            ; Get drive letter.
        or      bl,bl                   ; Leave 0 alone.
        jz      ibch
        and     bl,not 20h              ; Convert letter to uppercase.
        sub     bl,'A'-1                ; Convert to drive number: 'A' = 1,
                                        ; 'B' = 2, etc.
ibch:
        mov     ax,4408h                ; Set function code, Subfunction 08H.
        int     21h                     ; Call MS-DOS.
        jnc     ibchx                   ; Branch if no error, AX = 0 or 1.
        cmp     ax,1                    ; Treat error code of 1 as "disk is
                                        ; fixed."
        je      ibchx
        mov     ax,-1                   ; Return -1 for other errors.
ibchx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 09H

3.1 and later

IOCTL: Check If Block Device Is Remote

Function 44H Subfunction 09H checks whether the specified block device is local (attached to the computer running the program) or remote (redirected to a network server).

To Call

AH = 44H
AL = 09H
BL = drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)

Returns

If function is successful:

Carry flag is clear.

DX = device attribute word:

bit 12 = 1	drive is remote
bit 12 = 0	drive is local

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
0FH	invalid drive

Programmer's Notes

- This subfunction should be avoided. Application programs should not distinguish between files on local and remote devices.
- When the carry flag is set, error code 01H can mean either that the function number is invalid or that the network has not been started.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;        Function 44H, Subfunction 09H:                       ;
        ;                      IOCTL Remote Block Device Query        ;
        ;                                                             ;
        ;        int ioctl_block_redir(drive_ltr)                     ;
        ;            int drive_ltr;                                   ;
        ;                                                             ;
        ;        Returns -1 for all errors, 1 if disk is remote       ;
        ;        (redirected), 0 if disk is local.                    ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_block_redir,PUBLIC
parmB   drive_ltr
cBegin
        mov     bl,drive_ltr    ; Get drive letter.
        or      bl,bl           ; Leave 0 alone.
        jz      ibr
        and     bl,not 20h      ; Convert letter to uppercase.
        sub     bl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
ibr:
        mov     ax,4409h        ; Set function code, Subfunction 09H.
        int     21h             ; Call MS-DOS.
        mov     ax,-1           ; Assume error.
        jc      ibrx            ; Branch if error, returning -1.
        inc     ax              ; Set AX = 0.
        test    dh,10h          ; Is bit 12 set?
        jz      ibrx            ; If not, disk is local: Return 0.
        inc     ax              ; Return 1 for remote disk.
ibrx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 0AH

3.1 and later

IOCTL: Check If Handle Is Remote

Function 44H Subfunction 0AH checks whether the handle in BX refers to a file or device that is local (on the computer running the program) or remote (redirected to a network server).

To Call

AH = 44H
AL = 0AH
BX = handle

Returns

If function is successful:

Carry flag is clear.

DX = attribute word for file or device:

bit 15 = 1	remote
bit 15 = 0	local

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
06H	invalid handle

Programmer's Notes

- Application programs should not distinguish between files on local and remote devices.
- When the carry flag is set, error code 01H can mean either that the function number is invalid or that the network has not been started.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;    Function 44H, Subfunction 0AH:                           ;
        ;                  IOCTL Remote Handle Query                  ;
        ;                                                             ;
        ;    int ioctl_char_redir(handle)                             ;
        ;        int handle;                                          ;
        ;                                                             ;
        ;    Returns -1 for all errors, 1 if device/file is remote    ;
        ;    (redirected), 0 if it is local.                          ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_char_redir,PUBLIC
parmW   handle
cBegin
        mov     bx,handle       ; Get handle.
        mov     ax,440ah        ; Set function code, Subfunction 0AH.
        int     21h             ; Call MS-DOS.
        mov     ax,-1           ; Assume error.
        jc      icrx            ; Branch on error, returning -1.
        inc     ax              ; Set AX = 0.
        test    dh,80h          ; Is bit 15 set?
        jz      icrx            ; If not, device/file is local:
                                ; Return 0.
        inc     ax              ; Return 1 for remote.
icrx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 0BH

3.1 and later

IOCTL: Change Sharing Retry Count

Function 44H Subfunction 0BH sets the number of times MS-DOS retries a disk operation after a failure caused by a file-sharing violation before it returns an error to the requesting process.

To Call

AH = 44H
AL = 0BH
CX = pause between retries
DX = number of retries

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:
    01H invalid function

Programmer's Notes

- The pause between retries is a machine-dependent value determined by the CPU and CPU clock speed. MS-DOS performs a delay loop that consists of 65,536 machine instructions for each iteration specified by the value in CX. The actual code is as follows:

  ```
  xor    cx,cx
  loop   $
  ```

  The default number of retries is 3, with a pause of one loop between retries—equivalent to calling this subfunction with DX = 3 and CX = 1.
- When the carry flag is set, error code 01H indicates either that the function code is invalid or that file sharing (SHARE.EXE) is not loaded.
- Subfunction 0BH can be used to tune the system if file-contention problems are likely to arise with shared files but are expected to last only a short while.
- If file contention is expected and if some applications will lock regions of the file for an appreciable period of time, the user may need to be informed. The best procedure is to set an initial small number of retries with a short pause period. After notifying the user, the application can wait a reasonable amount of time for file access by adjusting the retry or pause-period values.

- If a process uses this subfunction, it should restore the original default values for the pause and number of retries before terminating, to avoid unwanted effects on the behavior of subsequent processes.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;     Function 44H, Subfunction 0BH:                         ;
        ;                   IOCTL Change Sharing Retry Count         ;
        ;                                                            ;
        ;     int ioctl_set_retry(num_retries,wait_time)             ;
        ;         int num_retries;                                   ;
        ;         int wait_time;                                     ;
        ;                                                            ;
        ;     Returns 0 for success, otherwise returns error code.   ;
        ;                                                            ;
        ;************************************************************;

cProc   ioctl_set_retry,PUBLIC,<ds,si>
parmW   num_retries
parmW   wait_time
cBegin
        mov     dx,num_retries  ; Get parameters.
        mov     cx,wait_time
        mov     ax,440bh        ; Set function code, Subfunction 0BH.
        int     21h             ; Call MS-DOS.
        jc      isrx            ; Branch on error.
        xor     ax,ax
isrx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 0CH

3.2

IOCTL: Generic I/O Control for Handles

Function 44H Subfunction 0CH sets or gets the output iteration count for character-oriented devices. *See also* Appendix A: MS-DOS Version 3.3.

To Call

AH	= 44H	
AL	= 0CH	
BX	= handle	
CH	= category code:	
	05H	printer
CL	= function (minor) code:	
	45H	set iteration count
	65H	get iteration count
DS:DX	= segment:offset of 2-byte buffer receiving or containing iteration-count word	

Returns

If function is successful:

Carry flag is clear.

If CL was 65H on call:

DS:DX = segment:offset of iteration-count word

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	06H	invalid handle

Programmer's Notes

- The iteration count controls the number of times the device driver tries to send output to the printer before assuming that the device is busy.
- With MS-DOS version 3.2, only category code 05H (printer) is supported by this subfunction.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;   Function 44H, Subfunction 0CH:                           ;
        ;                 Generic IOCTL for Handles                  ;
        ;                                                            ;
        ;   int ioctl_char_generic(handle,category,function,pbuffer) ;
        ;       int   handle;                                        ;
        ;       int   category;                                      ;
        ;       int   function;                                      ;
        ;       int  *pbuffer;                                       ;
        ;                                                            ;
        ;   Returns 0 for success, otherwise returns error code.     ;
        ;                                                            ;
        ;************************************************************;

cProc   ioctl_char_generic,PUBLIC,<ds>
parmW   handle
parmB   category
parmB   function
parmDP  pbuffer
cBegin
        mov     bx,handle       ; Get device handle.
        mov     ch,category     ; Get category
        mov     cl,function     ; and function.
        loadDP  ds,dx,pbuffer   ; Get pointer to data buffer.
        mov     ax,440ch        ; Set function code, Subfunction 0CH.
        int     21h             ; Call MS-DOS.
        jc      icgx            ; Branch on error.
        xor     ax,ax
icgx:
cEnd
```

Interrupt 21H (33) Function 44H (68) Subfunction 0DH

3.2

IOCTL: Generic I/O Control for Block Devices

Function 44H Subfunction 0DH includes six input/output tasks, or minor functions, related to block-oriented devices. The tasks perform the following operations: set or get device parameters; write, read, format and verify, or verify tracks on a logical drive.

This entry covers general information on Subfunction 0DH. Details on each minor code are presented in subsequent entries.

To Call

AH	= 44H	
AL	= 0DH	
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	
CH	= category code:	
	08H	disk drive
CL	= function (minor) code:	
	40H	set parameters for block device
	41H	write track on logical drive
	42H	format and verify track on logical drive
	60H	get parameters for block device
	61H	read track on logical drive
	62H	verify track on logical drive
DS:DX	= segment:offset of parameter block	

Returns

If function is successful:

Carry flag is clear.

If CL was 60H or 61H on call:

DS:DX = segment:offset of parameter block

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	02H	invalid drive

Programmer's Notes

- Set Device Parameters (minor code 40H) must be used before an attempt to write, read, format, or verify a track on a logical drive. In general, the following sequence applies to any of these operations:

1. Get the current parameters (minor code 60H). Examine and save them.
2. Set the new parameters (minor code 40H).
3. Perform the task.
4. Retrieve the original parameters and restore them (minor code 40H).

- With version 3.2 of MS-DOS, only category code 08H is supported by this subfunction.
- Parameter blocks in the data buffer vary with the task being performed.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;    Function 44H, Subfunction 0DH:                           ;
        ;                  Generic IOCTL for Block Devices            ;
        ;                                                             ;
        ;    int ioctl_block_generic(drv_ltr,category,func,pbuffer)   ;
        ;        int   drv_ltr;                                       ;
        ;        int   category;                                      ;
        ;        int   func;                                          ;
        ;        char *pbuffer;                                       ;
        ;                                                             ;
        ;    Returns 0 for success, otherwise returns error code.     ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_block_generic,PUBLIC,<ds>
parmB   drv_ltr
parmB   category
parmB   func
parmDP  pbuffer
cBegin
        mov     bl,drv_ltr      ; Get drive letter.
        or      bl,bl           ; Leave 0 alone.
        jz      ibg
        and     bl,not 20h      ; Convert letter to uppercase.
        sub     bl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
ibg:
        mov     ch,category     ; Get category
        mov     cl,func         ; and function.
        loadDP  ds,dx,pbuffer   ; Get pointer to data buffer.
        mov     ax,440dh        ; Set function code, Subfunction 0DH.
        int     21h             ; Call MS-DOS.
        jc      ibgx            ; Branch on error.
        xor     ax,ax
ibgx:
cEnd
```

Interrupt 21H (33)
Function 44H (68) Subfunction 0DH
Minor Code 40H

IOCTL: Generic I/O Control for Block Devices: Set Device Parameters

Function 44H Subfunction 0DH minor code 40H sets device parameters in the parameter block pointed to by DS:DX.

To Call

AH	= 44H
AL	= 0DH
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)
CH	= category code: 08H disk drive
CL	= 40H
DS:DX	= segment:offset of parameter block

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code: 01H invalid function 02H invalid drive

Programmer's Notes

- The parameter block is formatted as follows:

Special-functions field: offset 00H, length 1 byte

Bit	Value	Meaning
0	0	Device BIOS parameter block (BPB) field contains a new default BPB.
	1	Use current BPB.
1	0	Use all fields in parameter block.
	1	Use track layout field only.

(more)

Special-functions field: offset 00H, length 1 byte *(continued)*

Bit	Value	Meaning
2	0	Sectors in track may be different sizes. (This setting should not be used.)
	1	Sectors in track are all same size; sector numbers range from 1 to the total number of sectors in the track. (This setting should always be used.)
3–7	0	Reserved.

Device type field: offset 01H, length 1 byte

Value	Meaning
00H	320/360 KB 5.25-inch disk
01H	1.2 MB 5.25-inch disk
02H	720 KB 3.5-inch disk
03H	Single-density 8-inch disk
04H	Double-density 8-inch disk
05H	Fixed disk
06H	Tape drive
07H	Other type of block device

Device attributes field: offset 02H, length 1 word

Bit	Value	Meaning
0	0	Removable storage medium
	1	Nonremovable storage medium
1	0	Door lock not supported
	1	Door lock supported
2–15	0	Reserved

Number of cylinders field: offset 04H, length 1 word

Meaning: Maximum number of cylinders supported; set by device driver

Media type field: offset 06H, length 1 byte

Value	Meaning
00H (default)	1.2 MB 5.25-inch disk
01H	320/360 KB 5.25-inch disk

Device BPB field: offset 07H, length 31 bytes.

Meaning: *See* Programmer's Note below.

If bit 0 = 0 in special-functions field, this field contains the new default BPB for the device.

If bit 0 = 1 in special-functions field, BPB in this field is returned by the device driver in response to subsequent Build BPB requests.

Track layout field: offset 26H, variable-length table

Length	Meaning
Word	Number of sectors in track
Word	Number of first sector in track*
Word	Size of first sector in track*
	•
	•
	•
Word	Number of last sector in track
Word	Size of last sector in track

*Sector number and sector size fields are repeated for each sector on the track. If bit 2 of the special-functions field is set, all sector sizes in the track layout field must be the same.

- The device BPB field is a 31-byte data structure. Information contained in the device BPB field describes the current disk and disk control areas. The device BPB field is formatted as follows:

Byte	Meaning
00–01H	Number of bytes per sector
02H	Number of sectors per allocation unit
03–04H	Number of sectors reserved, beginning at sector 0
05H	Number of file allocation tables (FATs)
06–07H	Maximum number of root-directory entries
08–09H	Total number of sectors
0AH	Media descriptor
0B–0CH	Number of sectors per FAT
0D–0EH	Number of sectors per track
0F–10H	Number of heads
11–14H	Number of hidden sectors
15–1FH	Reserved

- When Set Device Parameters (minor code 40H) is used, the number of cylinders should not be reset — some or all of the volume may become inaccessible.
- Subfunction 0DH minor code 60H performs the complementary action, Get Device Parameters.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

None

Interrupt 21H (33)
Function 44H (68) Subfunction 0DH
Minor Code 60H

IOCTL: Generic I/O Control for Block Devices: Get Device Parameters

Function 44H Subfunction 0DH minor code 60H gets device parameters in the parameter block pointed to by DS:DX.

To Call

AH	= 44H	
AL	= 0DH	
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	
CH	= category code:	
	08H	disk drive
CL	= 60H	
DS:DX	= segment:offset of parameter block	

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	02H	invalid drive

Programmer's Notes

- The parameter block is formatted as follows:

Special-functions field: offset 00H, length 1 byte

Bit	Value	Meaning
0	0	Returns default BIOS parameter block (BPB) for the device.
	1	Returns BPB that the Build BPB device driver call would return.
1–7	0	Reserved (must be zero).

Device type field: offset 01H, length 1 byte

Value	Meaning
00H	320/360 KB 5.25-inch disk
01H	1.2 MB 5.25-inch disk
02H	720 KB 3.5-inch disk
03H	Single-density 8-inch disk
04H	Double-density 8-inch disk
05H	Fixed disk
06H	Tape drive
07H	Other type of block device

Device attributes field: offset 02H, length 1 word

Bit	Value	Meaning
0	0	Removable storage medium
	1	Nonremovable storage medium
1	0	Door lock not supported
	1	Door lock supported
2–15	0	Reserved

Number of cylinders field: offset 04H, length 1 word

Meaning: Maximum number of cylinders supported; set by device driver

Media type field: offset 06H, length 1 byte

Value	Meaning
00H (default)	1.2 MB 5.25-inch disk
01H	320/360 KB 5.25-inch disk

Device BPB field: offset 07H, length 31 bytes

Meaning: *See* Programmer's Note below.

If bit 0 = 0 in special-functions field, this field contains the new default BPB for the device.

If bit 0 = 1 in special-functions field, BPB in this field is returned by the device driver in response to subsequent Build BPB requests.

Track layout field: offset 26H
Unused

- The device BPB field is a 31-byte data structure. Information contained in the device BPB field describes the current disk and disk control areas. The device BPB field is formatted as follows:

Byte	Meaning
00–01H	Number of bytes per sector
02H	Number of sectors per allocation unit
03–04H	Number of sectors reserved, beginning at sector 0
05H	Number of file allocation tables (FATs)
06–07H	Maximum number of root-directory entries
08–09H	Total number of sectors
0AH	Media descriptor
0B–0CH	Number of sectors per FAT
0D–0EH	Number of sectors per track
0F–10H	Number of heads
11–14H	Number of hidden sectors
15–1FH	Reserved

- Subfunction 0DH minor code 40H performs the complementary action, Set Device Parameters.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

None

Interrupt 21H (33)
Function 44H (68) Subfunction 0DH
Minor Codes 41H and 61H

IOCTL: Generic I/O Control for Block Devices: Write Track on Logical Drive; Read Track on Logical Drive

Function 44H Subfunction 0DH minor code 41H writes a track on the logical drive specified in BL and minor code 61H reads a track on the logical drive specified in BL, using information in the parameter block pointed to by DS:DX.

To Call

AH	= 44H	
AL	= 0DH	
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	
CH	= category code:	
	08H	disk drive
CL	= function (minor) code:	
	41H	write a track
	61H	read a track
DS:DX	= segment:offset of parameter block	

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	02H	invalid drive

Programmer's Notes

- The parameter block is formatted as follows:

Offset	Size	Meaning
00H	Byte	Special-functions field; must be 0.
01H	Word	Head field; contains number of disk head used for read/write.
03H	Word	Cylinder field; contains number of disk cylinder used for read/write.
05H	Word	First-sector field; contains number of first sector to read or write (first sector on track = sector 0).
07H	Word	Number-of-sectors field; contains number of sectors to transfer.
09H	Dword	Transfer address field; contains address of buffer to use for data transfer.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

None

Interrupt 21H (33)
Function 44H (68) Subfunction 0DH
Minor Codes 42H and 62H

IOCTL: Generic I/O Control for Block Devices: Format and Verify Track on Logical Drive; Verify Track on Logical Drive

Function 44H Subfunction 0DH minor code 42H formats and verifies a track on the specified logical drive and minor code 62H verifies a track on the specified logical drive, using information in the parameter block pointed to by DS:DX.

To Call

AH	= 44H	
AL	= 0DH	
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	
CH	= category code:	
	08H	disk drive
CL	= function (minor) code:	
	42H	format and verify
	62H	verify
DS:DX	= segment:offset of parameter block	

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function
	02H	invalid drive

Programmer's Notes

- The parameter block is formatted as follows:

Offset	Size	Meaning
00H	Byte	Special-functions field; must be 0.
01H	Word	Head field; contains number of disk head used for format/ verify.
03H	Word	Cylinder field; contains number of cylinder used for format/ verify.

- This driver subfunction allows the writing of generic formatting programs that are minimally hardware dependent.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

None

Interrupt 21H (33) 3.2
Function 44H (68) Subfunctions 0EH and 0FH

IOCTL: Get Logical Drive Map; Set Logical Drive Map

Function 44H Subfunction 0EH allows a process to determine whether more than one logical drive is assigned to a block device. Subfunction 0FH sets the next logical drive number that will be used to reference a block device.

To Call

AH	= 44H	
AL	= 0EH	get logical drive map
	0FH	set logical drive map
BL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)	

Returns

If function is successful:

Carry flag is clear.

AL = mapping code:

00H	only one letter assigned to the block device
01–1AH	logical drive letter (A through Z) mapped to block device

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function
0FH	invalid drive

Programmer's Notes

- If a drive has not been assigned a logical mapping with Function 44H Subfunction 0FH, the logical and physical drive references are the same. (The default is that logical drive A and physical drive A both refer to physical drive A.)
- If this function is used to map logical drives to physical drives, the result is similar to MS-DOS's treatment of a single physical drive as both A and B on a system with one floppy-disk drive. With MS-DOS version 3.2, however, the installable device driver DRIVER.SYS extends this type of physical/logical referencing to other drives. Therefore, processes can prompt for disks themselves, instead of using the prompt provided by MS-DOS.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;       Function 44H, Subfunctions 0EH, 0FH:                  ;
        ;                     IOCTL Get/Set Logical Drive Map         ;
        ;                                                             ;
        ;       int ioctl_drive_owner(setflag, drv_ltr)               ;
        ;           int setflag;                                      ;
        ;           int drv_ltr;                                      ;
        ;                                                             ;
        ;       Set setflag = 1 to change drive’s map, 0 to get       ;
        ;       current map.                                          ;
        ;                                                             ;
        ;       Returns -1 for all errors, otherwise returns          ;
        ;       the block device’s current logical drive letter.      ;
        ;                                                             ;
        ;*************************************************************;

cProc   ioctl_drive_owner,PUBLIC
parmB   setflag
parmB   drv_ltr
cBegin
        mov     al,setflag      ; Load setflag.
        and     al,1            ; Keep only lsb.
        add     al,0eh          ; AL = 0EH for get, 0FH for set.
        mov     bl,drv_ltr      ; Get drive letter.
        or      bl,bl           ; Leave 0 alone.
        jz      ido
        and     bl,not 20h      ; Convert letter to uppercase.
        sub     bl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
ido:
        mov     bh,0
        mov     ah,44h          ; Set function code.
        int     21h             ; Call MS-DOS.
        mov     ah,0            ; Clear high byte.
        jnc     idox            ; Branch if no error.
        mov     ax,-1-'A'       ; Return -1 for errors.
idox:
        add     ax,'A'          ; Return drive letter.
cEnd
```

Interrupt 21H (33) Function 45H (69)

2.0 and later

Duplicate File Handle

Function 45H obtains an additional handle for a currently open file or device.

To Call

AH = 45H
BX = handle for open file or device

Returns

If function is successful:

Carry flag is clear.

AX = new handle number

If function is not successful:

Carry flag is set.

AX = error code:

04H	too many open files
06H	invalid handle

Programmer's Notes

- The file pointer for the new handle is set to the same position as the pointer for the original handle. Any subsequent changes to the file are reflected in both handles. Thus, using either handle for a read or write operation moves the file pointer associated with both.
- Function 45H is often used to duplicate the handle assigned to standard input (0) or standard output (1) before a call to Function 46H (Force Duplicate File Handle). The handle forced by Function 46H can then be used for redirected input or output from or to a file or device.
- Another use for Function 45H is to keep a file open while its directory entry is being updated to reflect a change in length. If a new handle is obtained with Function 45H and then closed with Function 3EH (Close File), the directory and FAT entries for the file are updated. At the same time, because the original handle remains open, the file need not be reopened for additional read or write operations.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

46H (Force Duplicate File Handle)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;             Function 45H: Duplicate File Handle            ;
        ;                                                            ;
        ;             int dup_handle(handle)                         ;
        ;                 int handle;                                ;
        ;                                                            ;
        ;             Returns -1 for errors,                         ;
        ;             otherwise returns new handle.                  ;
        ;                                                            ;
        ;************************************************************;

cProc   dup_handle,PUBLIC
parmW   handle
cBegin
        mov     bx,handle       ; Get handle to copy.
        mov     ah,45h          ; Set function code.
        int     21h             ; Ask MS-DOS to duplicate handle.
        jnb     dup_ok          ; Branch if copy was successful.
        mov     ax,-1           ; Else return -1.
dup_ok:
cEnd
```

Interrupt 21H (33)
Function 46H (70)

2.0 and later

Force Duplicate File Handle

Function 46H forces the open handle specified in CX to track the same file or device specified by the handle in BX.

To Call

AH = 46H
BX = open handle to be duplicated
CX = open handle to be forced

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

04H	too many open files
06H	invalid handle

Programmer's Notes

- The handle in BX must refer either to an open file or to any of the five standard handles reserved by MS-DOS: standard input, standard output, standard error, standard auxiliary, or standard printer.
- If the handle in CX refers to an open file, the file is closed.
- The file pointer for the duplicate handle is set to the same position as the pointer for the original handle. Changing the position of either file pointer moves the pointer associated with the other handle as well.
- When used with Function 45H (Duplicate File Handle), Function 46H can be used to redirect input and output as follows:

  1. Duplicate the handle from which input or output will be redirected with Function 45H (Duplicate File Handle). Save the duplicated handle for later reference (Step 3).
  2. Call Function 46H, with the handle to be redirected from in the CX register and the handle to be redirected to in the BX register.
  3. To restore I/O redirection to its original state, call Function 46H again, with the redirected file handle from Step 2 in the CX register and the duplicated file handle from Step 1 in the BX register.

This procedure is normally used to redirect a standard device, but it can redirect any device referenced by handles.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

45H (Duplicate File Handle)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 46H: Force Duplicate File Handle         ;
        ;                                                            ;
        ;          int dup_handle2(existhandle,newhandle)            ;
        ;              int existhandle,newhandle;                    ;
        ;                                                            ;
        ;          Returns -1 for errors,                            ;
        ;          otherwise returns newhandle unchanged.            ;
        ;                                                            ;
        ;************************************************************;

cProc   dup_handle2,PUBLIC
parmW   existhandle
parmW   newhandle
cBegin
        mov     bx,existhandle  ; Get handle of existing file.
        mov     cx,newhandle    ; Get handle to copy into.
        mov     ah,46h          ; Close handle CX and then
        int     21h             ; duplicate BX's handle into CX.
        mov     ax,newhandle    ; Prepare return value.
        jnb     dup2_ok         ; Branch if close/copy was successful.
        mov     ax,-1           ; Else return -1.
dup2_ok:
cEnd
```

Interrupt 21H (33) Function 47H (71)

2.0 and later

Get Current Directory

Function 47H returns the path, excluding the drive and leading backslash, of the current directory for the specified drive.

To Call

AH	= 47H
DL	= drive number (0 = default drive, 1 = drive A, 2 = drive B, and so on)
DS:SI	= segment:offset of 64-byte buffer

Returns

If function is successful:

Carry flag is clear.

Buffer is filled in with ASCIIZ pathname.

If function is not successful:

Carry flag is set.

AX = error code:
  0FH invalid drive

Programmer's Notes

- The string representing the pathname is returned as a null-terminated ASCII string (ASCIIZ).
- This function does not return an error if the buffer is too small or is incorrectly identified. MS-DOS pathnames can be as long as 64 characters; if the buffer is less than 64 bytes, MS-DOS can overwrite sections of memory outside the buffer.
- The path returned by Function 47H starts at the root directory and fully specifies the path to the current directory but does not include a drive code or a leading backslash (\) character.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

3BH (Change Current Directory)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;            Function 47H: Get Current Directory              ;
        ;                                                             ;
        ;            int get_dir(drive_ltr,pbuffer)                   ;
        ;                int drive_ltr;                               ;
        ;                char *pbuffer;                               ;
        ;                                                             ;
        ;            Returns -1 for bad drive,                        ;
        ;            otherwise returns pointer to pbuffer.            ;
        ;                                                             ;
        ;*************************************************************;

cProc   get_dir,PUBLIC,<ds,si>
parmB   drive_ltr
parmDP  pbuffer
cBegin
        loadDP  ds,si,pbuffer   ; Get pointer to buffer.
        mov     dl,drive_ltr    ; Get drive number.
        or      dl,dl           ; Leave 0 alone.
        jz      gdir
        and     dl,not 20h      ; Convert letter to uppercase
        sub     dl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
gdir:
        mov     ah,47h          ; Set function code.
        int     21h             ; Call MS-DOS.
        mov     ax,si           ; Return pointer to buffer ...
        jnb     gd_ok
        mov     ax,-1           ; ... unless an error occurred.
gd_ok:
cEnd
```

Interrupt 21H (33) Function 48H (72)

2.0 and later

Allocate Memory Block

Function 48H allocates a block of memory, in paragraphs (1 paragraph = 16 bytes), to the requesting process.

To Call

AH = 48H
BX = number of paragraphs to allocate

Returns

If function is successful:

Carry flag is clear.

AX = segment address of base of allocated block

If function is not successful:

Carry flag is set.

AX = error code:

07H	memory control blocks damaged
08H	insufficient memory to allocate as requested

BX = size of largest available block (paragraphs)

Programmer's Notes

- If the allocation succeeds, the address returned in AX is the segment of the base of the block. This address would be copied to a segment register (usually DS or ES) to access the memory within the block.
- If the amount of memory requested is greater than the amount in any available contiguous block of memory, the number of paragraphs in the largest available memory block is returned in the BX register.
- The default memory-management strategy in MS-DOS is to choose the first contiguous block of memory that fits the request, no matter how good the fit. With MS-DOS versions 3.0 and later, however, the memory-management strategy can be altered with Function 58H (Get/Set Allocation Strategy).
- If a process actively allocates and frees blocks of memory, the transient program area (TPA) can become fragmented — that is, small blocks of memory can be orphaned because the memory-management strategy seeks contiguous blocks of memory.
- If a process writes to memory outside the limits of the allocated block, it can destroy control structures for other memory blocks. This could result in failure of subsequent memory-management functions, and it will cause MS-DOS to print an error message and halt when the process terminates.

- Initially, the MS-DOS loader allocates all available memory to .COM programs. Function 4AH (Resize Memory Block) can free memory for dynamic reallocation by a process or by its children.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

49H (Free Memory Block)
4AH (Resize Memory Block)
58H (Get/Set Allocation Strategy)

Example

```
                ;************************************************************;
                ;                                                            ;
                ;          Function 48H: Allocate Memory Block               ;
                ;                                                            ;
                ;          int get_block(nparas,pblocksegp,pmaxparas)        ;
                ;              int nparas,*pblockseg,*pmaxparas;             ;
                ;                                                            ;
                ;          Returns 0 if nparas are allocated OK and          ;
                ;          pblockseg has segment address of block,           ;
                ;          otherwise returns error code with pmaxparas       ;
                ;          set to maximum block size available.              ;
                ;                                                            ;
                ;************************************************************;

cProc   get_block,PUBLIC,ds
parmW   nparas
parmDP  pblockseg
parmDP  pmaxparas
cBegin
        mov     bx,nparas       ; Get size request.
        mov     ah,48h          ; Set function code.
        int     21h             ; Ask MS-DOS for memory.
        mov     cx,bx           ; Save BX.
        loadDP  ds,bx,pmaxparas
        mov     [bx],cx         ; Return result, assuming failure.
        jb      gb_err          ; Exit if error, leaving error code
                                ; in AX.
        loadDP  ds,bx,pblockseg
        mov     [bx],ax         ; No error, so store address of block.
        xor     ax,ax           ; Return 0.
gb_err:
cEnd
```

Interrupt 21H (33) Function 49H (73)

2.0 and later

Free Memory Block

Function 49H releases a block of memory previously allocated with Function 48H (Allocate Memory Block).

To Call

AH = 49H
ES = segment address of memory block to release

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

07H	memory control blocks damaged
09H	incorrect memory segment specified

Programmer's Notes

- The memory segment pointed to by ES:0000H must have been allocated by Function 48H (Allocate Memory Block).
- If a program has inadvertently damaged any of the system's memory control blocks by writing outside an allocated block, an attempt to free allocated memory results in error code 07H (memory control blocks damaged).
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

48H (Allocate Memory Block)
4AH (Resize Memory Block)
58H (Get/Set Allocation Strategy)

Example

```
        ;***********************************************************;
        ;                                                           ;
        ;               Function 49H: Free Memory Block             ;
        ;                                                           ;
        ;               int free_block(blockseg)                    ;
        ;                   int blockseg;                           ;
        ;                                                           ;
        ;               Returns 0 if block freed OK,                ;
        ;               otherwise returns error code.               ;
        ;                                                           ;
        ;***********************************************************;

cProc   free_block,PUBLIC
parmW   blockseg
cBegin
        mov     es,blockseg     ; Get block address.
        mov     ah,49h          ; Set function code.
        int     21h             ; Ask MS-DOS to free memory.
        jb      fb_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if successful.
fb_err:
cEnd
```

Interrupt 21H (33) Function 4AH (74)

2.0 and later

Resize Memory Block

Function 4AH adjusts the size of a previously allocated block of memory.

To Call

AH = 4AH
BX = new size of memory block, in paragraphs
ES = segment address of previously allocated memory block

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

07H	memory control blocks damaged
08H	insufficient memory to allocate as requested
09H	incorrect memory segment specified

BX = maximum number of paragraphs available (if an increase was requested)

Programmer's Notes

- Function 4AH can be used to change the size of a memory block previously allocated with Function 48H (Allocate Memory Block) or to modify the amount of memory originally allocated to a process by MS-DOS.
- If a process is denied an increase in the amount of memory it has been allocated, MS-DOS places the size of the largest contiguous block available in the BX register. The process can then notify the user of the problem and exit, or it can continue to operate in a reduced memory environment.
- Because the MS-DOS loader allocates all available memory to .COM programs, such a program should use Function 4AH immediately (with the segment address of its program segment prefix, or PSP) to release any memory that is not needed. This is mandatory if the .COM program will either allocate memory dynamically or use Function 4BH (Load and Execute Program) to load a child process or overlay.

  In addition, if Function 4AH is used to adjust the amount of memory allocated to a .COM program, the stack pointer must be adjusted so that it is within the limits of the program's revised memory allocation.

- If this function is used to shrink an allocated block, any memory above the new limit is not owned by the process and should never be used. If this function is used to expand an allocated block, the contents of memory above the old boundary are unpredictable and the memory should be initialized before use.
- Although it is not possible to predict how much memory-resident software and how many installable device drivers will be used on a computer system, Function 4AH can reliably determine the amount of memory available to an application.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

48H (Allocate Memory Block)
49H (Free Memory Block)
58H (Get/Set Allocation Strategy)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 4AH: Resize Memory Block                ;
        ;                                                            ;
        ;           int modify_block(nparas,blockseg,pmaxparas)      ;
        ;               int nparas,blockseg,*pmaxparas;              ;
        ;                                                            ;
        ;           Returns 0 if modification was a success,         ;
        ;           otherwise returns error code with pmaxparas      ;
        ;           set to max number of paragraphs available.       ;
        ;                                                            ;
        ;************************************************************;

cProc   modify_block,PUBLIC,ds
parmW   nparas
parmW   blockseg
parmDP  pmaxparas
cBegin
        mov     es,blockseg     ; Get block address.
        mov     bx,nparas       ; Get nparas.
        mov     ah,4ah          ; Set function code.
        int     21h             ; Ask MS-DOS to change block size.
        mov     cx,bx           ; Save BX.
        loadDP  ds,bx,pmaxparas
        mov     [bx],cx         ; Set pmaxparas, assuming failure.
        jb      mb_exit         ; Branch if size change error.
        xor     ax,ax           ; Return 0 if successful.
mb_exit:
cEnd
```

Interrupt 21H (33) Function 4BH (75)

2.0 and later

Load and Execute Program (EXEC)

Function 4BH, often called EXEC, loads a program file into memory and, optionally, executes the program. This function can also be used to load a program overlay.

To Call

AH	= 4BH	
AL	= 00H	load and execute program
	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.

With MS-DOS versions 2.x, all registers except CS and IP can be destroyed; with MS-DOS versions 3.x, registers are preserved.

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function (AL did not contain 00H or 03H)
02H	file not found
03H	path not found
05H	access denied
08H	insufficient memory
0AH	bad environment
0BH	bad format (AL = 00H only)

Programmer's Notes

- The pathname must be a null-terminated ASCII string (ASCIIZ).
- The handles for any files opened by the parent process before the call to Function 4BH are inherited by the child process, unless the parent specified otherwise in calling Function 3DH (Open File with Handle).

  All standard devices also remain open and available to the child process. Thus, the parent process can control the files used by the child process and control redirection for the child process.

- If AL = 00H, the parameter block is 14 bytes long and formatted in four parts, as follows:

Offset	Length	Meaning
00H	Word	Segment address of environment to be passed; 00H indicates child program inherits environment of the current process.
02H	Dword	Segment:offset address of command tail for the new program segment prefix (PSP). Command tail must be 128 bytes or fewer and formatted as a count byte followed by an ASCII string and terminated by a carriage return, as follows: `db      7,'a:mydoc',0Dh` The carriage return is not included in the count; the command tail is placed at offset 80H in the new process's PSP.
06H	Dword	Segment:offset address of an FCB to be copied to the default FCB position at offset 5CH in the new process's PSP.
0AH	Dword	Segment:offset address of an FCB to be copied to the default FCB position at offset 6CH in the new process's PSP.

If AL = 03H, the parameter block is 4 bytes long and formatted in two parts, as follows:

Offset	Length	Meaning
00H	Word	Segment address where the overlay is to be loaded.
02H	Word	Relocation factor to be applied to the code image (.EXE files only); not needed if the file is a .COM program or is data.

- The first 2 bytes of the parameter block for Function 4BH Subfunction 00H contain either the segment address for an environment block to be passed to the new process or zero. If the value is zero, the child process inherits an exact copy of the parent process's environment.

  The environment block must be aligned on a paragraph boundary (a multiple of 16 bytes). It can be as large as 32 KB, and it consists of a block of ASCIIZ strings, each in the following form:

parameter=value

For example:

```
db      'VERIFY=ON',0
```

The final string in the environment block is followed by a second zero byte. With MS-DOS versions 3.0 and later, the second zero is followed by a word containing a count and an ASCIIZ string containing the drive and pathname of the program file.

The environment passed to the child process allows the parent process to send it messages regarding the system state or control parameters. The pathname included with MS-DOS versions 3.0 and later enables the child process to determine where it was loaded from.

- If AL = 00H, MS-DOS creates a PSP for the new process and sets the terminate and Control-C addresses to the instruction in the parent process that follows the call to Function 4BH. If AL = 03H, no PSP is created.
- Before AL = 00H is used to load and execute a process, the system must contain enough free memory to accommodate the new process. Function 4AH (Resize Memory Block) should be used, if necessary, to reduce the amount of memory allocated to the parent process. If the parent is a .COM program, allocated memory *must* be reduced, because a .COM program is given ownership of all available memory when it is executed.

  If Function 4BH is called with AL = 03H, free memory is not a factor, because MS-DOS assumes the new process is being loaded into the calling process's own address space.
- If Function 4BH is called with AL = 00H, the child process remains in control until it executes an exit request, such as Function 4CH (Terminate Process with Return Code), or until Control-C or Control-Break is received or a critical error occurs and the user responds *Abort* to the *Abort, Retry, Ignore?* message.
- With MS-DOS versions 2.x, SS and SP must be saved in the current code segment before Function 4BH is invoked with AL = 00H. When the parent process regains control, all registers other than CS:IP and the stack will most likely have been changed by loading and executing the child process.
- Function 4BH with AL = 03H is useful for loading program overlays or for loading data to be used by the parent process (if that data requires relocation).
- If the child process that is executed attempts to remain resident through either Interrupt 27H or Interrupt 21H Function 31H (Terminate and Stay Resident), system memory becomes permanently fragmented and subsequent processes can fail because of lack of memory.
- The EXEC function (with AL = 00H) is commonly used to load a new copy of COMMAND.COM and then execute an MS-DOS command from within another program.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

31H (Terminate and Stay Resident)
4CH (Terminate Process with Return Code)
4DH (Get Return Code of Child Process)

Examples

```
        ;*************************************************************;
        ;                                                             ;
        ;          Function 4BH: Load and Execute Program             ;
        ;                                                             ;
        ;          int execute(pprogname,pcmdtail)                    ;
        ;              char *pprogname,*pcmdtail;                     ;
        ;                                                             ;
        ;          Returns 0 if program loaded, ran, and              ;
        ;          terminated successfully, otherwise returns         ;
        ;          error code.                                        ;
        ;                                                             ;
        ;*************************************************************;

sBegin  data
$cmdlen =       126
$cmd    db      $cmdlen+2 dup (?) ; Make space for command line, plus
                                ; 2 extra bytes for length and
                                ; carriage return.

$fcb    db      0               ; Make dummy FCB.
        db      'dummy   fcb'
        db      0,0,0,0

                                ; Here's the EXEC parameter block:
$epb    dw      0               ; 0 means inherit environment.
        dw      dataOFFSET $cmd ; Pointer to cmd line.
        dw      seg dgroup
        dw      dataOFFSET $fcb ; Pointer to FCB #1.
        dw      seg dgroup
        dw      dataOFFSET $fcb ; Pointer to FCB #2.
        dw      seg dgroup
sEnd    data
sBegin  code

$sp     dw      ?               ; Allocate space in code seg
$ss     dw      ?               ; for saving SS and SP.

Assumes ES,dgroup

cProc   execute,PUBLIC,<ds,si,di>
parmDP  pprogname
parmDP  pcmdtail
cBegin
        mov     cx,$cmdlen      ; Allow command line this long.
        loadDP  ds,si,pcmdtail  ; DS:SI = pointer to cmdtail string.
```

(more)

```
        mov     ax,seg dgroup:$cmd    ; Set ES = data segment.
        mov     es,ax
        mov     di,dataOFFSET $cmd+1  ; ES:DI = pointer to 2nd byte of
                                      ; our command-line buffer.
copycmd:
        lodsb                   ; Get next character.
        or      al,al           ; Found end of command tail?
        jz      endcopy         ; Exit loop if so.
        stosb                   ; Copy to command buffer.
        loop    copycmd
endcopy:
        mov     al,13
        stosb                   ; Store carriage return at
                                ; end of command.
        neg     cl
        add     cl,$cmdlen      ; CL = length of command tail.
        mov     es:$cmd,cl      ; Store length in command-tail buffer.

        loadDP  ds,dx,pprogname ; DS:DX = pointer to program name.
        mov     bx,dataOFFSET $epb ; ES:BX = pointer to parameter
                                   ; block.

        mov     cs:$ss,ss       ; Save current stack SS:SP (because
        mov     cs:$sp,sp       ; EXEC function destroys stack).
        mov     ax,4b00h        ; Set function code.
        int     21h             ; Ask MS-DOS to load and execute
                                ; program.
        cli                     ; Disable interrupts.
        mov     ss,cs:$ss       ; Restore stack.
        mov     sp,cs:$sp
        sti                     ; Enable interrupts.
        jb      ex_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
ex_err:
cEnd
sEnd    code


        ;*************************************************************;
        ;                                                             ;
        ;   Function 4BH: Load an Overlay Program                     ;
        ;                                                             ;
        ;   int load_overlay(pfilename,loadseg)                       ;
        ;       char *pfilename;                                      ;
        ;       int  loadseg;                                         ;
        ;                                                             ;
        ;   Returns 0 if program has been loaded OK,                  ;
        ;   otherwise returns error code.                             ;
        ;                                                             ;
        ;   To call an overlay function after it has been             ;
        ;   loaded by load_overlay(), you can use                     ;
        ;   a far indirect call:                                      ;
```

(more)

```
        ;                                                             ;
        ;   1. FTYPE (far *ovlptr)();                                 ;
        ;   2. *((unsigned *)&ovlptr + 1) = loadseg;                  ;
        ;   3. *((unsigned *)&ovlptr) = offset;                       ;
        ;   4. (*ovlptr)(arg1,arg2,arg3,...);                         ;
        ;                                                             ;
        ;   Line 1 declares a far pointer to a                        ;
        ;   function with return type FTYPE.                         ;
        ;                                                             ;
        ;   Line 2 stores loadseg into the segment                    ;
        ;   portion (high word) of the far pointer.                   ;
        ;                                                             ;
        ;   Line 3 stores offset into the offset                      ;
        ;   portion (low word) of the far pointer.                    ;
        ;                                                             ;
        ;   Line 4 does a far call to offset                          ;
        ;   bytes into the segment loadseg                            ;
        ;   passing the arguments listed.                             ;
        ;                                                             ;
        ;   To return correctly, the overlay  must end with a far     ;
        ;   return instruction.  If the overlay is                    ;
        ;   written in Microsoft C, this can be done by               ;
        ;   declaring the overlay function with the                   ;
        ;   keyword "far".                                            ;
        ;                                                             ;
        ;*************************************************************;

sBegin  data
                                ; The overlay parameter block:
$lob    dw      ?               ; space for load segment;
        dw      ?               ; space for fixup segment.
sEnd    data

sBegin  code

cProc   load_overlay,PUBLIC,<ds,si,di>
parmDP  pfilename
parmW   loadseg
cBegin
        loadDP  ds,dx,pfilename ; DS:DX = pointer to program name.
        mov     ax,seg dgroup:$lob ; Set ES = data segment.
        mov     es,ax
        mov     bx,dataOFFSET $lob ; ES:BX = pointer to parameter
                                   ; block.
        mov     ax,loadseg      ; Get load segment parameter.
        mov     es:[bx],ax      ; Set both the load and fixup
        mov     es:[bx+2],ax    ; segments to that segment.

        mov     cs:$ss,ss       ; Save current stack SS:SP (because
        mov     cs:$sp,sp       ; EXEC function destroys stack).
        mov     ax,4b03h        ; Set function code.
        int     21h             ; Ask MS-DOS to load the overlay.
        cli                     ; Disable interrupts.
```

(more)

```
        mov     ss,cs:$ss       ; Restore stack.
        mov     sp,cs:$sp
        sti                     ; Enable interrupts.
        jb      lo_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
lo_err:
cEnd
sEnd    code
```

Interrupt 21H (33) Function 4CH (76)

2.0 and later

Terminate Process with Return Code

Function 4CH terminates the current process with a return code and returns control to the calling (parent) process.

To Call

AH = 4CH
AL = return code

Returns

Nothing

Programmer's Notes

- When a process is terminated with Function 4CH, MS-DOS restores the termination-handler (Interrupt 22H), Control-C handler (Interrupt 23H), and critical error handler (Interrupt 24H) addresses from the program segment prefix, or PSP (offsets 0AH, 0EH, and 12H). MS-DOS also flushes the file buffers to disk, updates the disk directory, closes all files with open handles belonging to the terminated process, and then transfers control to the termination-handler address.
- On termination with Function 4CH, all memory owned by the process is freed.
- Function 4CH is the recommended method for terminating all processes—particularly sizable .EXE files—that do not stay resident. This function should be used in preference to the other termination methods (Interrupt 20H, Interrupt 21H Function 00H, near RET for .COM files, or a jump to PSP:0000H). Memory-resident programs should be terminated with Function 31H (Terminate and Stay Resident).
- A return code of 00H is customarily used to indicate that the process executed successfully; a nonzero return code is used to indicate that the process terminated because of an error or lack of resources—for example, the file could not be opened, the process could not be allocated sufficient memory, and so on.
- If the terminated process was invoked by a command line or batch file, control returns to COMMAND.COM and the transient portion of the command interpreter is reloaded, if necessary. If a batch file was in progress, execution continues with the next line of the file and the return code can be tested with an IF ERRORLEVEL statement. Otherwise, the command prompt is issued.

  If the terminated process was loaded by a process other than COMMAND.COM, the parent process can retrieve the child's return code with Function 4DH (Get Return Code of Child Process).
- In a networking environment running under MS-DOS version 3.1 or later, all file locks should be removed by the process before it calls Function 4CH to terminate.

Related Functions

00H (Terminate Process)
31H (Terminate and Stay Resident)
4DH (Get Return Code of Child Process)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;       Function 4CH: Terminate Process with Return Code     ;
        ;                                                            ;
        ;       int terminate(returncode)                            ;
        ;           int returncode;                                  ;
        ;                                                            ;
        ;       Does NOT return at all!                              ;
        ;                                                            ;
        ;************************************************************;

cProc   terminate,PUBLIC
parmB   returncode
cBegin
        mov     al,returncode   ; Set return code.
        mov     ah,4ch          ; Set function code.
        int     21h             ; Call MS-DOS to terminate process.
cEnd
```

Interrupt 21H (33) Function 4DH (77)

2.0 and later

Get Return Code of Child Process

Function 4DH retrieves the return code of a child process that was invoked with Function 4BH (Load and Execute Program) and terminated with either Function 31H (Terminate and Stay Resident) or Function 4CH (Terminate Process with Return Code).

To Call

AH = 4DH

Returns

AH = termination method:

00H	normal termination (Interrupt 20H, or Interrupt 21H Function 00H or Function 4CH)
01H	terminated by entry of Control-C
02H	terminated by critical error handler (for example, user responded *Abort* to *Abort, Retry, Ignore?* prompt)
03H	terminated and stayed resident (Interrupt 27H or Interrupt 21H Function 31H)

AL = return code passed by child process

If terminated with Interrupt 20H, Interrupt 21H Function 00H, or Interrupt 27H:

AL = 00H

Programmer's Notes

- Function 4DH can be used only once to retrieve the return code of a terminated process. Subsequent calls do not yield meaningful results.
- Function 4DH does not set the carry flag to indicate an error. If no previous child process exists, the information returned in AH and AL is undefined.

Related Functions

31H (Terminate and Stay Resident)
4CH (Terminate Process with Return Code)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;        Function 4DH: Get Return Code of Child Process       ;
        ;                                                             ;
        ;        int child_ret_code()                                 ;
        ;                                                             ;
        ;        Returns the return code of the last                  ;
        ;        child process.                                       ;
        ;                                                             ;
        ;*************************************************************;

cProc   child_ret_code,PUBLIC
cBegin
        mov     ah,4dh          ; Set function code.
        int     21h             ; Ask MS-DOS to return code.
        cbw                     ; Convert AL to a word.
cEnd
```

Interrupt 21H (33) Function 4EH (78)

2.0 and later

Find First File

Function 4EH searches the specified directory for the first matching entry.

To Call

AH	= 4EH
CX	= attribute word
DS:DX	= segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

Current disk transfer area (DTA) contains the following information about the file:

Offset	Length (bytes)	Value
00H	21	Reserved for use by MS-DOS in subsequent call to Function 4FH (Find Next File)
15H	1	File attribute
16H	2	Time of last write
18H	2	Date of last write
1AH	2	Low word of file size
1CH	2	High word of file size
1EH	13	Filename and extension in ASCIIZ form with blanks removed and period inserted between filename and extension

If function is not successful:

Carry flag is set.

AX = error code:

02H	file not found
03H	path not found
12H	no more files; no match found

Programmer's Notes

- The pathname must be a null-terminated ASCII string (ASCIIZ).

- The filename and extension portions of the pathname can contain the MS-DOS wildcards ? (match any character) and * (match all remaining characters).
- The DTA should be set with Function 1AH (Set DTA Address) before Function 4EH is called. If no DTA address is set, MS-DOS uses a default 128-byte buffer at offset 80H in the program segment prefix (PSP).
- The attribute word in CX controls the search as follows:
  - If the attribute word is 00H, only normal files are included in the search.
  - If the attribute word has any combination of bits 1, 2, and 4 (hidden, system, and subdirectory bits) set, the search includes normal files as well as files with any of the attributes specified.
  - If the attribute word has bit 3 set (volume-label bit), only a matching volume label is returned.
  - Bits 0 and 5 (read-only and archive bits) are ignored by Function 4EH.
- If Function 4FH (Find Next File) is used in conjunction with Function 4EH, the DTA must be preserved, because the first 21 bytes contain information needed by Function 4FH.
- The time at which the file was last written is returned as a binary value in a word formatted as follows:

Bits	Meaning
0–4	Number of seconds divided by 2
5–10	Minutes (0 through 59)
11–15	Hours, based on a 24-hour clock (0 through 23).

- The date on which the file was last written is returned as a binary value in a word formatted as follows:

Bits	Meaning
0–4	Day of the month
5–8	Month (1 = January, 2 = February, 3 = March, and so on)
9–15	Number of the year minus 1980

- Function 4EH is preferred to Function 11H (Find First File) because it fully supports pathnames.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

11H (Find First File)
12H (Find Next File)
1AH (Set DTA Address)
4FH (Find Next File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;               Function 4EH: Find First File                ;
        ;                                                            ;
        ;               int find_first(ppathname,attr)               ;
        ;                   char *ppathname;                         ;
        ;                   int  attr;                               ;
        ;                                                            ;
        ;               Returns 0 if a match was found,              ;
        ;               otherwise returns error code.                ;
        ;                                                            ;
        ;************************************************************;

cProc   find_first,PUBLIC,ds
parmDP  ppathname
parmW   attr
cBegin
        loadDP  ds,dx,ppathname ; Get pointer to pathname.
        mov     cx,attr         ; Get search attributes.
        mov     ah,4eh          ; Set function code.
        int     21h             ; Ask MS-DOS to look for a match.
        jb      ff_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
ff_err:
cEnd
```

Interrupt 21H (33)
Function 4FH (79)

2.0 and later

Find Next File

Function 4FH continues a search initiated by a previously successful call to Function 4EH (Find First File). The search is based on the pathname and attributes specified in the call to Function 4EH and uses information left in the current disk transfer area (DTA) by the call to Function 4EH or by a preceding call to Function 4FH.

To Call

AH = 4FH

DTA contains information from prior search with Function 4EH or Function 4FH.

Returns

If function is successful:

Carry flag is clear.

DTA is filled in as for a call to Function 4EH:

Offset	Length (bytes)	Value
00H	21	Reserved for use by MS-DOS in subsequent call to Function 4FH
15H	1	File attribute
16H	2	Time of last write
18H	2	Date of last write
1AH	2	Low word of file size
1CH	2	High word of file size
1EH	13	Filename and extension in ASCIIZ form with blanks removed and period inserted between filename and extension

If function is not successful:

Carry flag is set.

AX = error code:

12H	no more files, no match found, or no previous call to Function 4EH

Programmer's Notes

- If multiple calls to Function 4FH are used to find more than one matching file, the DTA setting (Function 1AH) and contents must be preserved because they provide information needed for continuing the search.
- The time at which the file was last written is returned as a binary value in a word formatted as follows:

Bits	Meaning
0–4	Number of seconds divided by 2
5–10	Minutes (0 through 59)
11–15	Hours, based on a 24-hour clock (0 through 23).

- The date on which the file was last written is returned as a binary value in a word formatted as follows:

Bits	Meaning
0–4	Day of the month
5–8	Month (1 = January, 2 = February, 3 = March, and so on)
9–15	Number of the year minus 1980

- Function 4FH is preferred to Function 12H (Find Next File) because it fully supports pathnames.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

11H (Find First File)
12H (Find Next File)
1AH (Set DTA Address)
4EH (Find First File)

Example

```
;*************************************************************;
;                                                             ;
;               Function 4FH: Find Next File                  ;
;                                                             ;
;               int find_next()                               ;
;                                                             ;
;               Returns 0 if a match was found,               ;
;               otherwise returns error code.                 ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   find_next,PUBLIC
cBegin
        mov     ah,4fh          ; Set function code.
        int     21h             ; Ask MS-DOS to look for the next
                                ; matching file.
        jb      fn_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
fn_err:
cEnd
```

Interrupt 21H (33)
Function 54H (84)

2.0 and later

Get Verify Flag

Function 54H returns the current value of the MS-DOS verify flag.

To Call

AH = 54H

Returns

AL = verify flag:

00H	verify off; no read after write operation
01H	verify on; read after write operation

Programmer's Notes

- The default state of the verify flag is 00H (off).
- The state of the verify flag can be changed either through a call to Function 2EH (Set/Reset Verify Flag) or by the user with the VERIFY ON and VERIFY OFF commands.

Related Function

Function 2EH (Set/Reset Verify Flag)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;            Function 54H: Get Verify Flag                    ;
        ;                                                             ;
        ;            int get_verify()                                 ;
        ;                                                             ;
        ;            Returns current value of verify flag.            ;
        ;                                                             ;
        ;*************************************************************;

cProc   get_verify,PUBLIC
cBegin
        mov     ah,54h          ; Set function code.
        int     21h             ; Read flag from MS-DOS.
        cbw                     ; Clear high byte of return value.

cEnd
```

Interrupt 21H (33) Function 56H (86)

2.0 and later

Rename File

Function 56H renames a file and/or moves it to a new location in the hierarchical directory structure.

To Call

AH	= 56H
DS:DX	= segment:offset of existing ASCIIZ pathname for file
ES:DI	= segment:offset of new ASCIIZ pathname for file

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

02H	file not found
03H	path not found
05H	access denied
11H	not the same device

Programmer's Notes

- The pathnames must be null-terminated ASCII strings (ASCIIZ).
- The directory paths specified in DS:DX and ES:DI need not be identical. Thus, specifying different directory paths effectively moves a file from one directory to another.
- Function 56H cannot be used to move a file to a different drive. Both the existing pathname and the new one must either contain the same drive identifier or default to the same drive.
- If Function 56H returns error code 05H, the cause can be any of the following:
  - The new pathname would move the file to the root directory, but the root directory is full.
  - A file with the new pathname already exists.
  - The user is on a network and has insufficient access to either the existing file or the new subdirectory.
- Unlike Function 17H (Rename File), Function 56H does not support the use of MS-DOS wildcard characters (? and *).

- Function 56H should not be used to rename open files. An open file should be closed with Function 10H (Close File with FCB) or 3EH (Close File) before Function 56H is called to rename it.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

17H (Rename File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;                Function 56H: Rename File                   ;
        ;                                                            ;
        ;                int rename(poldpath,pnewpath)               ;
        ;                    char *poldpath,*pnewpath;               ;
        ;                                                            ;
        ;                Returns 0 if file moved OK,                 ;
        ;                otherwise returns error code.               ;
        ;                                                            ;
        ;************************************************************;

cProc   rename,PUBLIC,<ds,di>
parmDP  poldpath
parmDP  pnewpath
cBegin
        loadDP  es,di,pnewpath  ; ES:DI = pointer to newpath.
        loadDP  ds,dx,poldpath  ; DS:DX = pointer to oldpath.
        mov     ah,56h          ; Set function code.
        int     21h             ; Ask MS-DOS to rename file.
        jb      rn_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
rn_err:
cEnd
```

Interrupt 21H (33) Function 57H (87)

2.0 and later

Get/Set Date/Time of File

Function 57H retrieves or sets the date and time of a file's directory entry.

To Call

AH = 57H
AL = 00H get date and time
01H set date and time
BX = handle number

If AL = 01H:

CX = time; binary value formatted as follows:

Bits	Meaning
0–4	Number of seconds divided by 2
5–10	Minutes (0 through 59)
11–15	Hours, based on a 24-hour clock (0 through 23)

DX = date; binary value formatted as follows:

Bits	Meaning
0–4	Day of the month (1 through 31)
5–8	Month (1 = January, 2 = February, 3 = March, and so on)
9–15	Year minus 1980

Returns

If function is successful:

Carry flag is clear.

If AL was 00H on call:

CX = time file was last modified; format as described above
DX = date file was last modified; format as described above

If function is not successful:

Carry flag is set.

AX = error code:
01H invalid function (AL not 00H or 01H)
06H invalid handle

Programmer's Notes

- Before the date and time in a file's directory entry can be retrieved or changed with Function 57H, a handle must be obtained by opening or creating the file using one of the following functions:
  - 3CH (Create File with Handle)
  - 3DH (Open File with Handle)
  - 5AH (Create Temporary File)
  - 5BH (Create New File)
- Use of Function 57H to retrieve the date and time of a file is preferable to examining the fields of an open FCB directly.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

2AH (Get Date)
2BH (Set Date)
2CH (Get Time)
2DH (Set Time)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;       Function 57H: Get/Set Date/Time of File              ;
        ;                                                            ;
        ;       long file_date_time(handle,func,packdate,packtime)   ;
        ;            int handle,func,packdate,packtime;              ;
        ;                                                            ;
        ;       Returns a long -1 for all errors, otherwise packs    ;
        ;       date and time into a long integer,                   ;
        ;       date in high word, time in low word.                 ;
        ;                                                            ;
        ;************************************************************;

cProc   file_date_time,PUBLIC
parmW   handle
parmB   func
parmW   packdate
parmW   packtime
cBegin
        mov     bx,handle       ; Get handle.
        mov     al,func         ; Get function: 0 = read, 1 = write.
        mov     dx,packdate     ; Get date (if present).
        mov     cx,packtime     ; Get time (if present).
        mov     ah,57h          ; Set function code.
        int     21h             ; Call MS-DOS.
```

(more)

```
        mov     ax,cx           ; Set DX:AX = date/time, assuming no
                                ; error.
        jnb     dt_ok           ; Branch if no error.
        mov     ax,-1           ; Return -1 for errors.
        cwd                     ; Extend the -1 into DX.
dt_ok:
cEnd
```

Interrupt 21H (33)
Function 58H (88)

3.0 and later

Get/Set Allocation Strategy

Function 58H retrieves or sets the method MS-DOS uses to allocate memory blocks for a process that issues a memory-allocation request.

To Call

AH = 58H
AL = 00H get allocation strategy
01H set allocation strategy

If AL = 01H:

BX = allocation strategy:

00H	use first (lowest available) block that fits
01H	use block that fits best
02H	use last (highest available) block that fits

Returns

If function is successful:

Carry flag is clear.

If AL was 00H on call:

AX = allocation-strategy code:

00H	first fit
01H	best fit
02H	last fit

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function (AL not 00H or 01H)

Programmer's Notes

- Allocation strategies determine how MS-DOS finds and allocates a block of memory to an application that issues a memory-allocation request with either Function 48H (Allocate Memory Block) or Function 4AH (Resize Memory Block).

  The three strategies are carried out as follows:

  – First fit (the default): MS-DOS works upward from the lowest available block and allocates the first block it encounters that is large enough to satisfy the request for memory. This strategy is followed consistently, even if the block allocated is much larger than required.

  - Best fit: MS-DOS searches all available memory blocks and then allocates the smallest block that satisfies the request, regardless of its location in the empty-block chain. This strategy maximizes the use of dynamically allocated memory at a slight cost in speed of allocation.
  - Last fit (the reverse of first fit): MS-DOS works downward from the highest available block and allocates the first block it encounters that is large enough to satisfy the request for memory. This strategy is followed consistently, even if the block allocated is much larger than required.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

48H (Allocate Memory Block)
4AH (Resize Memory Block)

Example

```
        ;**************************************************************;
        ;                                                              ;
        ;           Function 58H: Get/Set Allocation Strategy          ;
        ;                                                              ;
        ;           int alloc_strategy(func,strategy)                  ;
        ;               int func,strategy;                             ;
        ;                                                              ;
        ;           Strategies:                                        ;
        ;                   0: First fit                               ;
        ;                   1: Best fit                                ;
        ;                   2: Last fit                                ;
        ;                                                              ;
        ;           Returns -1 for all errors, otherwise               ;
        ;           returns the current strategy.                      ;
        ;                                                              ;
        ;**************************************************************;

cProc   alloc_strategy,PUBLIC
parmB   func
parmW   strategy
cBegin
        mov     al,func         ; AL = get/set selector.
        mov     bx,strategy     ; BX = new strategy (for AL = 01H).
        mov     ah,58h          ; Set function code.
        int     21h             ; Call MS-DOS.
        jnb     no_err          ; Branch if no error.
        mov     ax,-1           ; Return -1 for all errors.
no_err:
cEnd
```

Interrupt 21H (33)
Function 59H (89)

3.0 and later

Get Extended Error Information

Function 59H returns extended error information, including a suggested response, for the function call immediately preceding it.

To Call

AH = 59H
BX = 00H

Returns

AX = extended error code:

00H	no error encountered
01H	invalid function number
02H	file not found
03H	path not found
04H	too many files open; 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
10H	attempt to remove current directory
11H	device not the same
12H	no more files
13H	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
1AH	non-MS-DOS disk
1BH	sector not found

1CH	printer out of paper
1DH	write fault
1EH	read fault
1FH	general failure
20H	sharing violation
21H	lock violation
22H	invalid disk change
23H	FCB unavailable
24H	sharing buffer exceeded
25–31H	reserved
32H	unsupported network request
33H	remote machine 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	remote adapt incompatible
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
49–4FH	reserved
50H	file already exists
51H	reserved
52H	cannot make directory
53H	failure on Interrupt 24H (critical error)
54H	out of structures
55H	already assigned
56H	invalid password
57H	invalid parameter
58H	net write fault

BH = error class:

01H	out of resource (such as storage)
02H	temporary situation, expected to end; not an error
03H	authorization problem
04H	internal error in system software
05H	hardware failure
06H	system-software failure, such as missing or incorrect configuration files; not the fault of the active process
07H	application-program error
08H	file or item not found
09H	file or item of invalid format or type or otherwise unsuitable
0AH	file or item interlocked
0BH	drive contains wrong disk, disk has bad spot, or other problem with storage medium
0CH	already exists
0DH	unknown

BL = suggested action:

01H	perform a reasonable number of retries before prompting user to choose Abort or Ignore in response to error message
02H	perform a reasonable number of retries, with pauses between, before prompting user to choose Abort or Ignore in response to error message
03H	prompt user to enter corrected information, such as drive letter or filename
04H	clean up and exit application
05H	exit immediately without cleanup
06H	ignore; informational error
07H	prompt user to remove cause of error (for example, change disks) and then retry

CH = location of error:

01H	unknown
02H	block device
03H	network
04H	serial device
05H	memory related

Programmer's Notes

- The extended error codes returned by Function 59H correspond to the error values returned in AX by functions in MS-DOS versions 2.0 and later that set the carry flag on error. Versions 2.x of MS-DOS, however, provide a smaller set of error codes (01H through 12H) than do later versions.

  Thus, although Function 59H itself is not available in versions of MS-DOS earlier than 3.0, the matching of error codes to earlier versions helps ensure downward compatibility. Function 59H was also designed to be open-ended so that additional error codes could be incorporated as needed. As a result, processes should remain flexible in their use of this function and should not rely on a fixed set of code numbers for error detection.

- Function 59H is useful in the following situations:
  - When MS-DOS encounters a hardware-related error condition and shifts control to an Interrupt 24H handler that has been created by the programmer
  - When a handle-related function sets the carry flag to indicate an error or when an FCB-related function indicates an error by returning 0FFH in the AL register
- If a function call results in an error, Function 59H returns meaningful information only if it is the next call to MS-DOS. An intervening call to another MS-DOS function, whether explicit or indirect, causes the error value for the unsuccessful function to be lost.
- Unlike most MS-DOS functions, Function 59H alters some registers that are not used to return results: CL, DX, SI, DI, ES, and DS. These registers must be preserved before a call to Function 59H if their contents are needed later.

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 59H: Get Extended Error Information      ;
        ;                                                            ;
        ;          int extended_error(err,class,action,locus)        ;
        ;              int *err;                                     ;
        ;              char *class,*action,*locus;                   ;
        ;                                                            ;
        ;          Return value is same as err.                      ;
        ;                                                            ;
        ;************************************************************;

cProc   extended_error,PUBLIC,<ds,si,di>
parmDP  perr
parmDP  pclass
parmDP  paction
parmDP  plocus
cBegin
        push    ds              ; Save DS.
        xor     bx,bx
        mov     ah,59h          ; Set function code.
        int     21h             ; Request error info from MS-DOS.
        pop     ds              ; Restore DS.
        loadDP  ds,si,perr      ; Get pointer to err.
        mov     [si],ax         ; Store err.
        loadDP  ds,si,pclass    ; Get pointer to class.
        mov     [si],bh         ; Store class.
        loadDP  ds,si,paction   ; Get pointer to action.
        mov     [si],bl         ; Store action.
        loadDP  ds,si,plocus    ; Get pointer to locus.
        mov     [si],ch         ; Store locus.
cEnd
```

Interrupt 21H (33) Function 5AH (90)

3.0 and later

Create Temporary File

Function 5AH uses the system clock to create a unique filename, appends the filename to the specified path, opens the temporary file, and returns a file handle that can be used for subsequent file operations.

To Call

AH	= 5AH	
CX	= file attribute:	
	00H	normal file
	01H	read-only file
	02H	hidden file
	04H	system file
DS:DX	= segment:offset of ASCIIZ path, ending with a backslash character (\) and followed by 13 bytes of memory (to receive the generated filename)	

Returns

If function is successful:

Carry flag is clear.

AX	= handle
DS:DX	= segment:offset of full pathname for temporary file

If function is not successful:

Carry flag is set.

AX	= error code:	
	03H	path not found
	04H	too many open files; no handle available
	05H	access denied

Programmer's Notes

- Only the drive and path to use for the new file should be specified in the buffer pointed to by DS:DX. The function appends an eight-character filename that is generated from the system time.
- Function 5AH is valuable in such situations as print spooling on a network, where temporary files are created by many users.
- The input string representing the path for the temporary file must be a null-terminated ASCII string (ASCIIZ).
- In networking environments running under MS-DOS version 3.1 or later, MS-DOS opens the temporary file in compatibility mode.

- MS-DOS does not delete temporary files; applications must do this for themselves.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

16H (Create File with FCB)
3CH (Create File with Handle)
5BH (Create New File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;             Function 5AH: Create Temporary File            ;
        ;                                                            ;
        ;             int create_temp(ppathname,attr)                ;
        ;                 char *ppathname;                           ;
        ;                 int attr;                                  ;
        ;                                                            ;
        ;             Returns -1 if file was not created,            ;
        ;             otherwise returns file handle.                 ;
        ;                                                            ;
        ;************************************************************;

cProc   create_temp,PUBLIC,ds
parmDP  ppathname
parmW   attr
cBegin
        loadDP  ds,dx,ppathname ; Get pointer to pathname.
        mov     cx,attr         ; Set function code.
        mov     ah,5ah          ; Ask MS-DOS to make a new file with
                                ; a unique name.
        int     21h             ; Ask MS-DOS to make a tmp file.
        jnb     ct_ok           ; Branch if MS-DOS returned handle.
        mov     ax,-1           ; Else return -1.
ct_ok:
cEnd
```

Interrupt 21H (33) Function 5BH (91)

3.0 and later

Create New File

Function 5BH creates a new file with the specified pathname. This function operates like Function 3CH (Create File with Handle) but fails if the pathname references a file that already exists.

To Call

AH	= 5BH	
CX	= file attribute:	
	00H	normal file
	01H	read-only file
	02H	hidden file
	04H	system file
DS:DX	= segment:offset of ASCIIZ pathname	

Returns

If function is successful:

Carry flag is clear.

AX	= handle

If function is not successful:

Carry flag is set.

AX	= error code:	
	03H	path not found
	04H	too many open files; no handle available
	05H	access denied
	50H	file already exists

Programmer's Notes

- The pathname must be a null-terminated ASCII string (ASCIIZ).
- In networking environments running under MS-DOS version 3.1 or later, the file is opened in compatibility mode. Function 5BH fails, however, if the user does not have Create access to the directory that is to contain the file.
- Function 5BH can be used to implement semaphores in the form of files across a local area network or in a multitasking environment. If the function succeeds, the semaphore has been acquired. To release the semaphore, the application simply deletes the file.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

16H (Create File with FCB)
3CH (Create File with Handle)
5AH (Create Temporary File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 5BH: Create New File                     ;
        ;                                                            ;
        ;          int create_new(ppathname,attr)                    ;
        ;              char *ppathname;                              ;
        ;              int attr;                                     ;
        ;                                                            ;
        ;          Returns -2 if file already exists,                ;
        ;                  -1 for all other errors,                  ;
        ;                  otherwise returns file handle.            ;
        ;                                                            ;
        ;************************************************************;

cProc   create_new,PUBLIC,ds
parmDP  ppathname
parmW   attr
cBegin
        loadDP  ds,dx,ppathname ; Get pointer to pathname.
        mov     cx,attr         ; Get new file's attribute.
        mov     ah,5bh          ; Set function code.
        int     21h             ; Ask MS-DOS to make a new file.
        jnb     cn_ok           ; Branch if MS-DOS returned handle.
        mov     bx,-2
        cmp     al,80           ; Did file already exist?
        jz      ae_err          ; Branch if so.
        inc     bx              ; Change -2 to -1.
ae_err:
        mov     ax,bx           ; Return error code.
cn_ok:
cEnd
```

Interrupt 21H (33)
Function 5CH (92)

3.0 and later

Lock/Unlock File Region

Function 5CH enables a process running in a networking or multitasking environment to lock or unlock a range of bytes in an open file.

To Call

AH	= 5CH	
AL	= 00H	lock region
	01H	unlock region
BX	= handle	
CX:DX	= 4-byte integer specifying beginning of region to be locked or unlocked (offset in bytes from beginning of file)	
SI:DI	= 4-byte integer specifying length of region (measured in bytes)	

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function (AL not 00H or 01H or file sharing not loaded)
	06H	invalid handle
	21H	lock violation
	24H	sharing buffer exceeded

Programmer's Notes

- A process that either closes a file containing a locked region or terminates with the file open leaves the file in an undefined state. Under either condition, MS-DOS might handle the file erratically. If the process can be terminated by Interrupt 23H (Control-C) or 24H (critical error), these interrupts should be trapped so that any locked regions in files can be unlocked before the process terminates.
- Locking a portion of a file with Function 5CH denies all other processes both read and write access to the specified region of the file. This restriction also applies when open file handles are passed to a child process with Function 4BH (Load and Execute Program). Duplicate file handles created with Function 45H (Duplicate File Handle) and 46H (Force Duplicate File Handle), however, are allowed access to locked regions of a file within the current process.
- Locking a region that goes beyond the end of a file does not cause an error.

- Function 5CH is useful primarily in ensuring that competing programs or processes do not interfere while a record is being updated. Locking at the file level is provided by the sharing parameter in Function 3DH (Open File with Handle).
- Function 5CH can also be used to check the lock status of a file. If an attempt to lock a needed portion of a file fails and error code 21H is returned in the AX register, the region is already locked by another process.
- Any region locked with a call to Function 5CH must also be unlocked, and the same 4-byte integer values must be used for each operation. Two adjacent regions of a file cannot be locked separately and then be unlocked with a single unlock call. If the region to unlock does not correspond exactly to a locked region, Function 5CH returns error code 21H.
- The length of time needed to hold locks can be minimized with the transaction-oriented programming model. This concept requires defining and performing an update in a uniform manner: Assert lock, read data, change data, remove lock.
- If file sharing is not loaded, an application receives a 01H (function number invalid) error status when it attempts to lock a file. An immediate call to Function 59H returns the error locus as an unknown or a serial device.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

45H (Duplicate File Handle)
46H (Force Duplicate File Handle)
4BH (Load and Execute Program) [EXEC]

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;            Function 5CH: Lock/Unlock File Region            ;
        ;                                                             ;
        ;            int locks(handle,onoff,start,length)             ;
        ;                int handle,onoff;                            ;
        ;                long start,length;                           ;
        ;                                                             ;
        ;            Returns 0 if operation was successful,           ;
        ;            otherwise returns error code.                    ;
        ;                                                             ;
        ;*************************************************************;

cProc   locks,PUBLIC,<si,di>
parmW   handle
parmB   onoff
parmD   start
parmD   length
```

(more)

```
cBegin
        mov     al,onoff        ; Get lock/unlock flag.
        mov     bx,handle       ; Get file handle.
        les     dx,start        ; Get low word of start.
        mov     cx,es           ; Get high word of start.
        les     di,length       ; Get low word of length.
        mov     si,es           ; Get high word of length.
        mov     ah,5ch          ; Set function code.
        int     21h             ; Make lock/unlock request.
        jb      lk_err          ; Branch on error.
        xor     ax,ax           ; Return 0 if no error.
lk_err:
cEnd
```

Interrupt 21H (33) Function 5EH (94) Subfunction 00H

3.1 and later

Network Machine Name/Printer Setup: Get Machine Name

If Microsoft Networks is running, Function 5EH Subfunction 00H retrieves the network name of the local computer.

To Call

AH	= 5EH
AL	= 00H
DS:DX	= segment:offset of 16-byte buffer

Returns

If function is successful:

Carry flag is clear.

CH	= validity of machine name:	
	00H	invalid
	nonzero	valid
CL	= NETBIOS number assigned to machine name	
DS:DX	= segment:offset of ASCIIZ machine name	

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function; Microsoft Networks not running

Programmer's Notes

- The NETBIOS number in CL and the name at DS:DX are valid only if the value returned in CH is nonzero.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

5FH (Get/Make Assign List Entry)

Example

None

Interrupt 21H (33) Function 5EH (94) Subfunctions 02H and 03H

3.1 and later

Network Machine Name/Printer Setup: Set Printer Setup;
Get Printer Setup

Function 5EH Subfunctions 02H and 03H respectively set and get the setup string that MS-DOS adds to the beginning of a file sent to a network printer.

To Call

AH = 5EH
AL = 02H set printer setup string
03H get printer setup string
BX = assign-list index number (obtained with Function 5FH Subfunction 02H)

If AL = 02H:

CX = length of setup string in bytes (64 bytes maximum)
DS:SI = segment:offset of ASCII setup string

If AL = 03H:

ES:DI = segment:offset of 64-byte buffer to receive string

Returns

If function is successful:

Carry flag is clear.

If AL was 03H on call:

CX = length of printer setup string in bytes
ES:DI = segment:offset of ASCII printer setup string

If function is not successful:

Carry flag is set.

AX = error code:
01H invalid subfunction

Programmer's Notes

- Function 5EH Subfunctions 02H and 03H enable multiple users on a network to configure a shared printer as required. The assign-list number is an index to a table that identifies the printer as a device on the network. A process can determine the assign-list number for the printer by using Function 5FH Subfunction 02H (Get Assign-List Entry).
- Error code 01H in the AX register may indicate either that Microsoft Networks is not running or that an invalid subfunction was selected.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

5FH (Get/Make Assign-List Entry)

Example

```
                ;*************************************************************;
                ;                                                             ;
                ;       Function 5EH Subfunction 02H:                         ;
                ;                    Set Printer Setup                        ;
                ;                                                             ;
                ;       int printer_setup(index,pstring,len)                  ;
                ;           int   index;                                      ;
                ;           char *pstring;                                    ;
                ;           int   len;                                        ;
                ;                                                             ;
                ;       Returns 0, otherwise returns -1 for all errors.       ;
                ;                                                             ;
                ;*************************************************************;

cProc   printer_setup,PUBLIC,<ds,si>
parmW   index
parmDP  pstring
parmW   len
cBegin
        mov     bx,index        ; BX = index of a net printer.
        loadDP  ds,si,pstring   ; DS:SI = pointer to string.
        mov     cx,len          ; CX = length of string.
        mov     ax,5e02h        ; Set function code.
        int     21h             ; Set printer prefix string.
        mov     al,0            ; Assume no error.
        jnb     ps_ok           ; Branch if no error,
        mov     al,-1           ; Else return -1.
ps_ok:
        cbw
cEnd
```

Interrupt 21H (33) Function 5FH (95) Subfunction 02H

3.1 and later

Get/Make Assign-List Entry: Get Assign-List Entry

Function 5FH Subfunction 02H obtains the local and remote (network) names of a device. To find the names, MS-DOS uses the device's user-assigned index number (set with Function 5FH Subfunction 03H) to search a table of redirected devices on the network. Microsoft Networks must be running with file sharing loaded for this subfunction to operate successfully.

To Call

AH	= 5FH
AL	= 02H
BX	= assign-list index number
DS:SI	= segment:offset of 16-byte buffer for local (device) name
ES:DI	= segment:offset of 128-byte buffer to receive remote (network) name

Returns

If function is successful:

Carry flag is clear.

BH	= device status:	
	00H	valid device
	01H	invalid device
BL	= device type:	
	03H	printer
	04H	drive
CX	= user data	
DS:SI	= segment:offset of ASCIIZ string representing local device name	
ES:DI	= segment:offset of ASCIIZ string representing network name	

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function or Microsoft Networks not running
	12H	no more files

Programmer's Notes

- All strings returned by this subfunction are null-terminated ASCII strings (ASCIIZ).
- A successful call to this subfunction destroys the contents of the DX and BP registers.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

5EH Subfunction 00H (Get Machine Name)

Example

```
                ;************************************************************;
                ;                                                            ;
                ;    Function 5FH Subfunction 02H:                           ;
                ;                  Get Assign-List Entry                     ;
                ;                                                            ;
                ;    int get_alist_entry(index,                              ;
                ;           plocalname,premotename,                          ;
                ;           puservalue,ptype)                                ;
                ;        int  index;                                         ;
                ;        char *plocalname;                                   ;
                ;        char *premotename;                                  ;
                ;        int  *puservalue;                                   ;
                ;        int  *ptype;                                        ;
                ;                                                            ;
                ;    Returns 0 if the requested assign-list entry is found,  ;
                ;    otherwise returns error code.                           ;
                ;                                                            ;
                ;************************************************************;

cProc   get_alist_entry,PUBLIC,<ds,si,di>
parmW   index
parmDP  plocalname
parmDP  premotename
parmDP  puservalue
parmDP  ptype
cBegin
        mov     bx,index        ; Get list index.
        loadDP  ds,si,plocalname  ; DS:SI = pointer to local name
                                  ; buffer.
        loadDP  es,di,premotename ; ES:DI = pointer to remote name
                                  ; buffer.
        mov     ax,5f02h        ; Set function code.
        int     21h             ; Get assign-list entry.
        jb      ga_err          ; Exit on error.
        xor     ax,ax           ; Else return 0.
        loadDP  ds,si,puservalue ; Get address of uservalue.
        mov     [si],cx         ; Store user value.
        loadDP  ds,si,ptype     ; Get address of type.
        mov     bh,0
        mov     [si],bx         ; Store device type to type.
ga_err:
cEnd
```

Interrupt 21H (33) Function 5FH (95) Subfunction 03H

3.1 and later

Get/Make Assign-List Entry: Make Assign-List Entry

Function 5FH Subfunction 03H redirects a local printer or disk drive to a network device and establishes an assign-list index number for the redirected device. Microsoft Networks must be running with file sharing loaded for this subfunction to operate successfully.

To Call

AH	= 5FH	
AL	= 03H	
BL	= device type:	
	03H	printer
	04H	drive
CX	= user data	
DS:SI	= segment:offset of 16-byte ASCIIZ local device name	
ES:DI	= segment:offset of 128-byte ASCIIZ remote (network) device name and password in the form	

machine name\pathname, null,password, null

For example:

```
string  db      '\\mymach\wp',0,'blibbet',0
```

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function or Microsoft Networks not running
	03H	path not found
	05H	access denied
	08H	insufficient memory
	0FH	redirection paused on server
	12H	no more files

Programmer's Notes

- The strings used by this subfunction must be null-terminated ASCII strings (ASCIIZ). The ASCIIZ string pointed to by ES:DI (the destination, or remote, device) cannot be more than 128 bytes including the password, which can be a maximum of 8 characters. If the password is omitted, the pathname must be followed by 2 null bytes.

- If BL = 03H, the string pointed to by DS:SI must be one of the following printer names: PRN, LPT1, LPT2, or LPT3. If the call is successful, output is redirected to a network print spooler, which must be named in the destination string. For printer redirection, MS-NET intercepts Interrupt 17H (BIOS Printer I/O). When redirection for a printer is canceled, all printing is sent to the first local printer (LPT1).

  If BL = 04H, the string pointed to by DS:SI can be a drive letter followed by a colon, such as E:, or it can be a null string. If the string represents a valid drive, a successful call redirects drive requests to the network directory named in the destination string. If DS:SI points to a null string, MS-DOS attempts to provide access to the network directory named in the destination string without redirecting any device.
- Only printer and disk devices are supported in MS-DOS versions 3.1 and later. COM1 and COM2 are not supported for network redirection, nor are the standard output or standard error devices supported.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

5EH Subfunction 00H (Get Machine Name)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;    Function 5FH Subfunction 03H:                           ;
        ;                Make Assign-List Entry                      ;
        ;    int add_alist_entry(psrcname,pdestname,uservalue,type)  ;
        ;        char *psrcname,*pdestname;                          ;
        ;        int  uservalue,type;                                ;
        ;                                                            ;
        ;    Returns 0 if new assign-list entry is made, otherwise   ;
        ;    returns error code.                                     ;
        ;                                                            ;
        ;************************************************************;

cProc   add_alist_entry,PUBLIC,<ds,si,di>
parmDP  psrcname
parmDP  pdestname
parmW   uservalue
parmW   type
cBegin
        mov     bx,type         ; Get device type.
        mov     cx,uservalue    ; Get uservalue.
        loadDP  ds,si,psrcname  ; DS:SI = pointer to source name.
        loadDP  es,di,pdestname ; ES:DI = pointer to destination name.
        mov     ax,5f03h        ; Set function code.
        int     21h             ; Make assign-list entry.
        jb      aa_err          ; Exit if there was some error.
        xor     ax,ax           ; Else return 0.
aa_err:
cEnd
```

Int 21H (33)
Function 5FH (95) Subfunction 04H

3.1 and later

Get/Make Assign-List Entry: Cancel Assign-List Entry

Function 5FH Subfunction 04H cancels the redirection of a local device to a network device previously established with Function 5FH Subfunction 03H (Make Assign-List Entry). Microsoft Networks must be running with file sharing loaded for this subfunction to operate successfully.

To Call

AH	= 5FH
AL	= 04H
DS:SI	= segment:offset of ASCIIZ device name or path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

01H	invalid function or Microsoft Networks not running
03H	path not found
05H	access denied
08H	insufficient memory
0FH	redirection paused on server
12H	no more files

Programmer's Notes

- The string pointed to by DS:SI must be a null-terminated ASCII string (ASCIIZ). This string can be any one of the following:
  - The letter, followed by a colon, of a redirected local drive. This function restores the drive letter to its original, physical meaning.
  - The name of a redirected printer: PRN, LPT1, LPT2, LPT3, or its machine-specific equivalent. This function restores the printer name to its original, physical meaning at the local workstation.
  - A string, beginning with two backslashes (\\) followed by the name of a network directory. This function terminates the connection between the local workstation and the directory specified in the string.

- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

5EH Subfunction 00H (Get Machine Name)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;    Function 5FH Subfunction 04H:                           ;
        ;                 Cancel Assign-List Entry                   ;
        ;                                                            ;
        ;    int cancel_alist_entry(psrcname)                        ;
        ;        char *psrcname;                                     ;
        ;                                                            ;
        ;    Returns 0 if assignment is canceled, otherwise returns  ;
        ;    error code.                                             ;
        ;                                                            ;
        ;************************************************************;

cProc   cancel_alist_entry,PUBLIC,<ds,si>
parmDP  psrcname
cBegin
        loadDP  ds,si,psrcname  ; DS:SI = pointer to source name.
        mov     ax,5f04h        ; Set function code.
        int     21h             ; Cancel assign-list entry.
        jb      ca_err          ; Exit on error.
        xor     ax,ax           ; Else return 0.
ca_err:
cEnd
```

Interrupt 21H (33)
Function 62H (98)

3.0 and later

Get Program Segment Prefix Address

Function 62H gets the segment address of the program segment prefix (PSP) for the current process.

To Call

AH = 62H

Returns

BX = segment address of PSP for current process

Programmer's Notes

- The PSP is constructed by MS-DOS at the base of the memory allocated for a .COM or .EXE program being loaded into memory by the EXEC function, 4BH (Load and Execute Program). The PSP is 100H bytes and contains information useful to an executing program, including
  - The command tail
  - Default file control blocks (FCBs)
  - A pointer to the program's environment block
  - Previous addresses for MS-DOS Control-C, critical error, and terminate handlers
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

None

Example

```
;*************************************************************;
;                                                             ;
;       Function 62H: Get Program Segment Prefix Address      ;
;                                                             ;
;       int get_psp()                                         ;
;                                                             ;
;       Returns PSP segment.                                  ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   get_psp,PUBLIC
cBegin
        mov     ah,62h          ; Set function code.
        int     21h             ; Get PSP address.
        mov     ax,bx           ; Return it in AX.
cEnd
```

Interrupt 21H (33) Function 63H (99)

2.25

Get Lead Byte Table

Function 63H, available only in MS-DOS version 2.25, includes three subfunctions that support 2-byte-per-character alphabets such as Kanji and Hangeul (Japanese and Korean characters sets). Subfunction 00H obtains the address of the legal lead byte ranges for the character sets; Subfunctions 01H and 02H set or obtain the value of the interim console flag, which determines whether interim characters are returned by certain console system calls.

To Call

AH	= 63H	
AL	= 00H	get lead byte table address
	01H	set or clear interim console flag
	02H	get interim console flag

If AL = 01H:

DL	= interim console flag:	
	00H	clear
	01H	set

Returns

If function is successful:

Carry flag is clear.

If AL was 00H on call:

DS:SI = segment:offset of lead byte table

If AL was 02H on call:

DL = value of interim console flag

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function

Programmer's Notes

- Function 63H does not necessarily preserve any registers other than SS:SP, so register values should be saved before a call to this function. To avoid saving registers repeatedly, a process can either copy the table or save the pointer to the table for later use.

- The lead byte table contains pairs of bytes that represent the inclusive boundary values for the lead bytes of the specified alphabet. Because of the way bytes are ordered by the 8086 microprocessor family, the values must be read as byte values, not as word values.
- If the interim console flag is set (DL = 01H) by a program through a call to Function 63H, the following functions return interim character information on request:
  - 07H (Character Input Without Echo)
  - 08H (Unfiltered Character Input Without Echo)
  - 0BH (Check Keyboard Status)
  - 0CH (Flush Buffer, Read Keyboard), if Function 07H or 08H is requested in AL

Related Functions

None

Example

```
        ;************************************************************;
        ;                                                            ;
        ;   Function 63H: Get Lead Byte Table                        ;
        ;                                                            ;
        ;   char far *get_lead_byte_table()                          ;
        ;                                                            ;
        ;   Returns far pointer to table of lead bytes for multibyte ;
        ;   characters.  Will work only in MS-DOS 2.25!              ;
        ;                                                            ;
        ;************************************************************;

cProc   get_lead_byte_table,PUBLIC,<ds,si>
cBegin
        mov     ax,6300h        ; Set function code.
        int     21h             ; Get lead byte table.
        mov     dx,ds           ; Return far pointer in DX:AX.
        mov     ax,si
cEnd
```

Interrupt 22H (34)

1.0 and later

Terminate Routine Address

The machine interrupt vector for Interrupt 22H (memory locations 0000:0088H through 0000:008BH) contains the address of the routine that receives control when the currently executing program terminates by means of Interrupt 20H, Interrupt 27H, or Interrupt 21H Function 00H, 31H, or 4CH.

To Call

This interrupt should never be issued directly.

Returns

Nothing

Programmer's Note

- The address in this vector is copied into offsets 0AH through 0DH of the program segment prefix (PSP) when a program is loaded but before it begins executing. The address is restored from the PSP (in case it was modified by the application) as part of MS-DOS's termination handling.

Example

None

Interrupt 23H (35)

1.0 and later

Control-C Handler Address

The machine interrupt vector for Interrupt 23H (memory locations 0000:008CH through 0000:008FH) contains the address of the routine that receives control when a Control-C (also Control-Break on IBM PC compatibles) is detected during any character I/O function and, if the Break flag is on, during most other MS-DOS function calls.

To Call

This interrupt should never be issued directly.

Returns

Nothing

Programmer's Notes

- The address in this vector is copied into offsets 0EH through 11H of the program segment prefix (PSP) when a program is loaded but before it begins executing. The address is restored from the PSP (in case it was modified by the application) as part of MS-DOS's termination handling.
- The initialization code for an application can use Interrupt 21H Function 25H (Set Interrupt Vector) to reset the Interrupt 23H vector to point to its own routine for Control-C handling. By installing its own Control-C handler, the program can avoid being terminated as a result of keyboard entry of a Control-C or Control-Break.
- When a Control-C is detected and the program's Interrupt 23H handler receives control, MS-DOS sets all registers to the original values they had when the function call that is being interrupted was made. The program's interrupt handler can then do any of the following:
  - Set a local flag for later inspection by the application (or take any other appropriate action) and then perform a return from interrupt (IRET) to return control to MS-DOS. (All registers must be preserved.) The MS-DOS function in progress is then restarted and proceeds to completion, and control finally returns to the application in the normal manner.
  - Take appropriate action and then perform a far return (RET FAR) to give control back to MS-DOS. MS-DOS uses the state of the carry flag to determine what action to take: If the carry flag is set, the application is terminated; if the carry flag is clear, the application continues in the normal manner.
  - Retain control by transferring to an error-handling routine within the application and then resume execution or take other appropriate action, never performing a RET FAR or IRET to end the interrupt-handling sequence. This option causes no harm to the system.
- Any MS-DOS function call can be used within the body of an Interrupt 23H handler.

Example

None

Interrupt 24H (36)

1.0 and later

Critical Error Handler Address

The machine interrupt vector for Interrupt 24H (memory locations 0000:0090H through 0000:0093H) contains the address of the routine that receives control when a critical error (usually a hardware error) is detected.

To Call

This interrupt should never be issued directly.

Returns

Nothing

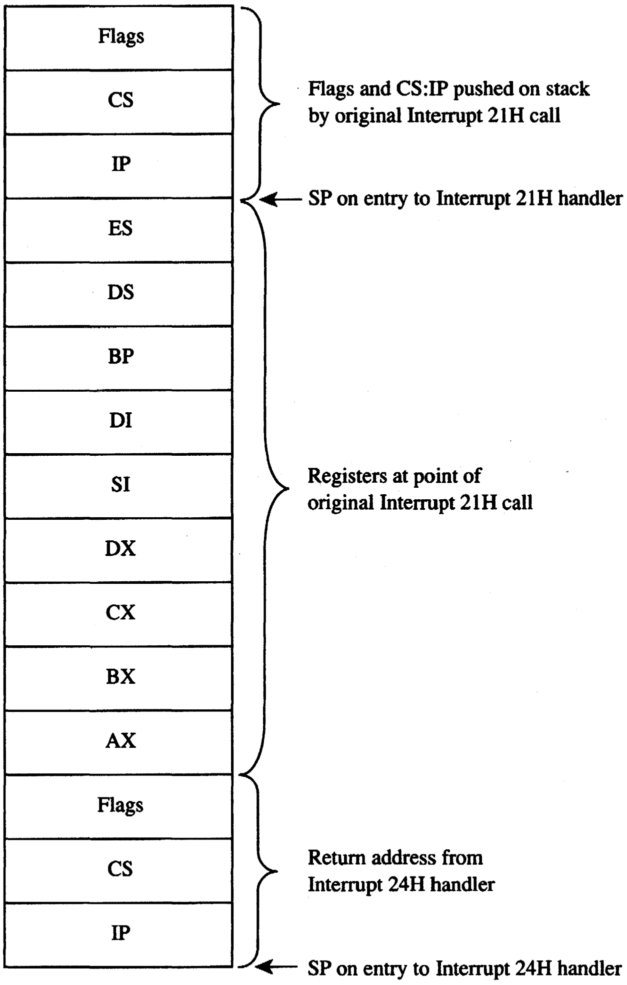
Programmer's Notes

- The address of this vector is copied into offsets 12H through 15H of the program segment prefix (PSP) when a program is loaded but before it begins executing. The address is restored from the PSP (in case it was modified by the application) as part of MS-DOS's termination handling.
- On entry to the critical error interrupt handler, bit 7 of register AH is clear (0) if the error was a disk I/O error; otherwise, it is set (1). BP:SI contains the address of a device-header control block from which additional information can be obtained. Interrupts are disabled. MS-DOS sets up the registers for a retry operation and one of the following error codes is in the lower byte of the DI register (the upper byte is undefined):

Code	Meaning
00H	Write-protect error
01H	Unknown unit
02H	Drive not ready
03H	Unknown command
04H	Data error (bad CRC)
05H	Bad request structure length
06H	Seek error
07H	Unknown media type
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0FH	Invalid disk change

These are the same error codes returned by the device drivers in the request header.

- On a disk error, MS-DOS retries the operation three times before transferring to the Interrupt 24H handler.
- On entry to the Interrupt 24H handler, the stack is set up as follows:


Flags
CS
IP
ES
DS
BP
DI
SI
DX
CX
BX
AX
Flags
CS
IP
Flags and CS:IP pushed on stack by original Interrupt 21H call
SP on entry to Interrupt 21H handler
Registers at point of original Interrupt 21H call
Return address from Interrupt 24H handler
SP on entry to Interrupt 24H handler

- Interrupt 24H handlers must preserve the SS, SP, DS, ES, BX, CX, and DX registers. Only Interrupt 21H Functions 01H through 0CH, 30H, and 59H can be used by an Interrupt 24H handler; other calls will destroy the MS-DOS stack and its ability to retry or ignore an error.

- Before issuing a RETURN FROM INTERRUPT (IRET), the Interrupt 24H handler should place an action code in AL that will be interpreted by MS-DOS as follows:

Code	Meaning
00H	Ignore error.
01H	Retry operation.
02H	Terminate program through Interrupt 23H.
03H	Fail system call in progress (versions 3.1 and later).

- If an Interrupt 24H routine returns to the user program rather than to MS-DOS, it must restore the user program's registers, removing all but the last three words from the stack, and issue an IRET. Control returns to the instruction immediately following the Interrupt 21H function call that resulted in an error. This leaves MS-DOS in an unstable state until a call is made to an Interrupt 21H function higher than 0CH.

Example

None

Interrupt 25H (37)

1.0 and later

Absolute Disk Read

Interrupt 25H provides direct linkage to the MS-DOS BIOS module to read data from a logical disk sector into a specified memory location.

To Call

AL	= drive number (0 = drive A, 1 = drive B, and so on)
CX	= number of sectors to read
DX	= starting relative (logical) sector number
DS:BX	= segment:offset of disk transfer area (DTA)

Returns

If operation is successful:

Carry flag is clear.

If operation is not successful:

Carry flag is set.

AX	= error code

Programmer's Notes

- Interrupt 25H might destroy all registers except the segment registers.
- When Interrupt 25H returns, the CPU flags originally pushed onto the stack by the INT 25H instruction are still on the stack. The stack must be cleared by a POPF or ADD SP,2 instruction to prevent uncontrolled stack growth and to make accessible any other values that were pushed onto the stack before the call to Interrupt 25H.
- Logical sector numbers are zero based and are obtained by numbering each disk sector sequentially from track 0, head 0, sector 1 and continuing until the last sector on the disk is counted. The head number is incremented before the track number. Because of interleaving, logically adjacent sectors might not be physically adjacent for some types of disks.
- The lower byte of the error code (AL) is the same error code that is returned in the lower byte of DI when an Interrupt 24H is issued. The upper byte (AH) contains one of the following codes:

Code	Meaning
80H	Device failed to respond
40H	Seek operation failure
20H	Controller failure

(more)

Code	Meaning
10H	Data error (bad CRC)
08H	Direct memory access (DMA) failure
04H	Requested sector not found
03H	Write-protect fault
02H	Bad address mark
01H	Bad command

- ***Warning:*** Interrupt 25H bypasses the MS-DOS file system. This function must be used with caution to avoid damaging the disk structure.

Example

```
;*****************************************************************;
;                                                                 ;
;       Interrupt 25H: Absolute Disk Read                         ;
;                                                                 ;
;       Read logical sector 1 of drive A into the memory area     ;
;       named buff. (On most MS-DOS floppy disks, this sector     ;
;       contains the beginning of the file allocation table.)     ;
;                                                                 ;
;*****************************************************************;

        mov     al,0            ; Drive A.
        mov     cx,1            ; Number of sectors.
        mov     dx,1            ; Beginning sector number.
        mov     bx,seg buff     ; Address of buffer.
        mov     ds,bx
        mov     bx,offset buff
        int     25h             ; Request disk read.
        jc      error           ; Jump if read failed.
        add     sp, 2           ; Clear stack.
        .
        .
        .
error:                          ; Error routine goes here.
        .
        .
        .
buff    db      512 dup (?)
```

Interrupt 26H (38)

1.0 and later

Absolute Disk Write

Interrupt 26H provides direct linkage to the MS-DOS BIOS module to write data from a specified memory buffer to a logical disk sector.

To Call

AL	= drive number (0 = drive A, 1 = drive B, and so on)
CX	= number of sectors to write
DX	= starting relative (logical) sector number
DS:BX	= segment:offset of disk transfer area (DTA)

Returns

If operation is successful:

Carry flag is clear.

If operation is not successful:

Carry flag is set.

AX	= error code

Programmer's Notes

- When Interrupt 26H returns, the CPU flags originally pushed onto the stack by the INT 26H instruction are still on the stack. The stack must be cleared by a POPF or ADD SP,2 instruction to prevent uncontrolled stack growth and to make accessible any other values that were pushed on the stack before the call to Interrupt 26H.
- Logical sector numbers are zero based and are obtained by numbering each disk sector sequentially from track 0, head 0, sector 1 and continuing until the last sector on the disk is counted. The head number is incremented before the track number. Because of interleaving, logically adjacent sectors might not be physically adjacent for some types of disks.
- The lower byte of the error code (AL) is the same error code that is returned in the lower byte of DI when an Interrupt 24H is issued. The upper byte (AH) contains one of the following codes:

Code	Meaning
80H	Device failed to respond
40H	Seek operation failure
20H	Controller failure
10H	Data error (bad CRC)

(more)

Code	Meaning
08H	Direct memory access (DMA) failure
04H	Requested sector not found
03H	Write-protect fault
02H	Bad address mark
01H	Bad command

- ***Warning:*** Interrupt 26H bypasses the MS-DOS file system. This function must be used with caution to avoid damaging the disk structure.

Example

```
;*****************************************************************;
;                                                                 ;
;       Interrupt 26H: Absolute Disk Write                        ;
;                                                                 ;
;       Write the contents of the memory area named buff          ;
;       into logical sector 3 of drive C.                         ;
;                                                                 ;
;       WARNING: Verbatim use of this code could damage           ;
;       the file structure of the fixed disk. It is meant         ;
;       only as a general guide. There is, unfortunately,         ;
;       no way to give a really safe example of this interrupt.   ;
;                                                                 ;
;*****************************************************************;

        mov     al,2            ; Drive C.
        mov     cx,1            ; Number of sectors.
        mov     dx,3            ; Beginning sector number.
        mov     bx,seg buff     ; Address of buffer.
        mov     ds,bx
        mov     bx,offset buff
        int     26h             ; Request disk write.
        jc      error           ; Jump if write failed.
        add     sp,2            ; Clear stack.
        .
        .
        .
error:                          ; Error routine goes here.
        .
        .
        .
buff    db      512 dup (?)     ; Data to be written to disk.
```

Interrupt 27H (39)

1.0 and later

Terminate and Stay Resident

Interrupt 27H terminates execution of the currently executing program but reserves part or all of its memory so that it will not be overlaid by the next transient program to be loaded.

To Call

DX = offset of last byte plus 1 (relative to the program segment prefix, or PSP) of program to be protected
CS = segment address of PSP

Returns

Nothing

Programmer's Notes

- In response to an Interrupt 27H call, MS-DOS takes the following actions:
  - Restores the termination vector (Interrupt 22H) from PSP:000AH.
  - Restores the Control-C vector (Interrupt 23H) from PSP:000EH.
  - With MS-DOS versions 2.0 and later, restores the critical error handler vector (Interrupt 24H) from PSP:0012H.
  - Transfers to the termination handler address.
- If the program is returning to COMMAND.COM rather than to another program, control transfers first to COMMAND.COM's resident portion, which reloads COMMAND.COM's transient portion (if necessary) and passes it control. If a batch file is in progress, the next line of the file is then fetched and interpreted; otherwise, a prompt is issued for the next user command.
- This interrupt is typically used to allow user-written drivers or interrupt handlers to be loaded as ordinary .COM or .EXE programs and then remain resident. Subsequent entrance to the code is by means of a hardware or software interrupt.
- The maximum amount of memory that can be reserved with this interrupt is 64 KB. Therefore, Interrupt 27H should be used only for applications that must run under MS-DOS versions 1.x.

  With versions 2.0 and later, the preferred method to terminate and stay resident is to use Interrupt 21H Function 31H, which allows the program to reserve more than 64 KB of memory and does not require CS to contain the PSP address.
- Interrupt 27H should not be called by .EXE programs that are loaded into the high end of memory (that is, linked with the /HIGH switch), because this would reserve the memory that is ordinarily used by the transient portion of COMMAND.COM. If COMMAND.COM cannot be reloaded, the system will fail.

- Because execution of Interrupt 27H results in the restoration of the terminate routine (Interrupt 22H), Control-C (Interrupt 23H), and critical error (Interrupt 24H) vectors, it cannot be used to permanently install a user-written critical error handler.
- Interrupt 27H does not work correctly when DX contains values in the range FFF1H through FFFFH. In this case, MS-DOS discards the high bit of the contents of DX, resulting in 32 KB less resident memory than was actually requested by the program.

Example

```
;****************************************************************;
;                                                                ;
;       Interrupt 27H: Terminate and Stay Resident               ;
;                                                                ;
;       Exit and stay resident, reserving enough memory          ;
;       to protect the program's code and data.                  ;
;                                                                ;
;****************************************************************;

Start:   .
         .
         .
         mov     dx,offset pgm_end  ; DX = bytes to reserve.
         int     27h                ; Terminate, stay resident.
         .
         .
         .
pgm_end  equ     $
         end     start
```

Interrupt 2FH (47)

2.0 and later

Multiplex Interrupt

Interrupt 2FH with AH = 01H submits a file to the print spooler, removes a file from the print spooler's queue of pending files, or obtains the status of the printer. Other values for AH are used by various MS-DOS extensions, such as APPEND.

To Call

AH	= 01H	print spooler call
AL	= 00H	get installed status
	01H	submit file to be printed
	02H	remove file from print queue
	03H	cancel all files in queue
	04H	hold print jobs for status read
	05H	end hold for status read

If AL is 01H:

DS:DX	= segment:offset of packet address

If AL is 02H:

DS:DX	= segment:offset of ASCIIZ file specification

Returns

If operation is successful:

Carry flag is clear.

If AL was 00H on call:

AL	= status:	
	00H	not installed, OK to install
	01H	not installed, not OK to install
	FFH	installed

If AL was 04H on call:

DX	= error count
DS:SI	= segment:offset of print queue

If operation is not successful:

Carry flag is set.

AX	= error code:	
	01H	function invalid
	02H	file not found
	03H	path not found

(more)

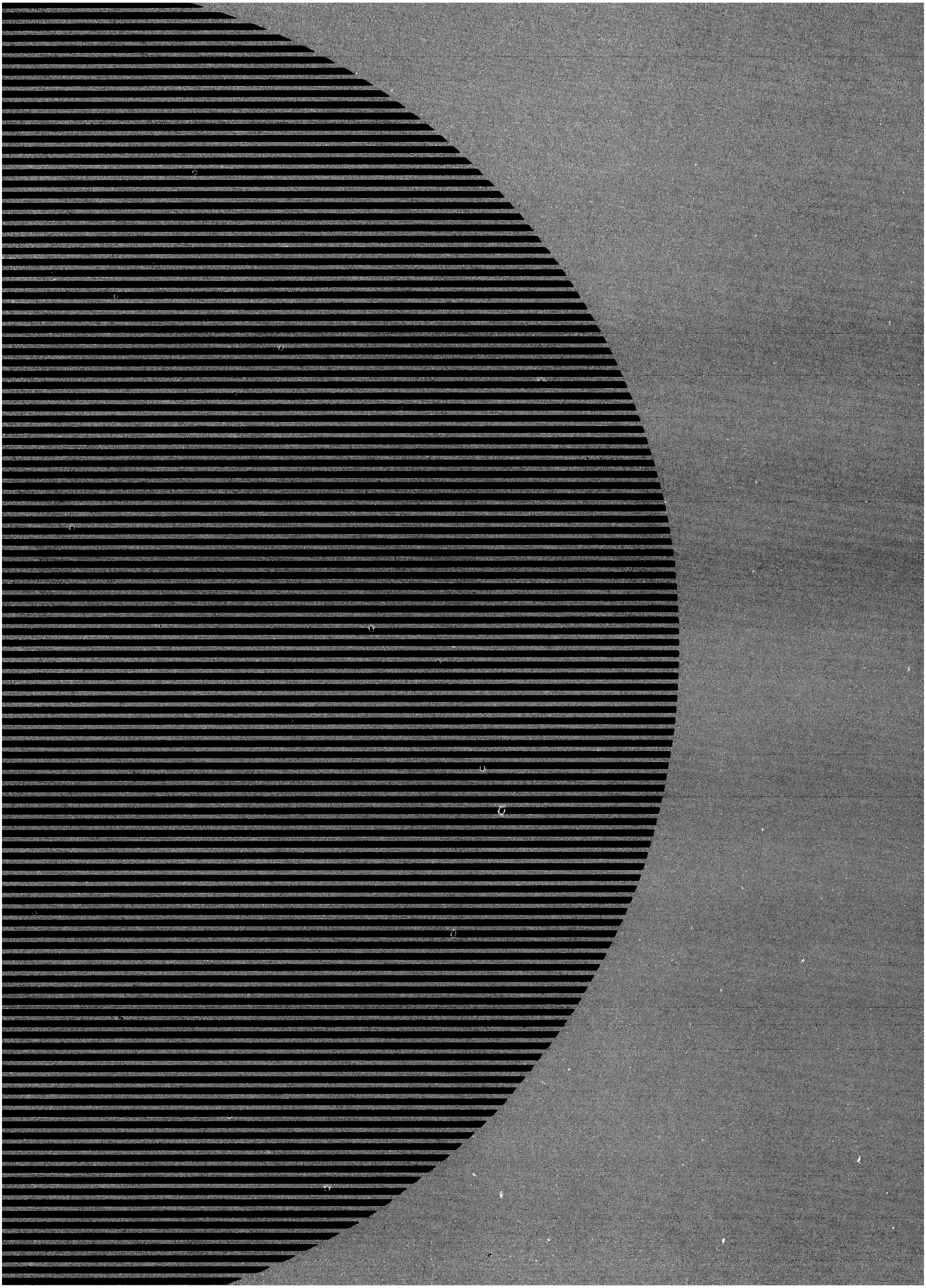
04H	too many open files
05H	access denied
08H	queue full
09H	spooler busy
0CH	name too long
0FH	drive invalid

Programmer's Notes

- For Subfunction 01H, the packet consists of 5 bytes. The first byte contains the level (must be zero), the next 4 bytes contain the doubleword address (segment and offset) of an ASCIIZ file specification. (The filename cannot contain wildcard characters.) If the file exists, it is added to the end of the print queue.
- For Subfunction 02H, wildcard characters (*and ?) are allowed in the file specification, making it possible to delete multiple files from the print queue with one call.
- For Subfunction 04H, the address returned for the print queue points to a series of filename entries. Each entry in the queue is 64 bytes and contains an ASCIIZ file specification. The first file specification in the queue is the one currently being printed. The last slot in the queue has a null (zero) in the first byte.

Example

None



Appendixes

Appendix A
MS-DOS Version 3.3

For the MS-DOS user, version 3.3 incorporates some long-awaited capabilities, runs faster in places, and requires about 9 KB more memory than version 3.2. Its most apparent changes, however, relate to a new, more flexible method of supporting different national languages. For the MS-DOS programmer, version 3.3 offers several enhancements in the areas of file management and internationalization support. This appendix offers an overview of these new features.

Version 3.3 User Considerations

MS-DOS version 3.3 has introduced several changes at the user level. A new external command, FASTOPEN, speeds up the filing system by keeping file locations in memory. A new batch command, CALL, lets a batch file call another batch file and, when that file terminates, continue execution with the next command in the original batch file rather than return to MS-DOS as in previous versions. Two commands previously present only in PC-DOS, COMP and SELECT, have been added to MS-DOS. Five commands have additional capabilities: APPEND, ATTRIB, BACKUP, FDISK, and MODE. In addition, the TIME and DATE commands automatically set the CMOS clock-calendar on the IBM PC/AT and PS/2 machines, making use of the separate SETUP program unnecessary for these functions. Changes to the national language support involve four new commands, three new options to the MODE command, two new or modified system information files, and two new device drivers. Each of these new or modified commands is discussed individually below.

The FASTOPEN command

When MS-DOS searches for a program file, it searches each directory specified in the PATH search path. A lengthy path that has to search many levels of a directory structure can make this a slow process. The FASTOPEN command loads a terminate-and-stay-resident (TSR) program that caches the locations of the most recently accessed directories and files on one or more fixed disks in the system. The number of files and directories to be cached is under the user's control; the default is 10. When it needs a file, MS-DOS looks first in the FASTOPEN list; if the file is found in the list, MS-DOS can bypass inspection of the search path specified by PATH. When the FASTOPEN list is filled and a new file is opened, the new file replaces the least recently used file on the FASTOPEN list.

The improvement in file-system performance depends on the number of open files and the frequency of file access. The FASTOPEN command can be entered only once during a session and, if desired, can be placed in the AUTOEXEC.BAT file.

The FASTOPEN command has two parameters:

FASTOPEN *drive*:[=*entries*][...]

The *drive* parameter is the drive letter, followed by a colon, of a fixed disk for which FASTOPEN is to keep track of the most recently accessed directories and files. More than one drive can be specified by separating the drive identifiers with spaces; the maximum is four drives. A drive associated with a JOIN, SUBST, or ASSIGN command cannot be specified, nor can a drive assigned to a network.

The optional *entries* parameter is the number of directory entries FASTOPEN is to keep in memory. The value of *entries* can be from 10 through 999; the default is 34. If more than one *entries* value is specified, their sum cannot exceed 999. Each entry subtracts 40 bytes from the RAM normally available to run application programs.

Examples: The following command tells MS-DOS to keep track of the last 50 directories and files on drive C:

```
C>FASTOPEN C:=50  <Enter>
```

The next command tells MS-DOS to keep track of the last 34 files on drives C and D:

```
C>FASTOPEN C: D:  <Enter>
```

Changes to batch-file processing

Batch-file processing also gains power in MS-DOS version 3.3. The user can now suppress the echo of all batch commands and call one batch file from another without terminating the first batch file.

@

With MS-DOS version 3.3, any line in a batch file preceded by @ is not echoed to the screen when the batch file is executed.

CALL

A batch file no longer needs to load an additional copy of COMMAND.COM in order to execute another batch file and return control to the calling batch file. The CALL command executes a batch file and returns to the next command in the calling batch file.

CALL commands can be nested. If an exit condition is provided, a batch file can even call itself; however, the input or output of a called batch file cannot be redirected or piped.

The CALL command has two parameters:

CALL *batch-file* [*parameters*]

The *batch-file* parameter is the name of the batch file to be executed. The file must be in the current drive and directory or in a drive and/or directory specified in the command path.

The optional *parameters* parameter represents any parameters that may be required by *batch-file.*

Example: Suppose the batch file SORTFILE.BAT accepts one parameter. The following command calls SORTFILE.BAT, specifying NAMES.TXT as the parameter:

```
CALL SORTFILE NAMES.TXT
```

If NAMES.TXT was specified as a command-line parameter to the *calling* batch file, the CALL command could be

```
CALL SORTFILE %1
```

Commands from PC-DOS

Two commands have been added to MS-DOS from earlier versions of PC-DOS: COMP, present in PC-DOS version 1.0, and SELECT, present in PC-DOS version 2.0.

COMP

The COMP command compares two files or sets of files and reports any differences encountered. FC, a similar file-comparison command present in MS-DOS versions 2.0 and later, is still included with MS-DOS 3.3. *See* USER COMMANDS: COMP; FC.

Syntax for the COMP command is

COMP [*drive*:][*filename1*] [*drive*:][*filename2*]

The optional *drive* parameter is the drive letter, followed by a colon, of the drive containing the file to be compared. The *filename1* parameter is the name and location of the file to compare to *filename2*; *filename2* is the name and location of the file to be compared against. Both filenames can be preceded by a path; wildcard characters are permitted in either filename.

Example: The following command tells MS-DOS to compare the file NEWFILE.TXT in the current drive and directory to the file OLDFILE.TXT in the \ARCHIVE directory on drive D and report any differences encountered:

```
C>COMP NEWFILE.TXT D:\ARCHIVE\OLDFILE.TXT  <Enter>
```

SELECT

The SELECT command creates a system disk with the time format, date format, and keyboard layout configured for a selected country. The syntax for SELECT is

SELECT [[*drive1*:] [*drive2*:][*path*]] [*country*][*keyboard*]

The optional *drive1* parameter is the drive containing a disk with the MS-DOS operating-system files, the FORMAT program, and the country configuration files. The *drive2* parameter is the drive containing the disk to be formatted with the country-specific information; this drive specifier can be followed by a path. The *country* parameter is a code that selects the date and time format; the information is taken from the COUNTRY.SYS system file. The *keyboard* parameter is a code that selects the desired keyboard layout. *See* KEYB below.

The SELECT command

- Formats the target disk.
- Creates CONFIG.SYS and AUTOEXEC.BAT files on the target disk.
- Copies the contents of the source disk to the destination disk.

Example: The following command, which assumes drive A contains a valid system disk and drive B contains the disk to be formatted, creates a bootable system disk that includes country-specific information and keyboard layout for Germany:

```
C>SELECT A: B: 049 GR  <Enter>
```

Enhanced commands

Several existing MS-DOS user commands have been given expanded capabilities in version 3.3. These are presented alphabetically in the next few pages. *See* USER COMMANDS: APPEND; ATTRIB; BACKUP; FDISK; MODE.

APPEND

The APPEND command specifies a search path for data files — files whose extensions are neither .COM, .EXE, nor .BAT — similar to the command path specified by the PATH command, which searches only for executable files *with* those extensions. APPEND has three forms, depending on whether it is being entered for the first time. When it is entered the first time, the APPEND command now has two optional switches:

APPEND [/E] [/X]

The /E switch makes the data path part of the environment, like the command path. The data path can then be displayed or changed with both the SET and APPEND commands and is inherited by child processes. (However, any changes made to the data path by the child process are lost when the child returns to its parent process.)

The /X switch causes calls to the Find First File functions (Interrupt 21H Functions 11H and 4EH) and the EXEC function (Interrupt 21H Function 4BH) to search the data path. If /X is not specified, only Interrupt 21H Function 0FH (Open File with FCB), Interrupt 21H Function 23H (Get File Size), and Interrupt 21H Function 3DH (Open File with Handle) system calls search the data path.

If either /X or /E is specified the first time APPEND is entered, a pathname cannot be included.

Subsequent uses of the command must take the form

APPEND [[*drive*:]*path*] [;[*drive*:]*path* ...]

or

APPEND ;

The *path* parameter is the name of a directory that is to be made part of the data path. The user can specify as many directory names as will fit in the 128 characters of the command line. Entries must be separated by semicolons. If APPEND is followed only by a semicolon, any previous APPEND paths are deleted.

Example: The following two APPEND commands make the data path part of the environment and put the directories C:\WORD\PROPOSAL, C:\WORD\REPORTS, and C:\123\BUDGET in the data path:

```
C>APPEND /E  <Enter>
C>APPEND C:\WORD\PROPOSAL;C:\WORD\REPORTS;C:\123\BUDGET  <Enter>
```

Because the data path usually involves frequently used directories, the APPEND command ordinarily is placed in the AUTOEXEC.BAT file.

Note: APPEND is a new command in PC-DOS version 3.3.

ATTRIB

The /S switch has been added to the ATTRIB command so that any attribute changes can be applied to all files in subdirectories contained in the specified directory.

Example: The following command sets the read-only attribute of all files in the directory C:\DOS and in all its subdirectories:

```
C>ATTRIB +R C:/DOS /S  <Enter>
```

BACKUP

A formatting parameter has been added to the BACKUP command in MS-DOS version 3.3. The /F switch tells MS-DOS to format the backup diskette if it hasn't been formatted. The /F switch formats the backup diskette to the maximum capacity of the backup drive, so a disk of lower capacity, such as a 360 KB diskette in a 1.2M drive, should not be used. If this switch is used, FORMAT.COM must be available in the current drive and directory or in one of the directories named in the environment's PATH string.

Performance of the BACKUP command has also been improved. Instead of storing each file separately on the backup disk, BACKUP stores only two files: BACKUP.*nnn*, which contains all the backed-up files, and CONTROL.*nnn*, which contains the pathnames of the backed-up files.

FDISK

FDISK can now create a new type of MS-DOS partition called an extended partition on a fixed disk. An extended partition can contain multiple logical drives and allows the use of very large fixed disks. Each logical drive is still limited to 32 MB.

An extended partition is not bootable. In order for the fixed disk to be bootable, it must also contain a primary MS-DOS partition that has been formatted using the FORMAT command with the /S switch so that it contains a system boot record and the operating-system files.

MODE

The MODE command now supports two additional serial ports (COM3 and COM4) and increases the maximum serial transmission rate to 19,200 baud.

Some additional options have been added to MODE to support code-page switching. *See* MODE Command Changes below.

New national language support

The new national language support in MS-DOS version 3.3 replaces the methods used in previous versions to change the keyboard layout and the display and printer character sets so that more than one language could be used. These changes are extensive: four new or modified system files, three new commands, four new options for the MODE command, a new parameter for the GRAFTABL command, and a new parameter for the COUNTRY and DEVICE configuration commands.

Code pages and code-page switching

The key element of the new national language support is the code page, a table of 256 character correspondence codes. MS-DOS recognizes both a hardware code page, which is the character correspondence table built into a device, and a prepared code page, which is an alternate character correspondence table available through MS-DOS. The current code page is the code page most recently selected.

The hardware code page for a device is determined by the country for which the device was manufactured. The user selects a prepared code page, from a list of five included with MS-DOS version 3.3, by using the new CP PREPARE option of the MODE command. *See* MODE Command Changes below.

The new national language support is often referred to as code-page switching because, after the devices and code pages required by the system have been defined, the only commands the user must deal with simply switch from one code page to another. In order to use the new national language support, device drivers must support code-page switching and the devices must be able to display the full character sets.

Code pages are numbered. The identifying numbers have no relationship to the country code introduced with previous versions of MS-DOS and used by the COUNTRY configuration command. Five code pages are included with version 3.3:

Page Number	Configuration
437	United States
850	Multilingual
860	Portugal
863	Canadian French
865	Norway/Denmark

Code page 437 is the character correspondence table used in previous versions of MS-DOS. Its character set supports United States English and includes many accented characters used in other languages. It is the hardware code page for most countries.

Code page 850 replaces two of the four box-drawing sets and some of the mathematical symbols in code page 437 with additional accented characters. It supports English and most Latin-based European languages.

Code page 860 is for Portuguese, code page 863 is for Canadian French, and code page 865 is for Norwegian/Danish. These pages are the hardware code pages for the specified countries.

Setting up the system for code-page switching

Although several commands are required to manage national language support, the process is fairly straightforward. Setting up the system requires the following:

- A DEVICE configuration command in CONFIG.SYS to load a driver for each device that supports code-page switching.
- An NLSFUNC command in AUTOEXEC.BAT to load the memory-resident national language support functions.
- A MODE CP PREPARE command in AUTOEXEC.BAT to prepare code pages for each device that supports code-page switching.
- A CHCP command in AUTOEXEC.BAT to select the initial code page.
- Optionally, a KEYB command in AUTOEXEC.BAT to select the initial keyboard layout.

After starting the system with these commands in CONFIG.SYS and AUTOEXEC.BAT, only a MODE CP SELECT command is required to change to a different language during an MS-DOS session.

The COUNTRY configuration command is still used to control country-specific characteristics such as the time and date format and currency symbol. An added parameter in the COUNTRY command lets the user also specify a code page. *See* Modified National Language Support Commands below.

The system files

MS-DOS version 3.3 includes four system files that support the national language functions: two device drivers and two system information files.

The device drivers are PRINTER.SYS and DISPLAY.SYS. These drivers implement code-page switching for the IBM Proprinter Model 4201 and Quietwriter III Model 5202 printers and for the EGA, PC Convertible LCD, and PS/2 display adapters. They also support all display adapters compatible with the EGA.

The information files are COUNTRY.SYS, which contains information such as time and date formats and currency symbols, and KEYBOARD.SYS, which contains the scan-code-to-ASCII translation tables for the various keyboard layouts.

The new support commands

The new national language support in MS-DOS version 3.3 adds three MS-DOS commands: Change Code Page (CHCP), Keyboard (KEYB), and National Language Support Functions (NLSFUNC).

CHCP

The Change Code Page (CHCP) command tells MS-DOS which code page to use for all devices that support code-page switching.

The NLSFUNC command must be executed before the CHCP command can be used.

CHCP is a system-wide command: It specifies the code page used by MS-DOS and each device attached to the system that supports code-page switching. The CP SELECT option of the MODE command, on the other hand, specifies the code page for a single device.

If the code page specified with CHCP is not compatible with a device, CHCP responds

```
Code page nnn not prepared for all devices
```

If the code page specified with CHCP was not first identified with the CP PREPARE option of the MODE command, CHCP responds

```
Code page nnn not prepared for system
```

The CHCP command has one optional parameter:

CHCP [*code-page*]

The *code-page* parameter is the three-digit number that specifies the code page MS-DOS is to use. If *code-page* is omitted, CHCP displays the current MS-DOS code page.

Examples: The following command changes the system code page to 850:

```
C>CHCP 850  <Enter>
```

If the current code page is 850 and CHCP is entered without parameters, MS-DOS responds:

```
Active code page: 850
```

KEYB

The Keyboard (KEYB) command selects a keyboard layout by changing the scan-code-to-ASCII translation table used by the keyboard driver. It replaces the KEYBxx commands used in earlier versions of MS-DOS to select keyboard layouts.

The first time KEYB is executed, it loads the memory-resident keyboard driver and the translation table, thereby increasing the size of MS-DOS by slightly more than 7 KB. Subsequent executions simply load a different translation table, which replaces the previously loaded translation table and accommodates a different country-specific keyboard layout.

The KEYB command has three optional parameters:

KEYB [*country*[,[*code-page*],*kbdfile*]]

The *country* parameter is one of the following two-character country codes:

Country	Code	Country	Code
Australia	US	Netherlands	NL
Belgium	BE	Norway	NO
Canada		Portugal	PO
English	US	Spain	SP
French	CF	Sweden	SV
Denmark	DK	Switzerland	
Finland	SU	French	SF
France	FR	German	SG
Germany	GR	United Kingdom	UK
Italy	IT	United States	US
Latin America	LA		

The *code-page* parameter is the three-digit number that specifies the code page defining the character set that MS-DOS is to use.

If the specified country code and code page aren't compatible, KEYB responds:

```
Code page requested nnn is not valid for given keyboard code
```

If KEYB is entered with no parameters, MS-DOS displays the currently active keyboard country code, keyboard code page, and console device code page.

Examples: The following command selects the French keyboard layout, code page 850, and the keyboard definition file named C:\DOS\KEYBOARD.SYS:

```
C>KEYB FR,850,C:\DOS\KEYBOARD.SYS  <Enter>
```

If the code page is omitted but the keyboard definition file is specified, the comma must be included to show the missing parameter:

```
C>KEYB FR,,C:\DOS\KEYBOARD.SYS  <Enter>
```

NLSFUNC

The National Language Support Function (NLSFUNC) command loads a memory-resident program that implements code-page switching. It also allows the user to name the file that contains country-specific information — such as date format, time format, and currency symbol — if there is no COUNTRY configuration command in CONFIG.SYS. NLSFUNC must be used before the Change Code Page (CHCP) command.

If national language support is needed for every session, NLSFUNC should be placed in the AUTOEXEC.BAT file.

The NLSFUNC command has one optional parameter:

NLSFUNC [*country-file*]

The *country-file* parameter is the name of the country information file (in most implementations of MS-DOS, COUNTRY.SYS). If *country-file* is omitted, MS-DOS defaults to the name of the country information file specified in the COUNTRY configuration command in CONFIG.SYS; if there is no COUNTRY configuration command in CONFIG.SYS, MS-DOS looks for a file named COUNTRY.SYS in the root directory of the current drive.

Example: The following command loads the NLSFUNC program and specifies C:\DOS\COUNTRY.SYS as the country information file:

```
C>NLSFUNC C:\DOS\COUNTRY.SYS  <Enter>
```

The modified support commands

The new national language support changes two configuration commands — COUNTRY and DEVICE — and two general MS-DOS commands — GRAFTABL and MODE.

COUNTRY

The COUNTRY configuration command now has three parameters:

COUNTRY=*country-code*,[*code-page*],[*country-file*]

The *country-code* parameter is one of the following three-digit country codes (identical to the specified country's international telephone prefix):

Country	Code	Country	Code
Arabia	785	Latin America	003
Australia	061	Netherlands	031
Belgium	032	Norway	047
Canada		Portugal	351
English	001	Spain	034
French	002	Sweden	046
Denmark	045	Switzerland	
Finland	358	French	041
France	033	German	041
Germany	049	United Kingdom	044
Israel	972	United States	001
Italy	039		

The *code-page* parameter is the three-digit number that specifies the code page defining the character set that MS-DOS is to use.

The *country-file* parameter is the name of the file that contains the country-specific information; the name of the file can be preceded by a drive and/or path. If *country-file* is omitted, MS-DOS defaults to the file COUNTRY.SYS, which it looks for in the root directory of the current drive.

The COUNTRY command is not required; if it is not included in CONFIG.SYS, MS-DOS defaults to country 001 (US), code page 437, and country information file COUNTRY.SYS in the root directory of the current drive.

Example: The following CONFIG.SYS command specifies the French country code, code page 850, and C:\DOS\COUNTRY.SYS as the country information file:

```
COUNTRY=033,850,C:\DOS\COUNTRY.SYS
```

DEVICE

Two options have been added to the DEVICE configuration command that allow the user to specify the display and printer drivers that support code-page switching.

The display driver that supports code-page switching is DISPLAY.SYS. It supports the IBM Enhanced Graphics Adapter (EGA), the IBM Personal System/2 display adapter, and all display adapters compatible with either of these. The Monochrome Display Adapter (MDA) and the Color/Graphics Adapter (CGA) do not support code-page switching.

If the ANSI.SYS display driver is also used, the DEVICE command that defines it must precede the DEVICE command that defines DISPLAY.SYS.

When used to specify the display driver, the DEVICE command has five parameters:

DEVICE=*driver* CON=(*type*[,[*hwcp*][,*prepcp*[,*sub-fonts*]]])

The *driver* parameter is the name of the file that contains the display driver; the filename can be preceded by a drive and/or path. If *driver* is omitted, MS-DOS defaults to the file DISPLAY.SYS, which it looks for in the root directory of the current drive.

The *type* parameter defines the type of display adapter attached to the system. It must be one of the following:

Code	Adapter
MONO	Monochrome display/printer adapter
CGA	Color/graphics adapter
EGA	Enhanced graphics adapter or IBM Personal System/2 display adapter
LCD	IBM PC Convertible liquid crystal display

The *hwcp* parameter is the three-digit number that specifies the hardware code page supported by the display adapter:

Code	Configuration
437	United States (default)
850	Multilingual
860	Portugal
863	Canadian French
865	Norway/Denmark

The *prepcp* parameter is the number of additional code pages the display can support. These are referred to as prepared code pages and must be defined by the CP PREPARE option of the MODE command. If *type* is either MONO or CGA, *prepcp* must be 0; the default is 0. If *type* is either EGA or LCD, *prepcp* can be any value from 1 through 12; the default is 1. If *hwcp* is 437, *prepcp* should be allowed to default to 1; if *hwcp* is not 437, *prepcp* should be set to 2.

The *sub-fonts* parameter is the number of subfonts supported for each code page. If *type* is either MONO or CGA, *sub-fonts* must be 0; the default is 0. If *type* is EGA, *sub-fonts* can be 1 or 2; the default is 2. If *type* is LCD, *sub-fonts* can be 1 or 2; the default is 1.

Example: The following CONFIG.SYS command specifies C:\DOS\DISPLAY.SYS as the display driver for an EGA whose hardware code page is 437. The parameter for prepared code pages is allowed to default to 1 and the parameter for subfonts is allowed to default to 2.

```
DEVICE=C:\DOS\DISPLAY.SYS CON=(EGA,437)
```

The printer driver that supports code-page switching is PRINTER.SYS. It supports the IBM Proprinter Model 4201, the IBM Quietwriter III Printer Model 5202, and all printers compatible with either of these.

When used to specify the printer driver, the DEVICE configuration command has five parameters:

DEVICE=*driver port*=(*type*[,[*hwcp*][,*prepcp*]])

The *driver* parameter is the name of the file that contains the printer driver; the filename can be preceded by a drive and/or path. If *driver* is omitted, MS-DOS defaults to the file PRINTER.SYS, which it looks for in the root directory of the current drive.

The *port* parameter is the MS-DOS device name of the printer port being defined: LPT1 (or PRN), LPT2, or LPT3. A different set of *type*, *hwcp*, and *prepcp* parameters can be specified for each of the three printer ports.

The *type* parameter defines the type of printer attached to the printer port. It must be one of the following:

Code	Printer
4201	IBM Proprinter Model 4201
5202	IBM Quietwriter III Printer Model 5202

The *hwcp* parameter is a three-digit number that specifies the hardware code page supported by the hardware:

Code	Configuration
437	United States (default)
850	Multilingual
860	Portugal
863	Canadian French
865	Norway/Denmark

If *type* is 5202, two hardware code-page numbers can be specified, enclosed in parentheses and separated by a comma. If two hardware code pages are specified, *prepcp* must be 0.

The *prepcp* parameter is the number of additional code pages (referred to as prepared code pages) for which MS-DOS must reserve buffer space; its value can be from 0 through 12. These additional code pages must be defined by the CP PREPARE option of the MODE command. If *hwcp* is 437, *prepcp* should be set to 1; if *hwcp* is not 437 and only one *hwcp* value is specified, *prepcp* should be set to 2.

Examples: The following CONFIG.SYS command defines C:\DOS\PRINTER.SYS as the printer driver for the PRN device. The printer is an IBM Proprinter Model 4201 whose hardware code page is 437, and MS-DOS is instructed to allow for one prepared code page:

```
DEVICE=C:\DOS\PRINTER.SYS PRN=(4201,437,1)
```

The next CONFIG.SYS command defines C:\DOS\PRINTER.SYS as the printer driver for ports LPT1 and LPT2. The printer attached to LPT1 is the same as in the previous command; the printer attached to LPT2 is an IBM Quietwriter III Printer Model 5202 with two hardware code pages (437 and 850). For the second printer, MS-DOS is instructed to allow for no prepared code pages.

```
DEVICE=C:\DOS\PRINTER.SYS LPT1=(4201,437,1) LPT2=(5202,(437,850),0)
```

GRAFTABL

The GRAFTABL command now has two forms:

GRAFTABL [*code-page*]

or

GRAFTABL /STATUS

The first form of the command loads a code page for the color/graphics adapter (CGA) so that its character set matches that used by MS-DOS and other devices when displaying the upper 128 characters. The *code-page* parameter is the three-digit number that specifies the code page defining the character set that GRAFTABL is to use.

The /STATUS switch causes GRAFTABL to display the name of the graphics character set table currently in use.

MODE

National language support adds four options to the MODE command:

Option	Action
CODEPAGE	Displays the code pages available and active.
CODEPAGE PREPARE	Defines the code pages selected for use.
CODEPAGE REFRESH	Restores code-page contents damaged by hardware error or other causes.
CODEPAGE SELECT	Selects a code page for a particular device.

(CODEPAGE can be abbreviated to CP in the command line.)

When used to display the status of the code pages, the MODE command has one parameter:

MODE *device* CP

The *device* parameter is the name of the device whose code-page status is to be displayed. It can be CON, PRN, LPT1, LPT2, or LPT3.

Example: The following command displays the status of the console device:

```
C>MODE CON CP  <Enter>
```

When used to define the code page or pages to be used with a device, the MODE command has three parameters:

MODE *device* CP PREPARE=(*code-page font-file*)

The *device* parameter is the name of the device for which the code page or pages are to be prepared. It can be CON, PRN, LPT1, LPT2, or LPT3.

The *code-page* parameter is one or more of the three-digit numbers, enclosed in parentheses, that specify the code page to be used with *device.* If more than one code-page number is specified, the numbers must be separated with spaces.

The *font-file* parameter is the name of the code-page file that contains the font information for *device.* The files provided for IBM devices include

File	Device
EGA.CPI	IBM Enhanced Graphics Adapter (EGA) and EGA-compatible display adapters
4201.CPI	IBM Proprinter Model 4201
5202.CPI	IBM Quietwriter III Printer Model 5202
LCD.CPI	IBM Convertible liquid crystal display

Example: Assume the display is attached to an EGA. The following command prepares code pages 437 and 850 for the console, specifying C:\DOS\EGA.CPI as the code-page information file:

```
C>MODE CON CP PREPARE=((437 850) C:\DOS\EGA.CPI)  <Enter>
```

When used to select a code page for a device, the MODE command has two parameters:

MODE *device* CP SELECT=*code-page*

The *device* parameter is the name of the device for which the code page is to be selected. Permissible values are CON, PRN, LPT1, LPT2, and LPT3.

The *code-page* parameter is the three-digit number that specifies the code page to be used with *device.*

Example: The following command selects code page 850 for the console:

```
C>MODE CON CP SELECT=850  <Enter>
```

Setting up code-page switching for an EGA-only system

Figure A-1 shows the commands required to implement the new national language support for a system that includes only a display attached to an EGA or EGA-compatible adapter. The hardware code page of the EGA is 437 (United States English) and the system is set up to handle code pages 437 and 850. All MS-DOS files are assumed to be in the directory \DOS on the disk in drive C. If the ANSI.SYS driver is not used, the configuration command DEVICE=C:\DOS\ANSI.SYS should be omitted from CONFIG.SYS; if ANSI.SYS is used, however, the DEVICE configuration command that defines it must precede the DEVICE configuration command that defines DISPLAY.SYS.

Commands in CONFIG.SYS:

```
COUNTRY=001,437,C:\DOS\COUNTRY.SYS
DEVICE=C:\DOS\ANSI.SYS
DEVICE=C:\DISPLAY.SYS CON=(EGA,437,1)
```

Commands in AUTOEXEC.BAT:

```
NLSFUNC C:\DOS\COUNTRY.SYS
MODE CON CP PREPARE=((437 850) C:\DOS\EGA.CPI)
MODE CON CP SELECT=437
KEYB US,437,C:\DOS\KEYBOARD.SYS
```

Figure A-1. Setup commands for a system with an EGA only.

When the system is started, code page 437 is selected for MS-DOS, the display, and the keyboard. To change to code page 850 during the session, simply type

```
C>CHCP 850  <Enter>
```

Setting up code-page switching for a PS/2 and printer

Figure A-2 shows the commands required to implement the new national language support for an IBM Personal System/2 or compatible system that includes both a PS/2, EGA, or EGA-compatible display adapter and an IBM Proprinter Model 4201. The hardware code page of both devices is 437 (United States English) and the system is set up to handle code pages 437 and 850.

Commands in CONFIG.SYS:

```
COUNTRY=001,437,C:\DOS\COUNTRY.SYS
DEVICE=C:\DOS\ANSI.SYS
DEVICE=C:\DISPLAY.SYS CON=(EGA,437,1)
DEVICE=C:\DOS\PRINTER.SYS PRN=(4201,437,1)
```

Commands in AUTOEXEC.BAT:

```
NLSFUNC C:\DOS\COUNTRY.SYS
MODE CON CP PREPARE=((437 850) C:\DOS\EGA.CPI)
MODE PRN CP PREPARE=((437 850) C:\DOS\4202.CPI)
MODE CON CP SELECT=850
MODE PRN CP SELECT=850
KEYB US,850,C:\DOS\KEYBOARD.SYS
```

Figure A-2. Setup commands for a PS/2 with display and printer.

Again, all MS-DOS files are assumed to be in the directory \DOS on the disk in drive C. If the ANSI.SYS driver is not used, the configuration command DEVICE=C:\DOS\ANSI.SYS should be omitted from CONFIG.SYS; if ANSI.SYS is used, however, the DEVICE configuration command that defines it must precede the DEVICE configuration command that defines DISPLAY.SYS.

Version 3.3 Programming Considerations

The changes introduced in MS-DOS version 3.3 that are of primary interest to the programmer include

- New Interrupt 21H function calls for file management and internationalization support
- An extension to the definition of the MS-DOS IOCTL function for code-page switching, plus the addition of the underlying device-driver support
- Support for extended MS-DOS partitions on fixed disks

Each of these areas is discussed in detail below.

New file-management functions

MS-DOS version 3.3 includes two new Interrupt 21H file-management functions: Set Handle Count (Function 67H) and Commit File (Function 68H).

Set Handle Count

The Set Handle Count function (Interrupt 21H Function 67H) allows a single process to have more than 20 handles for files or devices open simultaneously. Function 67H is invoked by issuing a software Interrupt 21H with

AH = 67H
BX = number of desired handles

On return,

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code

For each process, the operating system maintains a table that relates handle numbers for the process to MS-DOS's internal global table for all open files in the system. In MS-DOS versions 3.0 and later, the per-process table is ordinarily stored within the reserved area of the program segment prefix (PSP) and has only enough room for 20 handle entries. If 20 or fewer handles are requested in register BX, Function 67H takes no action and returns a success signal. If more than 20 handles are requested, however, Function 67H allocates on behalf of the calling program a new block of memory that is large enough to hold the expanded table of handle numbers and then copies the process's old handle table to the new table. Because the function will fail if the system does not have sufficient free memory to allocate the new block, most programs need to make a call to Interrupt 21H Function 4AH (Resize Memory Block) to "shrink" their initial memory block allocations before calling Function 67H.

Function 67H does not fail if the number requested is larger than the available entries in the system's global table for file and device handles. However, a subsequent attempt to open a file or device or to create a new file will fail if all the entries in the system's global file table are in use, even if the requesting process has not used up all its own handles. (The size of the global table is controlled by the FILES entry in the CONFIG.SYS file. *See* USER COMMANDS: CONFIG.SYS: FILES; PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.)

Example: Set the maximum handle count for the current process to 30, so that the process can have as many as 25 files or devices open simultaneously (5 of the handles are already expended by the MS-DOS standard devices when the process starts up). Note that a FILES=30 (or greater value) entry in the CONFIG.SYS file also is required for the process to successfully open 30 files or devices.

```
        .
        .
        .
        mov     ah,67h          ; Function 67H = set handle count.
        mov     bx,30           ; Maximum number of handles.
        int     21h             ; Transfer to MS-DOS.
        jc      error           ; Jump if function failed.
        .
        .
        .
```

Commit File

The Commit File function (Interrupt 21H Function 68H) forces all data in MS-DOS's internal buffers that is associated with a given handle to be written to disk and forces the corresponding disk directory and file allocation table (FAT) information to be updated. By calling this function at appropriate points within its execution, a program can ensure that newly entered data will not be lost if there is a power failure, if the program crashes, or if the user fails to terminate the program properly before turning off the machine. Function 68H is called by issuing a software Interrupt 21H with

AH = 68H
BX = handle for previously opened file.

On return,

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code

The effect of Function 68H is equivalent to closing and reopening the file or to duplicating a file handle with Interrupt 21H Function 45H (Duplicate File Handle) and then closing the duplicate. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management. However, Function 68H has the advantages that the application will not lose control of the file (as could happen with the close-open sequence in a networking environment) and that it will not fail because of a lack of handles (as the duplicate handle method might).

Note: Function 68H operations requested on a handle associated with a character device return a success flag but have no effect.

Example: Assume that the file MYFILE.DAT has been opened previously and that the handle for the file is stored in the variable *fhandle*. Call Function 68H to ensure that any data in MS-DOS's internal buffers associated with the handle is written out to disk and that the directory and FAT are up-to-date.

```
fname   db      'MYFILE.DAT',0  ; ASCIIZ filename.
fhandle dw      ?               ; Handle from Open operation.
        .
        .
        .
        mov     ah,68h          ; Function 68H = commit file.
        mov     bx,fhandle      ; Handle from previous open.
        int     21h             ; Transfer to MS-DOS.
        jc      error           ; Jump if function failed.
        .
        .
        .
```

New internationalization support functions

MS-DOS version 3.3 includes two new Interrupt 21H internationalization support functions: Get Extended Country Information (Function 65H) and Select Code Page (Function 66H).

Get Extended Country Information

The Get Extended Country Information function (Interrupt 21H Function 65H) returns a superset of the internationalization information obtained with Interrupt 21H Function 38H (Get/Set Current Country). Function 65H is called by issuing a software Interrupt 21H with

AH	= 65H	
AL	= information ID code:	
	01H	get general internationalization information
	02H	get pointer to uppercase table
	04H	get pointer to filename uppercase table
	06H	get pointer to collating sequence table
BX	= code page of interest (active CON device = –1)	
CX	= length of buffer to receive information (error returned if less than 5)	
DX	= country ID (default = –1)	
ES:DI	= address of buffer to receive information	

On return,

If function is successful:

Carry flag is clear.

Requested data is in calling program's buffer.

If function is not successful:

Carry flag is set.

AX = error code

Function 65H may fail if either the country code or the code-page number is invalid or if the code page does not match the country code. If the buffer to receive the information is at least 5 bytes but is too short for the requested information, the data is truncated and no error is returned.

The format of the data returned by Subfunction 01H in the calling program's buffer is

Field	Size
Information ID code (01H)	Byte
Length of following buffer (38 or less)	Word
Country ID	Word
Code-page number	Word
Date format	Word
Currency symbol	5 bytes
Thousands separator	Word
Decimal separator	Word
Date separator	Word
Time separator	Word
Currency format flags	Byte
Digits in currency	Byte
Time format	Byte
Monocase routine entry point	Doubleword
Data list separator	Word
Reserved	10 bytes

See SYSTEM CALLS: INTERRUPT 21H: Function 38H.

The format of the data returned by Subfunctions 02H, 04H, and 06H is

Field	Size
Information ID code (02H, 04H, or 06H)	Byte
Pointer to table	Doubleword

The uppercase and filename uppercase tables are 130 bytes. The first 2 bytes contain the size of the table; the subsequent 128 bytes contain the uppercase equivalents, if any, for character codes 80H through 0FFH. The main use of these tables is to map accented or otherwise modified vowels to their plain vowel equivalents. Text translated using these tables can be sent to devices that do not support the IBM graphics character set or can be used to create filenames that do not require a special keyboard configuration for entry.

The collating table is 258 bytes. The first 2 bytes contain the table length and the next 256 bytes contain the values to be used for the corresponding character codes (0–0FFH) during a sort operation. Among other things, this table maps uppercase and lowercase ASCII characters to the same collating codes (so that sorts will be case insensitive) and maps accented vowels to their plain vowel equivalents.

Note: In some cases, a truncated translation table might be presented to the program by MS-DOS. Applications should always check the length specified at the beginning of the table to be sure the table contains a translation code for the character of interest.

Example: Obtain the extended country information associated with the default country and code page 437.

```
buffer  db      41 dup (0)      ; Receives country information.
        .
        .
        .
        mov     ax,6501h        ; Function = get extended info.
        mov     bx,437          ; Code page.
        mov     cx,41           ; Length of buffer.
        mov     dx,-1           ; Default country.
        mov     di,seg buffer   ; ES:DI = buffer address.
        mov     es,di
        mov     di,offset buffer
        int     21h             ; Transfer to MS-DOS.
        jc      error           ; Jump if function failed.
        .
        .
        .
```

In this case, MS-DOS fills the following extended country information into the buffer:

```
buffer  db      1               ; Information ID code
        dw      38              ; Length of following buffer
        dw      1               ; Country ID (USA)
        dw      437             ; Code-page number
        dw      0               ; Date format
        db      '$',0,0,0,0     ; Currency symbol
        db      ',',0           ; Thousands separator
        db      '.',0           ; Decimal separator
        db      '-',0           ; Date separator
        db      ':',0           ; Time separator
        db      0               ; Currency format flags
        db      2               ; Digits in currency
        db      0               ; Time format
        dd      026ah:176ch     ; Monocase routine entry point
        db      ',',0           ; Data list separator
        db      10 dup (0)      ; Reserved
```

Example: Obtain the pointer to the uppercase table associated with the default country and code page 437.

```
buffer  db      5 dup (0)       ; Receives pointer information.
        .
        .
        .
        mov     ax,6502h        ; Function = get pointer to
                                ; uppercase table.
```

(more)

```
          mov     bx,437          ; Code page.
          mov     cx,5            ; Length of buffer.
          mov     dx,-1           ; Default country.
          mov     di,seg buffer   ; ES:DI = buffer address.
          mov     es,di
          mov     di,offset buffer
          int     21h             ; Transfer to MS-DOS.
          jc      error           ; Jump if function failed.
          .
          .
          .
```

In this case, MS-DOS fills the following values into the buffer:

```
buffer  db      2               ; Information ID code
        dw      0204h           ; Offset of uppercase table
        dw      1140h           ; Segment of uppercase table
```

The table at 1140:0204H contains the following data:

```
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
1140:0200              80 00 80 9A 45 41 8E 41 8F 80 45 45      ....EA.A..EE
1140:0210  45 49 49 49 8E 8F 90 92 92 4F 99 4F 55 55 59 99  EIII.....O.OUUY.
1140:0220  9A 9B 9C 9D 9E 9F 41 49 4F 55 A5 A5 A6 A7 A8 A9  ......AIOU......
1140:0230  AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9  ................
1140:0240  BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9  ................
1140:0250  CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9  ................
1140:0260  DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9  ................
1140:0270  EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9  ................
1140:0280  FA FB FC FD FE FF                                ......
```

Select Code Page

The Select Code Page function (Interrupt 21H Function 66H) queries or selects the current code page. Function 66H is called by issuing a software Interrupt 21H with

AH = 66H
AL = subfunction:
    01H get code page
    02H select code page
BX = code page to select if AL = 02H

On return,

If function is successful:

Carry flag is clear.

If AL was 01H on call:

BX = active code page
DX = default code page

If function is not successful:

Carry flag is set.

AX = error code

When Subfunction 02H is used, MS-DOS gets the new code page from the COUNTRY.SYS file. The device must be previously prepared for code-page switching by including the appropriate DEVICE command in the CONFIG.SYS file and by issuing the NLSFUNC and MODE CP PREPARE commands (usually by placing them in the AUTOEXEC.BAT file).

Example: Force the active code page to be the same as the system's default code page—that is, return to the code page that was active when the system was first booted.

```
.
.
.
mov     ax,6601h        ; Function = get code page.
int     21h             ; Transfer to MS-DOS.
jc      error           ; Jump if function failed.

mov     bx,dx           ; Force active page = default.

mov     ax,6602h        ; Function = set code page.
int     21h             ; Transfer to MS-DOS.
jc      error           ; Jump if function failed.
.
.
.
```

Extension of IOCTL

The MS-DOS IOCTL service (Interrupt 21H Function 44H) and its device-driver underpinnings have been extended to support code-page switching by the interactive CHCP and MODE commands or by application programs. The relevant IOCTL subfunction is 0CH (Generic IOCTL for Handles). An MS-DOS utility or application program gains access to this subfunction by executing a software Interrupt 21H with

AH = 44H
AL = 0CH
BX = handle for character device
CH = category code:

00H	unknown
01H	COM1, COM2, COM3, or COM4
03H	CON (keyboard and video display)
05H	LPT1, LPT2, or LPT3

(more)

CL	= function (minor) code:	
	4AH	select code page
	4CH	start code-page preparation
	4DH	end code-page preparation
	6AH	query selected code page
	6BH	query prepare list
DS:DX	= pointer to Generic IOCTL parameter block	

On return,

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX	= error code:	
	01H	invalid function number
	19H	bad data read from font file
	22H	unknown command
	26H	code page not prepared or selected
	27H	code page conflict or device or code page not found in file
	29H	device error
	31H	file contents not a valid font or no previous "start code-page preparation" call

Additional information about the cause of the error can be obtained with a call to Interrupt 21H Function 59H (Get Extended Error Information).

The parameter blocks for minor codes 4AH, 4DH, and 6AH have the following format:

Field	**Size**
Length of following data	Word
Code page ID	Word

The parameter block for minor code 4CH has the following format:

Field	**Size**
Flags	Word
Length of remainder of parameter block (2[*n*+1])	Word
Number of code pages in the following list (*n*)	Word

(more)

Field	Size
Code page 1	Word
Code page 2	Word
.	
.	
.	
Code page *n*	Word

The parameter block for minor code 6BH has the following format, assuming *n* hardware code pages and *m* prepared code pages (*n* <= 12, *m* <= 12):

Field	Size
Length of following data (2[*n*+*m*+2])	Word
Number of hardware code pages (*n*)	Word
Hardware code page 1	Word
Hardware code page 2	Word
.	
.	
.	
Hardware code page *n*	Word
Number of prepared code pages (*m*)	Word
Prepared code page 1	Word
Prepared code page 2	Word
.	
.	
.	
Prepared code page *m*	Word

After a Start Code-Page Preparation (minor code 4CH) call, the program must write the data defining the code-page font to the driver using one or more IOCTL Send Control Data to Character Device (Interrupt 21H Function 44H Subfunction 03H) calls. The format of the data is both device-specific and driver-specific. After the font data has been written to the driver, the program must issue an End Code-Page Preparation (minor code 4DH) call. If no data is written to the driver between the start and end calls, the driver interprets the newly prepared code pages as hardware code pages.

A special variation of Start Code-Page Preparation, called "refresh," is required to actually load the peripheral device with the prepared code pages. The refresh operation is obtained by calling minor code 4CH with each code-page position in the parameter block set to –1 and then immediately calling minor code 4DH.

The device-driver support that corresponds to IOCTL Subfunction 0CH is invoked by the MS-DOS kernel via the Generic IOCTL function (driver command code 19). The category (major) and function (minor) codes described above, along with a pointer to the parameter block, are passed to the driver in the request header. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Installable Device Drivers.

Extended MS-DOS partitions

An extended MS-DOS partition is indicated by a system indicator byte value of 05 in the partition table of the fixed disk's master boot record. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: STRUCTURE OF MS-DOS: MS-DOS Storage Devices. An extended partition is not bootable and can be created on a bootable fixed-disk drive only if that drive already contains a primary MS-DOS partition (system indicator type 01 or 04). Fixed disks that are not bootable can contain an extended partition without a primary partition.

An extended partition is subdivided into extended logical disk volumes, each consisting of an extended boot record and a logical block device. The extended boot record is analogous in structure to the partition table for the fixed disk as a whole; it contains a logical drive table describing the volume and a pointer to the next extended logical volume. The logical block device is an image of a normal MS-DOS disk, including a master block (logical sector 0 containing the BPB describing the device), root directory, FAT, and files area. Each extended volume must start and end on a cylinder boundary.

Van Wolverton
Ray Duncan

Appendix B
Critical Error Codes

Critical errors are returned via Interrupt 24H. If register AL bit 7 is 0, then the error was a disk error; if register AL bit 7 is 1, then the error was a nondisk error. The upper half of DI is undefined; the lower half of DI contains one of the following error-condition codes:

Code	**Description**
00H	Attempt to write on write-protected disk
01H	Unknown drive or unit
02H	Drive not ready
03H	Invalid command
04H	Data error (CRC failed)
05H	Bad request structure length
06H	Seek error
07H	Unknown media type
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0FH	Invalid disk change

Appendix C
Extended Error Codes

The extended error codes used by Interrupt 21H functions consist of four separate codes in the AX, BH, BL, and CH registers. These codes give as much detail as possible about the error and suggest how the issuing program should respond.

AX — Extended Error Code

If an error condition occurs in response to an Interrupt 21H function call, the carry flag is set and one of the following error codes is returned in AX:

Error	Description	Error	Description
01H	Invalid function code	16H	Invalid disk command
02H	File not found	17H	CRC error
03H	Path not found	18H	Invalid length (disk operation)
04H	Too many open files (no handles left)	19H	Seek error
		1AH	Not an MS-DOS disk
05H	Access denied	1BH	Sector not found
06H	Invalid handle	1CH	Out of paper
07H	Memory control blocks destroyed	1DH	Write fault
		1EH	Read fault
08H	Insufficient memory	1FH	General failure
09H	Invalid memory block address	20H	Sharing violation
0AH	Invalid environment	21H	Lock violation
0BH	Invalid format	22H	Wrong disk
0CH	Invalid access code	23H	FCB unavailable
0DH	Invalid data	24H	Sharing buffer overflow
0EH	Reserved	25–31H	Reserved
0FH	Invalid drive	32H	Network request not supported
10H	Attempt to remove the current directory	33H	Remote computer not listening
		34H	Duplicate name on network
11H	Not same device	35H	Network path not found
12H	No more files	36H	Network busy
13H	Disk is write-protected	37H	Network device no longer exists
14H	Bad disk unit	38H	Net BIOS command limit exceeded
15H	Drive not ready		

(more)

Error	Description	Error	Description
39H	Network adapter hardware error	45H	Net BIOS session limit exceeded
3AH	Incorrect response from network	46H	Sharing temporarily paused
		47H	Network request not accepted
3BH	Unexpected network error	48H	Print or disk redirection paused
3CH	Incompatible remote adapter	49–4FH	Reserved
3DH	Print queue full	50H	File exists
3EH	Print queue not full	51H	Reserved
3FH	Print file was canceled (not enough space)	52H	Cannot make directory entry
		53H	Fail on Interrupt 24H
40H	Network name was deleted	54H	Out of network structures
41H	Access denied	55H	Device already assigned
42H	Network device type incorrect	56H	Invalid password
43H	Network name not found	57H	Invalid parameter
44H	Network name limit exceeded	58H	Network data fault

BH — Error Class

BH returns a code that describes the class of error that occurred:

Class	Description
01H	Out of a resource, such as storage or channels
02H	Not an error, but a temporary situation (such as a locked region in a file) that can be expected to end
03H	Authorization problem
04H	An internal error in system software
05H	Hardware failure
06H	A system software failure not the fault of the active process (could be caused by missing or incorrect configuration files, for example)
07H	Application program error
08H	File or item not found
09H	File or item of invalid format or type or otherwise invalid or unsuitable
0AH	File or item interlocked
0BH	Wrong disk in drive, bad spot on disk, or other problem with storage medium
0CH	Other error

BL — Suggested Action

BL returns a code that suggests how the program should respond to the error:

Action	Description
01H	Retry, then prompt user.
02H	Retry after a pause.
03H	If the user entered data such as a drive letter or filename, prompt for it again.
04H	Terminate with cleanup.
05H	Terminate immediately. The system is so unhealthy that the program should exit as soon as possible without taking the time to close files and update indexes.
06H	Error is informational.
07H	Prompt the user to perform some action, such as changing disks, then retry the operation.

CH — Locus

CH returns a code that provides additional information to help locate the area involved in the failure. This code is particularly useful for hardware failures (BH = 05H).

Locus	Description
01H	Unknown
02H	Related to random-access block devices, such as a disk drive
03H	Related to network
04H	Related to serial-access character devices, such as a printer
05H	Related to random-access memory

Procedure

Programs should handle errors by noting the error returned in AX from the original system call and then invoking Interrupt 21H Function 59H to get the extended error information. If no extended error information is provided, the program should respond to the original error code.

The Function 59H system call is available during Interrupt 24H.

Appendix D ASCII Character Set and IBM Extended Character Set

	Number		
Char	**Dec**	**Hex**	**Control**
	0	00	NUL (Null)
☺	1	01	SOH (Start of heading)
☻	2	02	STX (Start of text)
♥	3	03	ETX (End of text)
♦	4	04	EOT (End of transmission)
♣	5	05	ENQ (Enquiry)
♠	6	06	ACK (Acknowledge)
•	7	07	BEL (Bell)
◘	8	08	BS (Backspace)
○	9	09	HT (Horizontal tab)
◙	10	0A	LF (Linefeed)
♂	11	0B	VT (Vertical tab)
♀	12	0C	FF (Formfeed)
♪	13	0D	CR (Carriage return)
♫	14	0E	SO (Shift out)
☼	15	0F	SI (Shift in)
►	16	10	DLE (Data link escape)
◄	17	11	DC1 (Device control 1)
↕	18	12	DC2 (Device control 2)
‼	19	13	DC3 (Device control 3)
¶	20	14	DC4 (Device control 4)
§	21	15	NAK (Negative acknowledge)
▬	22	16	SYN (Synchronous idle)
↨	23	17	ETB (End transmission block)
↑	24	18	CAN (Cancel)
↓	25	19	EM (End of medium)
→	26	1A	SUB (Substitute)
←	27	1B	ESC (Escape)
∟	28	1C	FS (File separator)
↔	29	1D	GS (Group separator)
▲	30	1E	RS (Record separator)
▼	31	1F	US (Unit separator)
<space>	32	20	
!	33	21	
”	34	22	
#	35	23	
$	36	24	
%	37	25	
&	38	26	
’	39	27	
(	40	28	
)	41	29	
*	42	2A	
+	43	2B	
,	44	2C	
-	45	2D	
.	46	2E	
/	47	2F	
0	48	30	
1	49	31	
2	50	32	
3	51	33	
4	52	34	
5	53	35	
6	54	36	
7	55	37	
8	56	38	
9	57	39	
:	58	3A	
;	59	3B	
<	60	3C	
=	61	3D	
>	62	3E	
?	63	3F	
@	64	40	
A	65	41	
B	66	42	
C	67	43	
D	68	44	
E	69	45	
F	70	46	
G	71	47	
H	72	48	

(more)

	Number			Number				Number	
Char	Dec	Hex	Char	Dec	Hex	Control	Char	Dec	Hex
I	73	49	z	122	7A		½	171	AB
J	74	4A	{	123	7B		¼	172	AC
K	75	4B	¦	124	7C		¡	173	AD
L	76	4C	}	125	7D		«	174	AE
M	77	4D	~	126	7E		»	175	AF
N	78	4E	Δ	127	7F	DEL	░	176	B0
O	79	4F	Ç	128	80		▒	177	B1
P	80	50	ü	129	81		▓	178	B2
Q	81	51	é	130	82		│	179	B3
R	82	52	â	131	83		┤	180	B4
S	83	53	ä	132	84		╡	181	B5
T	84	54	à	133	85		╢	182	B6
U	85	55	å	134	86		╖	183	B7
V	86	56	ç	135	87		╕	184	B8
W	87	57	ê	136	88		╣	185	B9
X	88	58	ë	137	89		║	186	BA
Y	89	59	è	138	8A		╗	187	BB
Z	90	5A	ï	139	8B		╝	188	BC
[	91	5B	î	140	8C		╜	189	BD
\	92	5C	ì	141	8D		╛	190	BE
]	93	5D	Ä	142	8E		┐	191	BF
^	94	5E	Å	143	8F		└	192	C0
_	95	5F	É	144	90		┴	193	C1
`	96	60	æ	145	91		┬	194	C2
a	97	61	Æ	146	92		├	195	C3
b	98	62	ô	147	93		─	196	C4
c	99	63	ö	148	94		┼	197	C5
d	100	64	ò	149	95		╞	198	C6
e	101	65	û	150	96		╟	199	C7
f	102	66	ù	151	97		╚	200	C8
g	103	67	ÿ	151	98		╔	201	C9
h	104	68	Ö	152	99		╩	202	CA
i	105	69	Ü	154	9A		╦	203	CB
j	106	6A	¢	155	9B		╠	204	CC
k	107	6B	£	156	9C		═	205	CD
l	108	6C	¥	157	9D		╬	206	CE
m	109	6D	₧	158	9E		╧	207	CF
n	110	6E	ƒ	159	9F		╨	208	D0
o	111	6F	á	160	A0		╤	209	D1
p	112	70	í	161	A1		╥	210	D2
q	113	71	ó	162	A2		╙	211	D3
r	114	72	ú	163	A3		╘	212	D4
s	115	73	ñ	164	A4		╒	213	D5
t	116	74	Ñ	165	A5		╓	214	D6
u	117	75	ª	166	A6		╫	215	D7
v	118	76	º	167	A7		╪	216	D8
w	119	77	¿	168	A8		┘	217	D9
x	120	78	⌐	169	A9		┌	218	DA
y	121	79	¬	170	AA		█	219	DB

(more)

	Number			Number			Number	
Char	**Dec**	**Hex**	**Char**	**Dec**	**Hex**	**Char**	**Dec**	**Hex**
▄	220	DC	Φ	232	E8	⌠	244	F4
▌	221	DD	Θ	233	E9	⌡	245	F5
▐	222	DE	Ω	234	EA	÷	246	F6
▀	223	DF	δ	235	EB	≈	247	F7
α	224	E0	∞	236	EC	°	248	F8
β	225	E1	φ	237	ED	•	249	F9
Γ	226	E2	ε	238	EE	·	250	FA
π	227	E3	∩	239	EF	√	251	FB
Σ	228	E4	≡	240	F0	η	252	FC
σ	229	E5	±	241	F1	²	253	FD
µ	230	E6	≥	242	F2	■	254	FE
τ	231	E7	≤	243	F3		255	FF

Appendix E
EBCDIC Character Set

	Number			Number			Number	
Char	**Dec**	**Hex**	**Char**	**Dec**	**Hex**	**Char**	**Dec**	**Hex**
NUL	0	00		41	29		82	52
SOH	1	01	SM	42	2A		83	53
STX	2	02	CU2	43	2B		84	54
ETX	3	03		44	2C		85	55
PF	4	04	ENQ	45	2D		86	56
HT	5	05	ACK	46	2E		87	57
LC	6	06	BEL	47	2F		88	58
DEL	7	07		48	30		89	59
GE	8	08		49	31	!	90	5A
RLF	9	09	SYN	50	32	$	91	5B
SMM	10	0A		51	33	*	92	5C
VT	11	0B	PN	52	34	)	93	5D
FF	12	0C	RS	53	35	;	94	5E
CR	13	0D	UC	54	36	¬	95	5F
SO	14	0E	EOT	55	37	-	96	60
SI	15	0F		56	38	/	97	61
DLE	16	10		57	39		98	62
DC1	17	11		58	3A		99	63
DC2	18	12	CU3	59	3B		100	64
TM	19	13	DC4	60	3C		101	65
RES	20	14	NAK	61	3D		102	66
NL	21	15		62	3E		103	67
BS	22	16	SUB	63	3F		104	68
IL	23	17	Sp	64	40		105	69
CAN	24	18		65	41	¦	106	6A
EM	25	19		66	42	,	107	6B
CC	26	1A		67	43	%	108	6C
CU1	27	1B		68	44	_	109	6D
IFS	28	1C		69	45	>	110	6E
IGS	29	1D		70	46	?	111	6F
IRS	30	1E		71	47		112	70
IUS	31	IF		72	48		113	71
DS	32	20		73	49		114	72
SOS	33	21	¢	74	4A		115	73
FS	34	22	.	75	4B		116	74
	35	23	<	76	4C		117	75
BYP	36	24	(	77	4D		118	76
LF	37	25	+	78	4E		119	77
ETB	38	26	|	79	4F		120	78
ESC	39	27	&	80	50		121	79
	40	28		81	51		122	7A

	Number			Number			Number	
Char	**Dec**	**Hex**	**Char**	**Dec**	**Hex**	**Char**	**Dec**	**Hex**
#	123	7B	y	168	A8	N	213	D5
@	124	7C	z	169	A9	O	214	D6
'	125	7D		170	AA	P	215	D7
=	126	7E		171	AB	Q	216	D8
"	127	7F		172	AC	R	217	D9
	128	80		173	AD		218	DA
a	129	81		174	AE		219	DB
b	130	82		175	AF		220	DC
c	131	83		176	B0		221	DD
d	132	84		177	B1		222	DE
e	133	85		178	B2		223	DF
f	134	86		179	B3	\	224	E0
g	135	87		180	B4		225	E1
h	136	88		181	B5	S	226	E2
i	137	89		182	B6	T	227	E3
	138	8A		183	B7	U	228	E4
	139	8B		184	B8	V	229	E5
	140	8C		185	B9	W	230	E6
	141	8D		186	BA	X	231	E7
	142	8E		187	BB	Y	232	E8
	143	8F		188	BC	Z	233	E9
	144	90		189	BD		234	EA
j	145	91		190	BE		235	EB
k	146	92		191	BF	⑁	236	EC
l	147	93	{	192	C0		237	ED
m	148	94	A	193	C1		238	EE
n	149	95	B	194	C2		239	EF
o	150	96	C	195	C3	0	240	F0
p	151	97	D	196	C4	1	241	F1
q	152	98	E	197	C5	2	242	F2
r	153	99	F	198	C6	3	243	F3
	154	9A	G	199	C7	4	244	F4
	155	9B	H	200	C8	5	245	F5
	156	9C	I	201	C9	6	246	F6
	157	9D		202	CA	7	247	F7
	158	9E		203	CB	8	248	F8
	159	9F	⑀	204	CC	9	249	F9
	160	A0		205	CD	|	250	FA
~	161	A1	⑂	206	CE		251	FB
s	162	A2		207	CF		252	FC
t	163	A3	}	208	D0		253	FD
u	164	A4	J	209	D1		254	FE
v	165	A5	K	210	D2	EO	255	FF
w	166	A6	L	211	D3			
x	167	A7	M	212	D4			

Appendix F
ANSI.SYS Key and Extended Key Codes

The following escape sequence allows redefinition of keyboard keys to a specified *string*:

ESC[*code*;*string*;...p

where:

string is either the ASCII code for a single character or a string contained in quotation marks. For example, both 65 and "A" can be used to represent an uppercase A.

code is one or more of the following values that represent keyboard keys. Semicolons shown in this table must be entered in addition to the required semicolons in the command line.

Key	**Code**			
	Alone	**Shift-**	**Ctrl-**	**Alt-**
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;86	0;96	0;106
F4	0;62	0;87	0;97	0;107
F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
Home	0;71	55	0;119	–
Up Arrow	0;72	56	–	–
Pg Up	0;73	57	0;132	–
Left Arrow	0;75	52	0;115	–
Down Arrow	0;77	54	0;116	–
End	0;79	49	0;117	–
Down Arrow	0;80	50	–	–
Pg Dn	0;81	51	0;118	–
Ins	0;82	48	–	–
Del	0;83	46	–	–
PrtSc	–	–	0;114	–
A	97	65	1	0;30

(more)

Key	Code			
	Alone	Shift-	Ctrl-	Alt-
B	98	66	2	0;48
C	99	67	3	0;46
D	100	68	4	0;32
E	101	69	5	0;18
F	102	70	6	0;33
G	103	71	7	0;34
H	104	72	8	0;35
I	105	73	9	0;23
J	106	74	10	0;36
K	107	75	11	0;37
L	108	76	12	0;38
M	109	77	13	0;50
N	110	78	14	0;49
O	111	79	15	0;24
P	112	80	16	0;25
Q	113	81	17	0;16
R	114	82	18	0;19
S	115	83	19	0;31
T	116	84	20	0;20
U	117	85	21	0;22
V	118	86	22	0;47
W	119	87	23	0;17
X	120	88	24	0;45
Y	121	89	25	0;21
Z	122	90	26	0;44
1	49	33	–	0;120
2	50	64	–	0;121
3	51	35	–	0;122
4	52	36	–	0;123
5	53	37	–	0;124
6	54	94	–	0;125
7	55	38	–	0;126
8	56	42	–	0;127
9	57	40	–	0;128
0	48	41	–	0;129
–	45	95	–	0;130
=	61	43	–	0;131
Tab	9	0;15	–	–
Null	0;3	–	–	–

Appendix G
File Control Block (FCB) Structure

Figures G-1 and G-2 (memory block diagrams) and Tables G-1 and G-2 describe the structure of normal and extended file control blocks (FCBs).

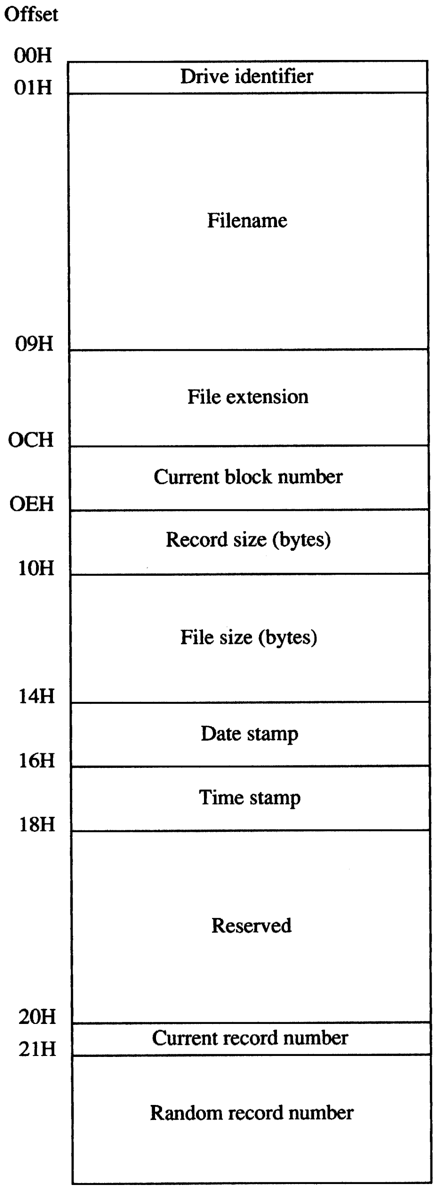


Figure G-1. Structure of a normal file control block.

Table G-1. Elements of a Normal File Control Block.

Element	Maintained by	Comments
Drive identifier	Program	Designates 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, MS-DOS alters the byte during the open or create operation to reflect the actual drive used.
Filename	Program	Standard eight-character filename; must be left justified and must be padded with blanks if fewer than eight characters. A device name (for example, PRN) can be used; there is no colon after a device name.
File extension	Program	Three-character file extension; must be left justified and must be padded with blanks if fewer than three characters.
Current block number	Program	Zero when the file is opened; the current block number and the current record number combined make up the record pointer during sequential file access.
Record size	Program	Set to 128 when the file is opened or created; the program can modify the field afterward to any desired record size.*
File size	MS-DOS	The size of the file in bytes; the first 2 bytes of this 4-byte field are the least significant bytes of the file size.
Date stamp	MS-DOS	The date of the last write operation on the file; follows the same format used by Interrupt 21H file handle Function 57H (Get/Set Time and Date): **Bits** — **Contents** 9–15 — Year (relative to 1980) 5–8 — Month (1–12) 0–4 — Day of month (1–31)
Time stamp	MS-DOS	The time of the last write operation on the file; follows the same format used by Interrupt 21H file handle Function 57H (Get/Set Time and Date): **Bits** — **Contents** 11–15 — Hours (0–23) 5–10 — Minutes (0–59) 0–4 — Number of 2-second increments (0–29)

(more)

Table G-1. *Continued.*

Element	Maintained by	Comments
Current record number	Program	Limited to the range 0 through 127; there are 128 records per block. The beginning of a file is record 0 of block 0. Together with the current block number, this field constitutes the record pointer used during sequential read and write operations. MS-DOS does not automatically initialize this field when a file is opened.
Random record pointer	Program	Identifies the record to be transferred by the Interrupt 21H 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).

* If the record size is made larger than 128 bytes, the default data transfer area (DTA) in the program segment prefix (PSP) cannot be used because it will collide with the program's own code or data.

Table G-2. Additional Elements of an Extended File Control Block.

Element	Maintained by	Comments
Extended FCB flag	Program	0FFH tells MS-DOS this is an extended (44-byte) FCB.
File attribute byte	Program	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

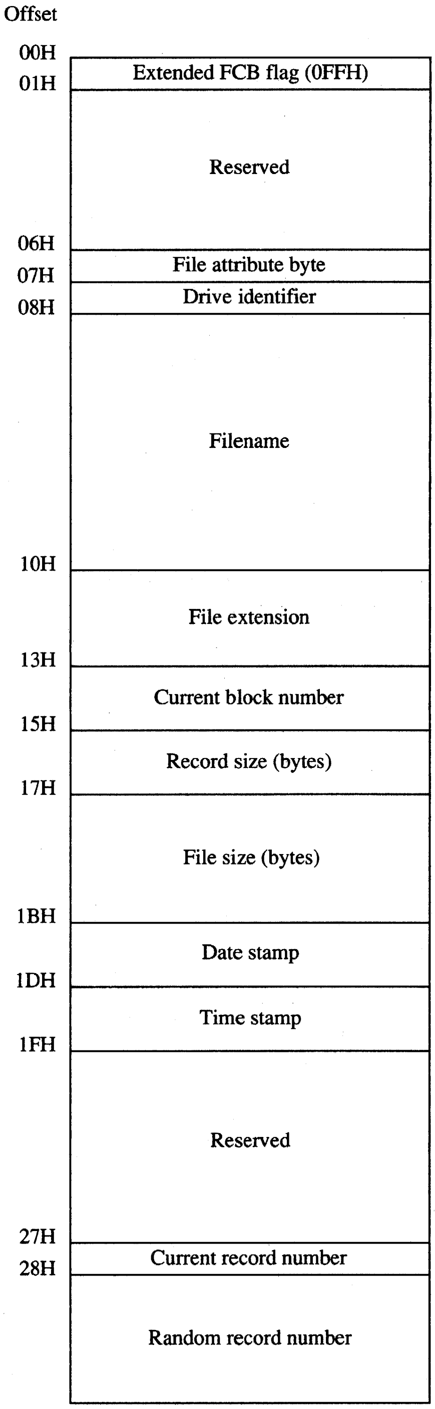


Figure G-2. Structure of an extended file control block.

Appendix H
Program Segment Prefix (PSP) Structure

Figure H-1 (memory block diagram) illustrates the structure of the program segment prefix (PSP).

Offset	Size (in bytes)	Contents
00H (0)	2	INT 20H instruction
02H (2)	2	Address of last segment allocated to program
04H (4)	1	Reserved; normally 0
05H (5)	5	Long call to MS-DOS function dispatcher
0AH (10)	4	Terminate program interrupt vector (Interrupt 22H)
0EH (14)	4	Ctrl-C handler interrupt vector (Interrupt 23H)
12H (18)	4	Critical error handler interrupt vector (Interrupt 24H)
16H (22)	22	Reserved
2CH (44)	2	Segment address of environment
2EH (46)	34	Reserved
50H (80)	3	INT 21H, RETF instructions
53H (83)	9	Reserved
5CH (92)	16	Default file control block 1
6CH (108)	20	Default file control block 2 (overlaid if FCB 1 opened)
80H (128)	127	Command tail and default DTA
FFH (255)		

Figure H-1. Structure of the program segment prefix.

Appendix I
8086/8088/80286/80386 Instruction Sets

The 8086/8088 Instruction Set

Mnemonic	Description	Mnemonic	Description
AAA	ASCII adjust after addition	JB	Jump on below
AAD	ASCII adjust before division	JBE	Jump on below or equal
AAM	ASCII adjust after multiplication	JC	Jump on carry
AAS	ASCII adjust after subtraction	JCXZ	Jump on CX zero
ADC	Add with carry	JE	Jump on equal
ADD	Add	JG	Jump on greater
AND	Logical AND	JGE	Jump on greater or equal
CALL	Call procedure	JL	Jump on less than
CBW	Convert byte to word	JLE	Jump on less than or equal
CLC	Clear carry flag	JMP	Jump unconditionally
CLD	Clear direction flag	JNA	Jump on not above
CLI	Clear interrupt flag	JNAE	Jump on not above or equal
CMC	Complement carry flag	JNB	Jump on not below
CMP	Compare	JNBE	Jump on not below or equal
CMPS	Compare string	JNC	Jump on no carry
CMPSB	Compare byte string	JNE	Jump on not equal
CMPSW	Compare word string	JNG	Jump on not greater
CWD	Convert word to doubleword	JNGE	Jump on not greater or equal
DAA	Decimal adjust for addition	JNL	Jump on not less than
DAS	Decimal adjust for subtraction	JNLE	Jump on not less than or equal
DEC	Decrement by 1	JNO	Jump on not overflow
DIV	Unsigned divide	JNP	Jump on not parity
ESC	Escape	JNS	Jump on not sign
HLT	Halt	JNZ	Jump on not zero
IDIV	Integer divide	JO	Jump on overflow
IMUL	Integer multiply	JP	Jump on parity
IN	Input from port	JPE	Jump on parity even
INC	Increment by 1	JPO	Jump on parity odd
INT	Call to interrupt procedure	JS	Jump on sign
INTO	Interrupt on overflow	JZ	Jump on zero
IRET	Interrupt on return	LAHF	Load AH with flags
JA	Jump on above	LDS	Load pointer into DS
JAE	Jump on above or equal	LEA	Load effective address

(more)

Mnemonic	Description	Mnemonic	Description
LES	Load pointer into ES	REPNE	Repeat while not equal
LOCK	Lock the bus	REPNZ	Repeat while not zero
LODS	Load string	REPZ	Repeat while zero
LODSB	Load byte (string)	RET	Return
LODSW	Load word (string)	ROL	Rotate left
LOOP	Loop	ROR	Rotate right
LOOPE	Loop while equal	SAHF	Store AH into flags
LOOPNE	Loop while not equal	SAL	Shift arithmetic left
LOOPNZ	Loop while not zero	SAR	Shift arithmetic right
LOOPZ	Loop while zero	SBB	Subtract with borrow
MOV	Move data	SCAS	Scan string
MOVS	Move data from string to string	SCASB	Scan byte (string)
MOVSB	Move byte (string)	SCASW	Scan word (string)
MOVSW	Move word (string)	SHL	Shift logical left
MUL	Multiply	SHR	Shift logical right
NEG	Negate	STC	Set carry flag
NOP	No operation	STD	Set direction flag
NOT	Logical NOT	STI	Set interrupt flag
OR	Logical OR	STOS	Store string
OUT	Output to port	STOSB	Store byte (string)
POP	Pop top of stack	STOSW	Store word (string)
POPF	Pop stack into flags	SUB	Subtract
PUSH	Push onto stack	TEST	Logical compare
PUSHF	Push flags onto stack	WAIT	Enter wait state
RCL	Rotate through carry left	XCHG	Exchange
RCR	Rotate through carry right	XLAT	Translate
REP	Repeat	XOR	Exclusive OR
REPE	Repeat while equal		

The 80286 Instruction Set

Mnemonic	Description	Mnemonic	Description
AAA	ASCII adjust after addition	AND	Logical AND
AAD	ASCII adjust before division	ARPL	Adjust RPL field of selector
AAM	ASCII adjust after multiplication	BOUND	Check array index against bounds
AAS	ASCII adjust after subtraction	CALL	Call procedure
ADC	Add with carry	CBW	Convert byte to word
ADD	Add	CLC	Clear carry flag

(more)

Mnemonic	Description	Mnemonic	Description
CLD	Clear direction flag	JNE	Jump on not equal
CLI	Clear interrupt flag	JNG	Jump on not greater
CLTS	Clear task switched flag	JNGE	Jump on not greater or equal
CMC	Complement carry flag	JNL	Jump on not less than
CMP	Compare	JNLE	Jump on not less than or equal
CMPS	Compare string	JNO	Jump on not overflow
CMPSB	Compare byte string	JNP	Jump on not parity
CMPSW	Compare word string	JNS	Jump on not sign
CWD	Convert word to doubleword	JNZ	Jump on not zero
DAA	Decimal adjust for addition	JO	Jump on overflow
DAS	Decimal adjust for subtraction	JP	Jump on parity
DEC	Decrement by 1	JPE	Jump on parity even
DIV	Unsigned divide	JPO	Jump on parity odd
ENTER	Make stack frame	JS	Jump on sign
	(for procedure parameters)	JZ	Jump on zero
ESC	Escape	LAHF	Load AH with flags
HLT	Halt	LAR	Load access-rights byte
IDIV	Integer divide	LDS	Load pointer into DS
IMUL	Integer multiply	LEA	Load effective address
IN	Input from port	LEAVE	High-level procedure exit
INC	Increment by 1	LES	Load pointer into ES
INS	Input string from port	LGDT	Load global descriptor table
INT	Call to interrupt procedure	LIDT	Load interrupt descriptor table
INTO	Interrupt on overflow	LLDT	Load local descriptor table
IRET	Interrupt on return	LMSW	Load machine status word
JA	Jump on above	LOCK	Lock the bus
JAE	Jump on above or equal	LODS	Load string
JB	Jump on below	LODSB	Load byte (string)
JBE	Jump on below or equal	LODSW	Load word (string)
JC	Jump on carry	LOOP	Loop
JCXZ	Jump on CX zero	LOOPE	Loop while equal
JE	Jump on equal	LOOPNE	Loop while not equal
JG	Jump on greater	LOOPNZ	Loop while not zero
JGE	Jump on greater or equal	LOOPZ	Loop while zero
JL	Jump on less than	LSL	Load segment limit
JLE	Jump on less than or equal	LTR	Load task register
JMP	Jump unconditionally	MOV	Move data
JNA	Jump on not above	MOVS	Move data from string to string
JNAE	Jump on not above or equal	MOVSB	Move byte (string)
JNB	Jump on not below	MOVSW	Move word (string)
JNBE	Jump on not below or equal	MUL	Multiply
JNC	Jump on no carry	NEG	Negate

(more)

Mnemonic	Description	Mnemonic	Description
NOP	No operation	SCAS	Scan string
NOT	Logical NOT	SCASB	Scan byte (string)
OR	Logical OR	SCASW	Scan word (string)
OUT	Output to port	SGDT	Store global descriptor table
OUTS	Output string to port	SHL	Shift logical left
POP	Pop top of stack	SHR	Shift logical right
POPA	Pop eight 16-bit registers	SIDT	Store interrupt descriptor table
POPF	Pop stack into flags	SLDT	Store local descriptor table
PUSH	Push onto stack	SMSW	Store machine status word
PUSHA	Push eight 16-bit registers	STC	Set carry flag
PUSHF	Push flags onto stack	STD	Set direction flag
RCL	Rotate through carry left	STI	Set interrupt flag
RCR	Rotate through carry right	STOS	Store string
REP	Repeat	STOSB	Store byte (string)
REPE	Repeat while equal	STOSW	Store word (string)
REPNE	Repeat while not equal	STR	Store task register
REPNZ	Repeat while not zero	SUB	Subtract
REPZ	Repeat while zero	TEST	Logical compare
RET	Return	VERR	Verify a segment for reading
ROL	Rotate left	VERW	Verify a segment for writing
ROR	Rotate right	WAIT	Enter wait state
SAHF	Store AH into flags	XCHG	Exchange
SAL	Shift arithmetic left	XLAT	Translate
SAR	Shift arithmetic right	XOR	Exclusive OR
SBB	Subtract with borrow		

The 80386 Instruction Set

Mnemonic	Description	Mnemonic	Description
AAA	ASCII adjust after addition	BSF	Bit scan forward
AAD	ASCII adjust before division	BSR	Bit scan reverse
AAM	ASCII adjust after multiplication	BT	Bit test
AAS	ASCII adjust after subtraction	BTC	Bit test and complement
ADC	Add with carry	BTR	Bit test and reset
ADD	Add	BTS	Bit test and set
AND	Logical AND	CALL	Call procedure
ARPL	Adjust RPL field of selector	CBW	Convert byte to word
BOUND	Check array index against bounds	CDQ	Convert doubleword to quad word

(more)

Mnemonic	Description	Mnemonic	Description
CLC	Clear carry flag	JMP	Jump unconditionally
CLD	Clear direction flag	JNA	Jump on not above
CLI	Clear interrupt flag	JNAE	Jump on not above or equal
CLTS	Clear task switched flag	JNB	Jump on not below
CMC	Complement carry flag	JNBE	Jump on not below or equal
CMP	Compare	JNC	Jump on no carry
CMPS	Compare string	JNE	Jump on not equal
CMPSB	Compare byte string	JNG	Jump on not greater
CMPSD	Compare doubleword string	JNGE	Jump on not greater or equal
CMPSW	Compare word string	JNL	Jump on not less than
CWD	Convert word to doubleword	JNLE	Jump on not less than or equal
DAA	Decimal adjust for addition	JNO	Jump on not overflow
DAS	Decimal adjust for subtraction	JNP	Jump on not parity
DEC	Decrement by 1	JNS	Jump on not sign
DIV	Unsigned divide	JNZ	Jump on not zero
ENTER	Make stack frame	JO	Jump on overflow
	(for procedure parameters)	JP	Jump on parity
ESC	Escape	JPE	Jump on parity even
HLT	Halt	JPO	Jump on parity odd
IDIV	Integer divide	JS	Jump on sign
IMUL	Integer multiply	JZ	Jump on zero
IN	Input from port	LAHF	Load AH with flags
INC	Increment by 1	LAR	Load access-rights byte
INS	Input string from port	LDS	Load pointer into DS
INSD	Input doubleword from port	LEA	Load effective address
INT	Call to interrupt procedure	LEAVE	High-level procedure exit
INTO	Interrupt on overflow	LES	Load pointer into ES
IRET	Interrupt on return	LFS	Load pointer into FS
IRETD	Interrupt return to	LGDT	Load global descriptor table
	virtual 8086 mode	LGS	Load pointer into GS
JA	Jump on above	LIDT	Load interrupt descriptor table
JAE	Jump on above or equal	LLDT	Load local descriptor table
JB	Jump on below	LMSW	Load machine status word
JBE	Jump on below or equal	LOCK	Lock the bus
JC	Jump on carry	LODS	Load string
JCXZ	Jump on CX zero	LODSB	Load byte (string)
JE	Jump on equal	LODSD	Load doubleword (string)
JECXZ	Jump on ECX zero	LODSW	Load word (string)
JG	Jump on greater	LOOP	Loop
JGE	Jump on greater or equal	LOOPE	Loop while equal
JL	Jump on less than	LOOPNE	Loop while not equal
JLE	Jump on less than or equal	LOOPNZ	Loop while not zero

(more)

Mnemonic	Description	Mnemonic	Description
LOOPZ	Loop while zero	ROL	Rotate left
LSL	Load segment limit	ROR	Rotate right
LSS	Load pointer into SS	SAHF	Store AH into flags
LTR	Load task register	SAL	Shift arithmetic left
MOV	Move data	SAR	Shift arithmetic right
MOVS	Move data from string to string	SBB	Subtract with borrow
MOVSB	Move byte (string)	SCAS	Scan string
MOVSD	Move doubleword (string)	SCASB	Scan byte (string)
MOVSW	Move word (string)	SCASD	Scan doubleword (string)
MOVSX	Move with sign extend	SCASW	Scan word (string)
MOVZX	Move with zero extend	SET	Byte set on condition
MUL	Multiply	SGDT	Store global descriptor table
NEG	Negate	SHL	Shift logical left
NOP	No operation	SHLD	Double precision shift left
NOT	Logical NOT	SHR	Shift logical right
OR	Logical OR	SHRD	Double precision shift right
OUT	Output to port	SIDT	Store interrupt descriptor table
OUTS	Output string to port	SLDT	Store local descriptor table
POP	Pop top of stack	SMSW	Store machine status word
POPA	Pop eight 16-bit registers	STC	Set carry flag
POPAD	Pop eight 32-bit registers	STD	Set direction flag
POPF	Pop stack into flags	STI	Set interrupt flag
POPFD	Loads doubleword into EFLAGS	STOS	Store string
PUSH	Push onto stack	STOSB	Store byte (string)
PUSHA	Push eight 16-bit registers	STOSD	Store doubleword (string)
PUSHAD	Push eight 32-bit registers	STOSW	Store word (string)
PUSHED	Push EFLAGS	STR	Store task register
PUSHF	Push flags onto stack	SUB	Subtract
RCL	Rotate through carry left	TEST	Logical compare
RCR	Rotate through carry right	VERR	Verify a segment for reading
REP	Repeat	VERW	Verify a segment for writing
REPE	Repeat while equal	WAIT	Enter wait state
REPNE	Repeat while not equal	XCHG	Exchange
REPNZ	Repeat while not zero	XLAT	Translate
REPZ	Repeat while zero	XOR	Exclusive OR
RET	Return		

Appendix J
Common MS-DOS Filename Extensions

The Microsoft systems programs and language products commonly use the following filename extensions:

Extension	Program/System	Description
.@@@	MS-DOS	Backup ID file
.$$$	EDLIN	Backup filename if out of disk space; error condition
.ASC	Generic	ASCII text file
.ASM	MASM	Assembly-language source code
.BAK	Generic	Backup file
.BAS	BASIC	BASIC language source code
.BAT	MS-DOS	Batch file (contains MS-DOS command lines)
.BIN	Generic	Binary file
.C	C	C language source code
.CAL	Windows	Calendar file
.COB	COBOL	COBOL language source code
.COD	Generic	Object listing file
.COM	MS-DOS	Executable program file
.CRD	Windows	Cardfile file
.CRF	MASM	Cross-reference file
.DAT	Generic	Data file
.DBG	COBOL	Debug file
.DEF	Windows	Module definition file
.DOC	Generic	Documentation or document file
.DRV	Generic	Driver file
.ERR	Generic	Error file
.EXE	MS-DOS	Executable program file
.FNT	Generic	Font file
.FON	Generic	Font file
.FOR	FORTRAN	FORTRAN language source code
.GRB	Windows	Grab file (snapshot)
.H	C	Include file
.HEX	MS-DOS	INTEL hexadecimal format file
.HLP	Generic	Help file
.INC	Generic	Include file
.INI	Windows	Initialization file

(more)

Extension	Program/System	Description
.INT	COBOL	Object file
.LIB	Generic	Library file
.LST	Generic	List file
.MAP	Generic	Address map file
.MOD	Generic	Module file
.MSG	COBOL	Message file
.MSP	Windows	Windows Paint file
.OBJ	Generic	Relocatable object module
.OVL	Generic	Overlay file
.OVR	COBOL	Compiler overlay file
.PAS	PASCAL	PASCAL language source code
.PIF	Windows	Program information file
.QLB	Generic	Library file for Microsoft's Quick products
.RC	Windows	Resource script file
.REF	CREF	Cross-reference listing file
.RES	Windows	Compiled resource file
.SCR	Generic	Script file
.SYM	Generic	Symbol file
.SYS	Generic	System file or device driver
.TMP	Generic	Temporary file
.TRM	Windows	Terminal file
.TXT	Generic	Text file or Windows Notepad file
.WRI	Windows	Write file

Appendix K
Segmented (New) .EXE File Header Format

Microsoft Windows requires much more information about a program than is available in the format of the .EXE executable file supported by MS-DOS. For example, Windows needs to identify the various segments of a program as code segments or data segments, to identify exported and imported functions, and to store the program's resources (such as icons, cursors, menus, and dialog-box templates). Windows must also support dynamically linkable library modules containing routines that programs and other library modules can call. For this reason, Windows programs use an expanded .EXE header format called the New Executable file header format. This format is used for Windows programs, Windows library modules, and resource-only files such as the Windows font resource files.

The Old Executable Header

The New Executable file header format incorporates the existing MS-DOS executable file header format. In fact, the beginning of a New Executable file is simply a normal MS-DOS .EXE header. The 4 bytes at offset 3CH are a pointer to the beginning of the New Executable header. (Offsets are from the beginning of the Old Executable header.)

Offset	**Length (bytes)**	**Contents**
00H	1	Signature byte *M*
01H	1	Signature byte *Z*
3CH	4	Offset of New Executable header from beginning of file

This normal MS-DOS .EXE header can contain size and relocation information for a non-Windows MS-DOS program that is contained within the .EXE file along with the Windows program. This program is run when the .EXE file is executed from the MS-DOS command line. Most Windows programmers use a standard program that simply prints the message *This program requires Microsoft Windows.*

The New Executable Header

The beginning of the New Executable file header contains information about the location and size of various tables within the header. (Offsets are from the beginning of the New Executable header.)

Offset	Length (bytes)	Contents
00H	1	Signature byte *N*
01H	1	Signature byte *E*
02H	1	LINK version number
03H	1	LINK revision number
04H	2	Offset of beginning of entry table relative to beginning of New Executable header
06H	2	Length of entry table
08H	4	32-bit checksum of entire contents of file, using zero for these 4 bytes
0CH	2	Module flag word (*see* below)
0EH	2	Segment number of automatic data segment (0 if neither SINGLEDATA nor MULTIPLEDATA flag is set in flag word)
10H	2	Initial size of local heap to be added to automatic data segment (0 if there is no local heap)
12H	2	Initial size of stack to be added to automatic data segment (0 for library modules)
14H	2	Initial value of instruction pointer (IP) register on entry to program
16H	2	Initial segment number for setting code segment (CS) register on entry to program
18H	2	Initial value of stack pointer (SP) register on entry to program (0 if stack segment is automatic data segment; stack should be set above static data area and below local heap in automatic data segment)

(more)

Offset	Length (bytes)	Contents
1AH	2	Segment number for setting stack segment (SS) register on entry to program (0 for library modules)
1CH	2	Number of entries in segment table
1EH	2	Number of entries in module reference table
20H	2	Number of bytes in nonresident names table
22H	2	Offset of beginning of segment table relative to beginning of New Executable header
24H	2	Offset of beginning of resource table relative to beginning of New Executable header
26H	2	Offset of beginning of resident names table relative to beginning of New Executable header
28H	2	Offset of beginning of module reference table relative to beginning of New Executable header
2AH	2	Offset of beginning of imported names table relative to beginning of New Executable header
2CH	4	Offset of nonresident names table relative to beginning of file
30H	2	Number of movable entry points listed in entry table
32H	2	Alignment shift count (0 is equivalent to 9)
34H	12	Reserved for expansion

The module flag word at offset 0CH in the New Executable header is defined as shown in Figure K-1.

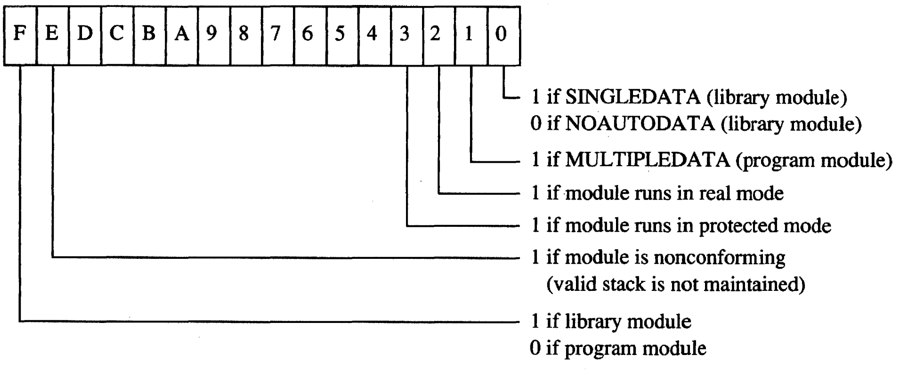

F
E
D
C
B
A
9
8
7
6
5
4
3
2
1
0
1 if SINGLEDATA (library module)
0 if NOAUTODATA (library module)
1 if MULTIPLEDATA (program module)
1 if module runs in real mode
1 if module runs in protected mode
1 if module is nonconforming
(valid stack is not maintained)
1 if library module
0 if program module

Figure K-1. The module flag word.

The segment table

This table contains one 8-byte record for every code and data segment in the program or library module. Each segment has an ordinal number associated with it. For example, the first segment has an ordinal number of 1. These segment numbers are used to reference the segments in other sections of the New Executable file. (Offsets are from the beginning of the record.)

Offset	Length (bytes)	Contents
00H	2	Offset of segment relative to beginning of file after shifting value left by alignment shift count
02H	2	Length of segment (0000H for segment of 65536 bytes)
04H	2	Segment flag word (*see* below)
06H	2	Minimum allocation size for segment; that is, amount of space Windows reserves in memory for segment (0000H for minimum allocation size of 65536 bytes)

The segment flag word is defined as shown in Figure K-2.

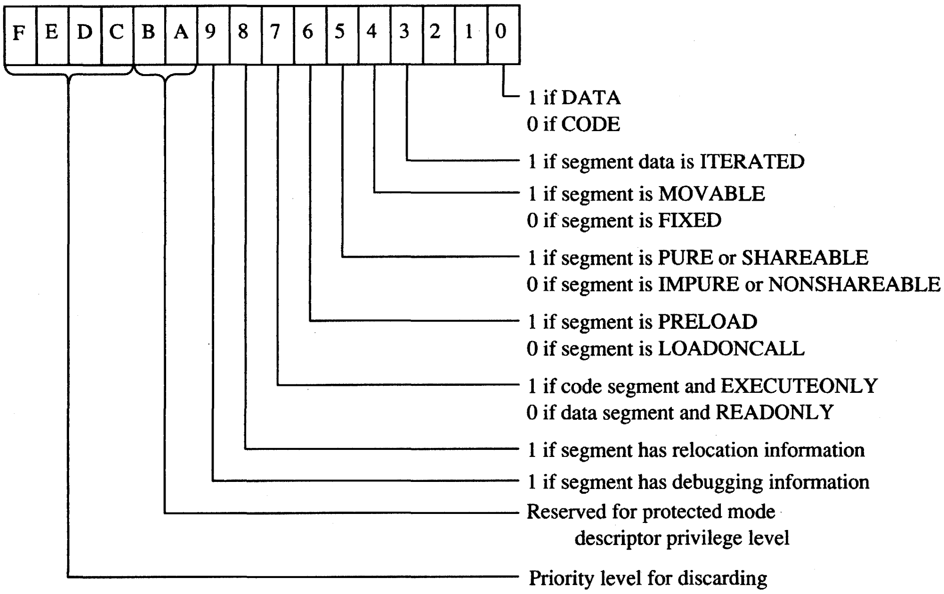

F
E
D
C
B
A
9
8
7
6
5
4
3
2
1
0
1 if DATA
0 if CODE
1 if segment data is ITERATED
1 if segment is MOVABLE
0 if segment is FIXED
1 if segment is PURE or SHAREABLE
0 if segment is IMPURE or NONSHAREABLE
1 if segment is PRELOAD
0 if segment is LOADONCALL
1 if code segment and EXECUTEONLY
0 if data segment and READONLY
1 if segment has relocation information
1 if segment has debugging information
Reserved for protected mode descriptor privilege level
Priority level for discarding

Figure K-2. The segment flag word.

The resource table

Resources are segments that contain data but are not included in a program's normal data segments. Resources are commonly used in Windows programs to store menus, dialog-box templates, icons, cursors, and text strings, but they can also be used for any type of read-only data. Each resource has a type and a name, both of which can be represented by either a number or an ASCII name.

The resource table begins with a resource shift count used for adjusting other values in the table. (Offsets are from the beginning of the table.)

Offset	Length (bytes)	Contents
00H	2	Resource shift count

This is followed by one or more resource groups, each defining one or more resources. (Offsets are from the beginning of the group.)

Offset	Length (bytes)	Contents
00H	2	Resource type (0 if end of table) If high bit set, type represented by predetermined number (high bit not shown): 1 Cursor 2 Bitmap 3 Icon 4 Menu template 5 Dialog-box template 6 String table 7 Font directory 8 Font 9 Keyboard-accelerator table If high bit not set, type is ASCII text string and this value is offset from beginning of resource table, pointing to 1-byte value with number of bytes in string followed by string itself.
02H	2	Number of resources of this type
04H	4	Reserved for run-time use
08H	12 each	Resource description

Each resource description requires 12 bytes. (Offsets are from the beginning of the description.)

Offset	Length (bytes)	Contents
00H	2	Offset of resource relative to beginning of file after shifting left by resource shift count
02H	2	Length of resource after shifting left by resource shift count
04H	2	Resource flag word (*see* below)
06H	2	Resource name If high bit set, represented by a number; otherwise, type is ASCII text string and this value is offset from beginning of resource table, pointing to 1-byte value with number of bytes in string followed by string itself.
08H	4	Reserved for run-time use

The resource flag word is defined as shown in Figure K-3.

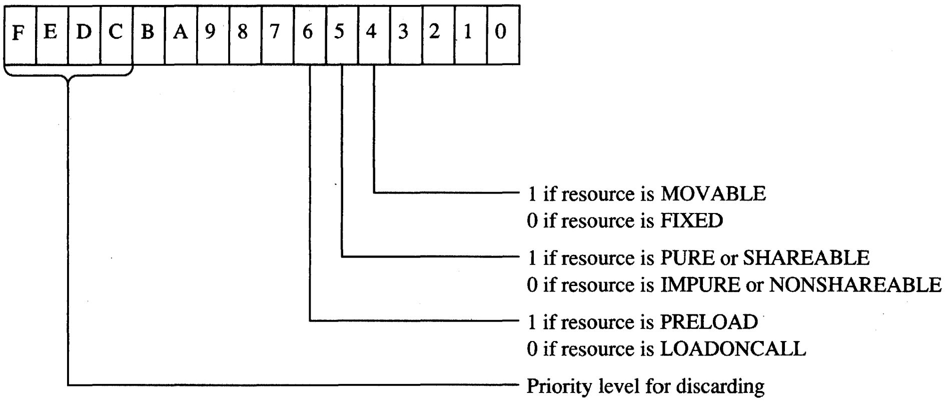

F
E
D
C
B
A
9
8
7
6
5
4
3
2
1
0
1 if resource is MOVABLE
0 if resource is FIXED
1 if resource is PURE or SHAREABLE
0 if resource is IMPURE or NONSHAREABLE
1 if resource is PRELOAD
0 if resource is LOADONCALL
Priority level for discarding

Figure K-3. The resource flag word.

The resident names table

This table contains a list of ASCII strings. The first string is the module name given in the module definition file. The other strings are the names of all exported functions listed in the module definition file that were not given explicit ordinal numbers or that were explicitly specified in the file as resident names. (Exported functions with explicit ordinal numbers in the module definition file are listed in the nonresident names table.)

Each string is prefaced by a single byte indicating the number of characters in the string and is followed by a word (2 bytes) referencing an element in the entry table, beginning at 1. The word that follows the module name is 0. (Offsets are from the beginning of the record.)

Offset	Length (bytes)	Contents
00H	1	Number of bytes in string (0 if end of table)
01H	*n*	ASCII string, not null-terminated
n+1	2	Index into entry table

The module reference table

The module reference table contains 2 bytes for every external module the program uses. These 2 bytes are an offset into the imported names table.

The imported names table

The imported names table contains a list of ASCII strings. These strings are the names of all other modules that are referenced through imported functions. The strings are prefaced with a single byte indicating the length of the string.

For most Windows programs, the imported names table includes KERNEL, USER, and GDI, but it can also include names of other modules, such as KEYBOARD and SOUND. (Offsets are from the beginning of the record.)

Offset	Length (bytes)	Contents
00H	1	Number of bytes in name string
01H	*n*	ASCII name string, not null-terminated

These strings do not necessarily start at the beginning of the imported names table; the names are referenced by offsets specified in the module reference table.

The entry table

This table contains one member for every entry point in the program or library module. (Every public FAR function or procedure in a module is an entry point.) The members in the entry table have ordinal numbers beginning at 1. These ordinal numbers are referenced by the resident names table and the nonresident names table.

LINK versions 4.0 and later bundle the members of the entry table. Each bundle begins with the following information. (Offsets are from the beginning of the bundle.)

Offset	Length (bytes)	Contents
00H	1	Number of entry points in bundle (0 if end of table)
01H	1	Segment number of entry points if entry points in bundle are in single fixed segment; 0FFH if entry points in bundle are in movable segments

For a bundle containing entry points in fixed segments, each entry point requires 3 bytes. (Offsets are from the beginning of the entry description.)

Offset	Length (bytes)	Contents
00H	1	Entry-point flag byte (*see* below)
01H	2	Offset of entry point in segment

For bundles containing entry points in movable segments, each entry point requires 6 bytes. (Offsets are from the beginning of the entry description.)

Offset	Length (bytes)	Contents
00H	1	Entry-point flag byte (*see* below)
01H	2	Interrupt 3FH instruction: CDH 3FH
03H	1	Segment number of entry point
04H	2	Offset of entry-point segment

The entry-point flag byte is defined as shown in Figure K-4.

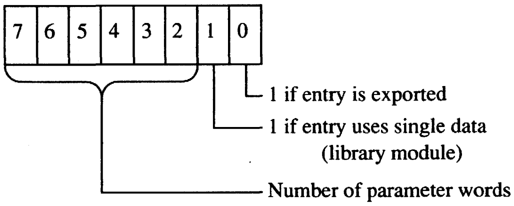

7
6
5
4
3
2
1
0
1 if entry is exported
1 if entry uses single data (library module)
Number of parameter words

Figure K-4. The entry-point flag.

The nonresident names table

This table contains a list of ASCII strings. The first string is the module description from the module definition file. The other strings are the names of all exported functions listed in the module definition file that have ordinal numbers associated with them. (Exported functions without ordinal numbers in the module definition file are listed in the resident names table.)

Each string is prefaced by a single byte indicating the number of characters in the string and is followed by a word (2 bytes) referencing a member of the entry table, beginning at 1. The word that follows the module description string is 0. (Offsets are from the beginning of the table.)

Offset	Length (bytes)	Contents
00H	1	Number of bytes in string (0 if end of table)
01H	*n*	ASCII string, not null-terminated
n+1	2	Index into entry table

The code and data segment

Following the various tables in the New Executable file header are the code and data segments of the program or library module.

If the code or data segment is flagged in the segment flag word as ITERATED, the segment is organized as follows. (Offsets are from the beginning of the segment.)

Offset	Length (bytes)	Contents
00H	2	Number of iterations of data
02H	2	Number of bytes of data
04H	*n*	Data

Otherwise, the size of the segment data is given by the length of the segment field in the segment table.

If the segment is flagged in the segment flag word as containing relocation information, then the relocation table begins immediately after the segment data. Windows uses the relocation table to resolve references within the segments to functions in other segments in the same module and to imported functions in other modules. (Offsets are from the beginning of the table.)

Offset	Length (bytes)	Contents
00H	2	Number of relocation items

Each relocation item requires 8 bytes. (Offsets are from the beginning of the relocation item.)

Offset	Length (bytes)	Contents
00H	1	Type of address to insert in segment: 01H Offset only 02H Segment only 03H Segment and offset

(more)

Offset	Length (bytes)	Contents
01H	1	Relocation type: 00H Internal reference 01H Imported ordinal 02H Imported name If bit 2 set, relocation type is additive (*see* below)
02H	2	Offset of relocation item within segment

The next 4 bytes depend on the relocation type. If the relocation type is an internal reference to a segment in the same module, these bytes are defined as follows. (Offsets are from the beginning of the relocation item.)

Offset	Length (bytes)	Contents
04H	1	Segment number for fixed segment; 0FFH for movable segment
05H	1	0
06H	2	If MOVABLE segment, ordinal number referenced in entry table; if FIXED segment, offset into segment

If the relocation type is an imported ordinal to another module, then these bytes are defined as follows. (Offsets are from the beginning of the relocation item.)

Offset	Length (bytes)	Contents
04H	2	Index into module reference table
06H	2	Function ordinal number

Finally, if the relocation type is an imported name of a function in another module, these bytes are defined as follows. (Offsets are from the beginning of the relocation item.)

Offset	Length (bytes)	Contents
04H	2	Index into module reference table
06H	2	Offset within imported names table to name of imported function

If the ADDITIVE flag of the relocation type is set, the address of the external function is added to the contents of the address in the target segment. If the ADDITIVE flag is not set, then the target contains an offset to another target within the same segment that requires the same relocation address. This defines a chain of target addresses that get the same address. The chain is terminated with a –1 entry.

Charles Petzold

Appendix L
Intel Hexadecimal Object File Format

The MCS-86 hexadecimal object file format provides a means of recording a program's binary (compiled or assembled) image in a text-only (printable) file format. This format makes it easy to transfer the program between computers over telephone lines without using special communications software. More important, it provides a ready means of transferring programs between computers and the various types of laboratory equipment typically used during the development of specialized programs.

The MCS-86 hexadecimal file format is a superset of Intel's older Intellec-8 hexadecimal object file format. Intel originally designed the Intellec-8 format for use with its 8-bit microprocessor line. The format rapidly gained acceptance among other microprocessor manufacturers. When Intel subsequently developed the MCS-86 microprocessor family, it also expanded the Intellec-8 hexadecimal file format into the MCS-86 hexadecimal file format to support the new microprocessors' extended addressing capabilities.

The MCS-86 hexadecimal object file format should not be confused with the object (.OBJ) files produced by the Microsoft Macro Assembler (MASM) and language compilers. The MCS-86 hexadecimal object file format is referred to as an *absolute* object file format because the code contained within the file has been completely linked and all address references have already been resolved. The object modules produced by the assembler and compilers (.OBJ files) are referred to as *relocatable* object modules because they contain the information necessary to relocate the enclosed code to any memory address for execution.

The MCS-86 hexadecimal object file format consists of four types of ASCII text records:

- Data record
- End-of-file record
- Extended-address record
- Start-address record

All records begin with a *record mark* consisting of a single ASCII colon character (:). The remainder of the record consists of a variable number of ASCII hexadecimal digit pairs (00–0FH), each representing an unsigned byte value (0–255 decimal). The first digit represents the value of the high nibble (bits 7–4) of the byte; the second digit represents the value of the low nibble (bits 3–0). These digit pairs begin immediately after the record mark and continue through the end of the record without any separation between them.

All records have the following fields, in the order listed:

- A fixed-length *record length* field
- A fixed-length *address* field (optional)
- A fixed-length *record type* field

- A fixed-length or variable-length *data* field
- A fixed-length *checksum* field

The fixed-length *record length* field consists of the first digit pair following the record mark and gives the length of the record-type-dependent variable-length data field.

The optional fixed-length *address* field consists of the second and third digit pairs following the record mark. The first digit pair of this field (second digit pair of the record) gives the high byte of a word address value (bits 15–8); the second digit pair (third digit pair of the record) gives the low byte of a word address value (bits 7–0). If the record type does not use the address field, then the field contains a fill-in value consisting of the four-character ASCII string *0000.*

The fixed-length *record type* field consists of the fourth digit pair of the record and indicates the type of data the record contains. The valid record-type values are

Value	Type
00H	Data record
01H	End-of-file record
02H	Extended-address record
03H	Start-address record

All records end with a fixed-length *checksum* field. This field contains the negative of the sum of all byte values represented by the digit pairs in the record, from the record length field through the last digit pair before the checksum field. The checksum field is used to determine whether an error occurred during the transmission of a record between computers or other pieces of equipment.

(The receiving equipment can easily perform this error checking as each record is received. It only has to add all digit pairs of the record, including the checksum, and ignore any overflow beyond 8 bits. The total should be 00H, because the checksum is the negative of the summation of all preceding digit pairs.)

The variable-length *data* field of the data record contains the actual data bytes of the program's image. In data records, the record length field indicates the number of bytes, each represented as a digit pair, contained within the data field; the address field gives the offset within the current memory segment at which to load the record's data into memory.

The fixed-length data field of the extended-address record establishes the memory segment into which subsequent data records are to be loaded. In extended-address records, the data field consists of a single field identical to the address field. The address field of an extended-address record always contains the ASCII 0000 filler, and the record length field always contains ASCII 02, which reflects the fixed length of the data field. The memory segment (also known as the memory frame) established by an extended-address record remains in effect until the next extended-address record is encountered; thus, all data records following the most recent extended-address record are loaded in the established memory segment. *See* PROGRAMMING IN MS-DOS: PROGRAMMING TOOLS: The Microsoft Object Linker.

Figures L-1 and L-2 show how the extended-address record and the data record combine to load the byte values 0FDH, 0B9H, 75H, 31H, 0ECH, 0A8H, 64H, and 20H into memory starting at address 9A6EH:429FH.

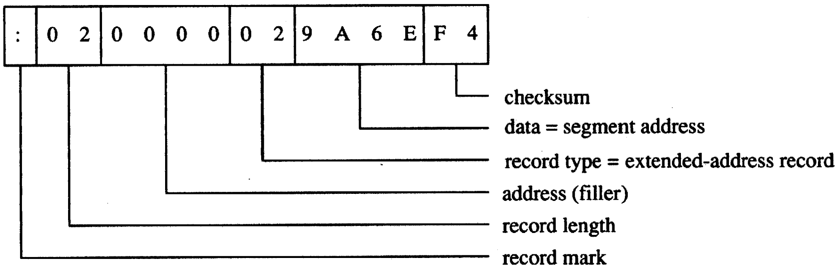


Figure L-1. The extended-address record.

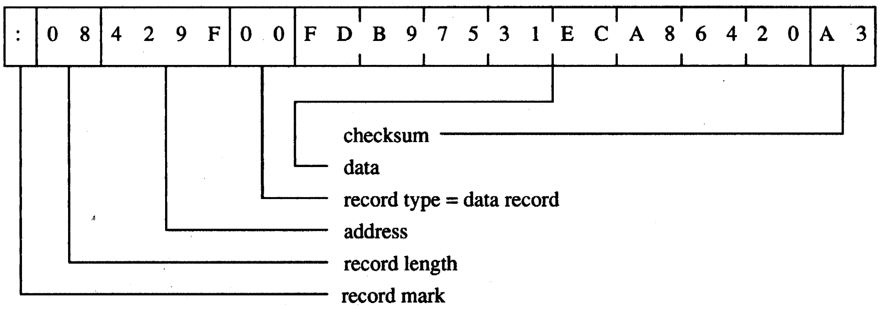


Figure L-2. The data record.

The start-address record provides the CS and IP register values at which program execution begins. This record contains the register values within the fixed-length data field. The address field of a start-address record always contains the ASCII 0000 filler, and the record length field always contains ASCII 04, which reflects the fixed length of the data field. The example in Figure L-3 shows a CS:IP setting (program entry point) of F924H:E69AH.

The end-of-file record marks the end of an MCS-86 hexadecimal file. Under the MCS-86 hexadecimal file definition, the end-of-file record does not contain any variable-value fields; the record always appears as shown in Figure L-4.

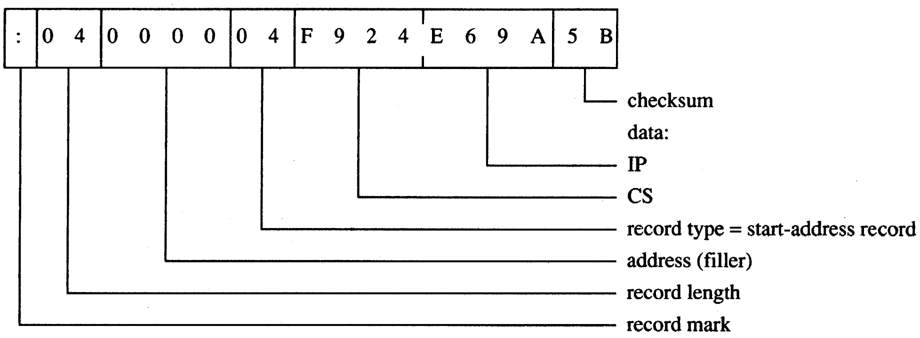

: 0 4 0 0 0 0 0 4 F 9 2 4 E 6 9 A 5 B
checksum
data:
IP
CS
record type = start-address record
address (filler)
record length
record mark

Figure L-3. The start-address record.

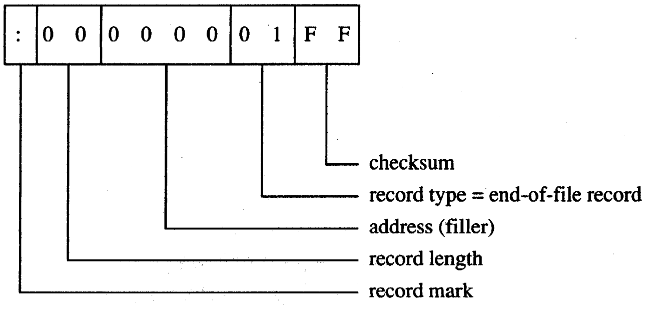


Figure L-4. The end-of-file record.

Traditionally, development equipment and programs that accept the MCS-86 hexadecimal file format as input also recognize an alternate end-of-file record. The alternate record consists of a data record that contains no data; therefore, its record length field contains 00. Figure L-5 shows this alternate end-of-file record.

DEBUG is the only program supplied with MS-DOS that accepts the MCS-86 hexadecimal file format. Even then, DEBUG only loads hexadecimal files into memory; it does not save a program back to disk as a hexadecimal file. (The same applies for SYMDEB and for CodeView.)

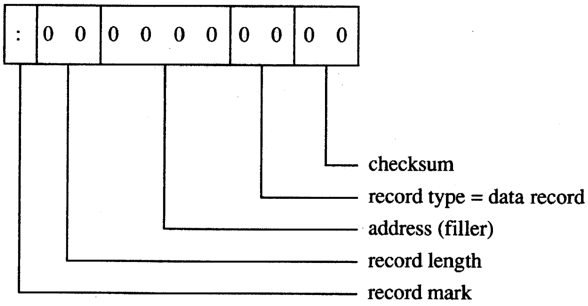


Figure L-5. The alternate end-of-file record.

While loading a hexadecimal file, DEBUG actually processes only data records and end-of-file records; it ignores both start-address records and any extended-address records. Thus, DEBUG actually supports only the older Intellec-8 hexadecimal file format but will not reject the file if it also contains the newer MCS-86 hexadecimal file records.

DEBUG does not support MCS-86 records because it must operate within the MS-DOS environment and MS-DOS does not support the loading of programs into absolute memory locations — a restriction imposed by most general-purpose operating systems. Because DEBUG cannot load the data records into the absolute segments indicated by the extended-address records, it simply loads the program image contained within the data records in a manner similar to that in which a .COM program is loaded. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: Structure of an Application Program. DEBUG uses the address field for the data records as the offset into the .COM program segment at which to load the contents of the records.

The sample QuickBASIC (versions 3.0 and later) program shown in Figure L-6 converts binary files, including .COM files, into limited MCS-86 hexadecimal files that DEBUG can load. Examining this program can provide additional understanding of the structure of Intel hexadecimal files.

```
'Binary-to-Hex file conversion utility.
'Requires Microsoft QuickBASIC version 3.0 or later.

DEFINT A-Z                                      ' All variables are integers
                                                ' unless otherwise declared.
CONST FALSE = 0                                 ' Value of logical FALSE.
CONST TRUE = NOT FALSE                          ' Value of logical TRUE.

DEF FNHXB$(X) = RIGHT$(HEX$(&H100 + X), 2)      ' Return 2-digit hex value for X.
DEF FNHXW$(X!) = RIGHT$("000" + HEX$(X!), 4)    ' Return 4-digit hex value for X!.
DEF FNMOD(X, Y) = X! - INT(X!/Y) * Y            ' X! MOD Y (the MOD operation is
                                                ' only for integers).
CONST SRCCNL = 1                                ' Source (.BIN) file channel.
CONST TGTCNL = 2                                ' Target (.HEX) file channel.

LINE INPUT "Enter full name of source .BIN file        :  ";SRCFIL$
OPEN SCRCFIL$ FOR INPUT AS SRCCNL               ' Test for source (.BIN) file.
SRCSIZ! = LOF(SRCCNL)                           ' Save file's size.
CLOSE SRCCNL
IF (SRCSIZ! > 65536) THEN                       ' Reject if file exceeds 64 KB.
    PRINT "Cannot convert file larger than 64 KB."
    END
END IF

LINE INPUT "Enter full name of target .HEX file        :  ";TGTFIL$
OPEN TGTFIL$ FOR OUTPUT AS TGTCNL               ' Test target (.HEX) filename.
CLOSE TGTCNL
```

Figure L-6. QuickBASIC binary-to-hexadecimal file conversion utility. *(more)*

```
DO
    LINE INPUT "Enter starting address of .BIN file in HEX :  ";L$
    ADRBGN! = VAL("&H" + L$)                  ' Convert ASCII HEX address value
                                              ' to binary value.
    IF (ADRBGN! < 0) THEN                     ' HEX values 8000-FFFFH convert
     ADRBGN! = 65536 + ADRBGN!                ' to negative values.
    END IF
    ADREND! = ADRBGN! + SRCSIZ! - 1           ' Calculate resulting end address.
    IF (ADREND! > 65535) THEN                 ' Reject if address exceeds FFFFH.
     PRINT "Entered start address causes end address to exceed FFFFH."
    END IF
LOOP UNTIL (ADRFLD! >= 0) AND (ADRFLD! <= 65535) AND (ADREND! <= 65535)

DO
    LINE INPUT "Enter byte count for each record in HEX    :  ";L$
    SRCRLN = VAL("&H" + L$)                   ' Convert ASCII HEX max record
                                              ' length value to binary value.
    IF (SRCRLN < 0) THEN                      ' HEX values 8000-FFFFH convert
     SRCRLN = 65536 + SRCRLN                  ' to negative values.
    END IF
LOOP UNTIL (SRCRLN > 0) AND (SRCRLN < 256)    ' Ask again if not 1-255.

OPEN SRCFIL$ AS SRCCNL LEN = SRCRLN           ' Reopen source for block I/O.
FIELD#SRCCNL,SRCRLN AS SRCBLK$
OPEN TGTFIL$ FOR OUTPUT AS TGTCNL             ' Reopen target for text output.
SRCREC = 0                                    ' Starting source block # minus 1.

FOR ADRFLD! = ADRBGN! TO ADREND! STEP SRCRLN  ' Convert one block per loop.
    SRCREC = SRCREC + 1                       ' Next source block.
    GET SRCCNL,SRCREC                         ' Read the source block.
    IF (ADRFLD! + SRCRLN > ADREND!) THEN      ' If last block less than full
     BLK$=LEFT$(SRCBLK$,ADREND!-ADRFLD!+1)    ' size:  trim it.
    ELSE                                      ' Else:
        BLK$ = SRCBLK$                        ' Use full block.
    END IF

    PRINT#TGTCNL, ":";                        ' Write record mark.

    PRINT#TGTCNL, FNHXB$(LEN(BLK$));          ' Write data field size.
    CHKSUM = LEN(BLK$)                        ' Initialize checksum accumulate
                                              ' with first value.
    PRINT#TGTCNL,FNHXW$(ADRFLD!);             ' Write record's load address.

' The following "AND &HFF" operations limit CHKSUM to a byte value.
    CHKSUM = CHKSUM + INT(ADRFLD!/256) AND &HFF    ' Add hi byte of adrs to csum.
    CHKSUM = CHKSUM + FNMOD(ADRFLD!,256) AND &HFF  ' Add lo byte of adrs to csum.

    PRINT#TGTCNL,FNHXB$(0);                   ' Write record type.
```

Figure L-6. Continued.

(more)

```
' Don't bother to add record type byte to checksum since it's 0.
    FOR IDX = 1 TO LEN(BLK$)                   ' Write all bytes.
     PRINT#TGTCNL,FNHXB$(ASC(MID$(BLK$,IDX,1)));        ' Write next byte.
     CHKSUM = CHKSUM + ASC(MID$(BLK$,IDX,1)) AND &HFF ' Incl byte in csum.
    NEXT IDX

    CHKSUM = 0 - CHKSUM AND &HFF               ' Negate checksum then limit
                                               ' to byte value.
    PRINT #TGTCNL,FNHXB$(CHKSUM)               ' End record with checksum.

NEXT ADRFLD!

PRINT#TGTCNL, ":00000001FF"                    ' Write end-of-file record.

CLOSE TGTCNL                                   ' Close target file.
CLOSE SRCCNL                                   ' Close source file.

END
```

Figure L-6. Continued.

Keith Burgoyne

Appendix M
8086/8088 Software Compatibility Issues

In general, the Intel 80286 microprocessor running in real mode executes 8086/8088 software correctly. The following is a list of the actions to take to compensate for the minor differences between the 8086/8088 and real mode of the 80286.

- *Do not rely on 8086/8088 instruction clock counts.* The 80286 takes fewer clocks for most instructions than the 8086/8088. The areas to look into are delays between I/O operations and assumed delays when the 8086/8088 is operating in parallel with an 8087 coprocessor.
- *Note that divide exceptions point to the DIV instruction.* Any interrupt on the 80286 always leaves the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS:IP value saved for a divide exception points to the next instruction.
- *Set up numeric exception handlers to allow prefixes.* The saved CS:IP value in the NPX environment save area points to any ESC instruction prefixes. On 8086/8088 systems, this value points only to the ESC instruction.
- *Do not attempt undefined 8086/8088 operations.* 8086/8088 instructions like POP CS or MOV CS,op either invoke exception 06H (Invalid Opcode) or perform a protection setup operation like LIDT on the 80286. Undefined bit encodings for bits 5–3 of the second byte of POP MEM or PUSH MEM invoke exception 13H on the 80286.
- *Do not rely on the value written by PUSH SP.* The 80286 pushes a different value on the stack for PUSH SP than does the 8086/8088. If the value pushed is important, replace PUSH SP instructions with the following instructions:

```
PUSH        BP
MOV         BP,SP
XCHG        BP,[BP]
```

  This code functions like the 8086/8088 PUSH SP instruction on the 80286.
- *Do not shift or rotate by more than 31 bits.* The 80286 masks all SHIFT/ROTATE counts to the low 5 bits. This MOD 32 operation limits the count to a maximum of 31 bits. With this change, the longest SHIFT/ROTATE instruction is 39 clocks. Without this change, the longest SHIFT/ROTATE instruction is 264 clocks, which delays interrupt response until the instruction completes execution.
- *Do not duplicate prefixes.* The 80286 sets an instruction-length limit of 10 bytes. The only way to exceed this limit is to include the same prefix two or more times before an instruction. Exception 06H occurs if the instruction-length limit is violated. The 8086/8088 has no instruction-length limit.
- *Do not rely on odd 8086/8088 LOCK characteristics.* The LOCK prefix and its corresponding output signal should be used only to prevent other bus masters from interrupting a data movement operation. The 80286 always asserts LOCK during an XCHG instruction with memory (even if the LOCK prefix was not used). LOCK should be

used only with the XCHG, MOV, MOVS, INS, and OUTS instructions. The 80286 LOCK signal will *not* go active during an instruction prefetch.

- *Do not rely on IDIV exceptions for quotients of 80H or 8000H.* The 80286 can generate the largest negative number as a quotient for IDIV instructions. The 8086/8088 generates exception 00H (Divide by Zero) instead.
- *Do not rely on address space wraparound.*
- *Do not use I/O ports 0F8–0FFH.* These are reserved for controlling the 80287 and future microprocessor extensions.

Appendix N
An Object Module Dump Utility

The program OBJDUMP.C displays the contents of an object file as individual object records. It can be used to study the structure of object modules as well as to verify the output of a language translator. The program recognizes all of the object record types discussed in PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Object Modules.

OBJDUMP.C should be executed with the following syntax:

OBJDUMP *filename*

where *filename* is a complete filename specification. For example, to dump the contents of the object file MYPROG.OBJ, the user would type

```
C>OBJDUMP MYPROG.OBJ  <Enter>
```

The following is a typical object record as displayed by OBJDUMP:

```
Record 9:  96h LNAMES
96 002Eh 00 06 44 47 52 4F 55 50 05 5F 54 45 58 54 04 43   ..DGROUP._TEXT.C
         4F 44 45 05 5F 44 41 54 41 04 44 41 54 41 05 43   ODE._DATA.DATA.C
         4F 4E 53 54 04 5F 42 53 53 03 42 53 53 3F         ONST._BSS.BSS?
```

This sample LNAMES record defines a null name and eight names used in subsequent SEGDEF and GRPDEF records. The first 3 bytes of the record (the identifying byte and the 2-byte record length) are displayed to the left of the hexadecimal and ASCII listings of the contents of the record.

```
/******************************************************************************
*                                                                             *
* OBJDUMP.C -- display contents of an object file                             *
*                                                                             *
*                                                                             *
*     Compile:  msc objdump;   (Microsoft C version 4.0 or later)             *
*     Link:     link objdump;                                                 *
*     Execute:  objdump <filename>                                            *
*                                                                             *
******************************************************************************/

#include        <fcntl.h>

#define         TRUE    1
#define         FALSE   0
```

(more)

```
main( argc, argv )
int         argc;
char        **argv;
{
        unsigned char         CurrentByte;
        int     ObjFileHandle;
        int     CurrentLineLength;               /* length of output line */
        int     ObjRecordNumber = 0;
        int     ObjRecordLength;
        int     ObjRecordOffset = 0;   /* offset into current object record */
        char    ASCIIEquiv[17];
        char    FormatString[24];
        char    *ObjRecordName();
        char    *memset();


/* open the object file */

        ObjFileHandle = open( argv[1],O_BINARY );

        if( ObjFileHandle == -1 )
        {
          printf( "\nCan't open object file\n" );
          exit( 1 );
        }

/* process the object file character by character */

        while( read( ObjFileHandle, &CurrentByte, 1 ) )
        {
          switch( ObjRecordOffset ) /* action depends on offset into record */
          {
            case(0):                              /* start of object record */
              printf( "\n\nRecord %d:  %02Xh %s",
                ++ObjRecordNumber, CurrentByte, ObjRecordName(CurrentByte) );
              printf( "\n%02X ", CurrentByte );
              ++ObjRecordOffset;
              break;

            case(1):                           /* first byte of length field */
              ObjRecordLength = CurrentByte;
              ++ObjRecordOffset;
              break;

            case(2):                          /* second byte of length field */
              ObjRecordLength += CurrentByte << 8; /* compute record length */
              printf( "%04Xh ", ObjRecordLength );            /* show length */
              CurrentLineLength = 0;
              memset( ASCIIEquiv, '\0' , 17 );           /* zero this string */
              ++ObjRecordOffset;
              break;
```

(more)

```
            default:                    /* remaining bytes in object record */
              printf( "%02X ", CurrentByte );                       /* hex */

              if( CurrentByte < 0x20 ¦¦ CurrentByte > 0x7F )      /* ASCII */
                CurrentByte = '.';
              ASCIIEquiv[CurrentLineLength++] = CurrentByte;

              if( CurrentLineLength == 16 ¦¦   /* if end of output line ... */
                 ObjRecordOffset == ObjRecordLength+2 )
              {                                          /* ... display it */
                sprintf( FormatString, "%%%ds%%s\n         ",
                  3*(16-CurrentLineLength)+2 );
                printf( FormatString, " ", ASCIIEquiv );
                memset( ASCIIEquiv, '\0', 17 );
                CurrentLineLength = 0;
              }

              if( ++ObjRecordOffset == ObjRecordLength+3 )   /* if done ... */
                ObjRecordOffset = 0;          /* ... process another record */
              break;
          }
        }

        if( CurrentLineLength )    /* display remainder of last output line */
          printf( "  %s", ASCIIEquiv );

        close( ObjFileHandle );

        printf( "\n%d object records\n", ObjRecordNumber );

        return( 0 );
}


char *ObjRecordName( n )                       /* return object record name */
int        n;                                            /* n = record type */
{
        int        i;

        static     struct
        {
          int        RecordNumber;
          char       *RecordName;
        }                RecordStruct[] =
                         {
                          0x80,"THEADR",
                          0x88,"COMENT",
                          0x8A,"MODEND",
                          0x8C,"EXTDEF",
                          0x8E,"TYPDEF",
                          0x90,"PUBDEF",
```

(more)

```
                    0x94,"LINNUM",
                    0x96,"LNAMES",
                    0x98,"SEGDEF",
                    0x9A,"GRPDEF",
                    0x9C,"FIXUPP",
                    0xA0,"LEDATA",
                    0xA2,"LIDATA",
                    0xB0,"COMDEF",
                    0x00,"******"
                  };

        int      RecordTableSize = sizeof(RecordStruct)/sizeof(RecordStruct[0]);


        for( i=0; i<RecordTableSize-1; i++ )         /* scan table for name */
          if ( RecordStruct[i].RecordNumber == n )
            break;

        return( RecordStruct[i].RecordName );
}
```

Richard Wilton

Appendix O
IBM PC ROM BIOS Calls

To invoke an IBM PC BIOS routine, set register AH to the desired function and execute the software interrupt (INT) for the desired routine.

Graphics pixel coordinates and cursor row and column coordinates are always zero based.

Interrupt 10H: Video Services

Function 00H: Set Video Mode

To call:

AH = 00H

AL = mode:

00H	16-shade gray text EGA: 64-color	40 by 25	B000:8000H
01H	16/8-color text EGA: 64-color	40 by 25	B000:8000H
02H	16-shade gray text EGA: 64-color	80 by 25	B000:8000H
03H	16/8-color text EGA: 64-color	80 by 25	B000:8000H
04H	4-color graphics	320 by 200	B000:8000H
05H	4-shade gray graphics	320 by 200	B000:8000H
06H	2-shade gray graphics	640 by 200	B000:8000H
07H	monochrome text	80 by 25	B000:0000H
08H	16-color graphics	160 by 200	B000:0000H
09H	16-color graphics	320 by 200	B000:0000H
0AH	4-color graphics	640 by 200	B000:0000H
0BH	Reserved		
0CH	Reserved		
0DH	16-color graphics	320 by 200	A000:0000H
0EH	16-color graphics	640 by 200	A000:0000H
0FH	monochrome graphics	640 by 350	A000:0000H
10H	16/64-color graphics	640 by 350	A000:0000H

Returns:

Nothing

Function 01H: Set Cursor Size and Shape

To call:

AH	= 01H
CH	= starting scan line
CL	= ending scan line

Note: CH < CL gives normal one-part cursor; CH > CL gives two-part cursor; CH = 20H gives no cursor.

Returns:

Nothing

Function 02H: Set Cursor Position

To call:

AH	= 02H
BH	= display page (0 in graphics)
DH	= row number
DL	= line number

Returns:

Nothing

Function 03H: Read Cursor Position, Size, and Shape

To call:

AH	= 03H
BH	= display page

Returns:

CH	= starting scan line
CL	= ending scan line
DH	= row number
DL	= column number

Function 04H: Read Light-Pen Position

To call:

AH	= 04H

Returns:

AH	= status:	
	01H	pen triggered
	00H	not triggered
BX	= pixel column number	
CH	= pixel line number	
CX	= pixel line number for some EGA modes	
DH	= character row number	
DL	= character column number	

Function 05H: Select Active Page

To call:

AH	= 05H	
AL	= page number:	
	00–07H	40-column text modes
	00–03H	80-column text modes
	varies	EGA graphics modes

Note: Each page = 2 KB in 40-column text mode, 4 KB in 80-column text mode.

Returns:

Nothing

Function 06H: Scroll Window Up
Function 07H: Scroll Window Down

To call:

AH	= 06H	scroll up
	= 07H	scroll down
AL	= number of lines to scroll (00H blanks screen)	
BH	= display attributes for blank lines	
CH	= row number of upper left corner	
CL	= column number of upper left corner	
DH	= row number of lower right corner	
DL	= column number of lower right corner	

Returns:

Nothing

Function 08H: Read Character and Attribute at Cursor

To call:

AH	= 08H
BH	= display page (for text mode only)

Returns:

If text mode:

AH	= color attributes of character
AL	= ASCII character from current location

If graphics mode:

AL	= ASCII character (00H if unmatched)

Function 09H: Write Character and Attribute

To call:

AH	= 09H
AL	= ASCII character to write
BH	= display page
BL	= text attribute or graphics foreground color
CX	= number of times to write character (must be > 0)

Returns:

Nothing

Note: Cursor position unchanged.

Function 0AH: Write Character Only

To call:

AH	= 0AH
AL	= ASCII character to write
BH	= display page
BL	= graphics foreground color (unused in text modes)
CX	= number of times to write character (must be > 0)

Returns:

Nothing

Note: Cursor position unchanged.

Function 0BH: Select Color Palette

To call:

AH	= 0BH
BH	= palette color ID
BL	= color or palette value

Returns:

Nothing

Function 0CH: Write Pixel Dot

To call:

AH	= 0CH
AL	= color attribute of pixel
CX	= pixel column number
DX	= pixel raster line number

Returns:

Nothing

Function 0DH: Read Pixel Dot

To call:

AH	= 0DH
CX	= pixel column number (0-based)
DX	= pixel raster line number (0-based)

Returns:

AL	= pixel color attribute

Function 0EH: Write Character as TTY

To call:

AH	= 0EH
AL	= ASCII character
BH	= display page
BL	= foreground color of character (unused in text mode)

Returns:

Nothing

Note: Cursor position advanced; beep, backspace, linefeed, and carriage return active; all other characters displayed.

Function 0FH: Get Current Video Mode

To call:

AH	= 0FH

Returns:

AH	= characters per line (20, 40, or 80)
AL	= current video mode (*see* Interrupt 10H Function 00H)
BH	= active display page

Function 13H: Write Character String

To call:

AH	= 13H	
AL	= subfunction number:	
	00H	string shares attribute in BL, cursor unchanged
	01H	string shares attribute in BL, cursor advanced
	02H	each character has attribute, cursor unchanged
	03H	each character has attribute, cursor advanced
BH	= active display page	
BL	= string attribute (for AL = 00H or 01H only)	
CX	= length of character string	
DH	= starting row number	
DL	= starting column number	
ES:BP	= address of string to be displayed	

Note: For AL = 00H or 01H, string = (*char*, *char*, *char*, ...). For AL = 02H or 03H, string = (*char*, *attr*, *char*, *attr*, ...).

Returns:

Nothing

Note: For AL = 01H or 03H, cursor position set to location following last character output.

Interrupt 11H: Get Peripheral Equipment List

Returns:

AX	= equipment list code word (bit settings PPMURRRUFFVVUUCI):	
	PP	number of printers installed
	M	1 if internal modem installed
	RRR	number of RS-232 ports installed
	U	unused
	FF	number of floppy-disk drives minus 1 (0 = one drive)
	VV	initial video mode:
		00 = reserved
		01 = 40-by-25 color
		10 = 80-by-25 color
		11 = 80-by-25 monochrome
	U	unused
	C	1 if math coprocessor installed
	I	1 if IPL (Initial Program Load) diskette installed

Interrupt 12H: Get Usable Memory Size (KB)

Returns:

AX	= available memory size in KB

Interrupt 13H: Disk Services

Function 00H: Reset Disk System

To call:

AH	= 00H	
AL	= drive number:	
	00–7FH	floppy disk
	80–FFH	fixed disk

Returns:

CF	= 0	no error
	1	error
AH	= error code (*see* Interrupt 13H Function 01H)	

Function 01H: Get Disk Status

To call:

AH	= 01H

Returns:

AH	= 00H	
AL	= disk status of previous disk operation:	
	00H	no error
	01H	invalid command
	02H	address mark not found
	03H	write attempt on write-protected disk (F)
	04H	sector not found
	05H	reset failed (H)
	06H	floppy disk removed (F)
	07H	bad parameter table (H)
	08H	DMA overflow (F)
	09H	DMA crossed 64 KB boundary
	0AH	bad sector flag (H)
	10H	uncorrectable CRC or ECC data error
	11H	ECC corrected data error (H)
	20H	controller failed

(more)

40H	seek failed
80H	time out
AAH	drive not ready (H)
BBH	undefined error (H)
CCH	write fault (H)
E0H	status error (H)

Note: H = fixed disk only, F = floppy disk only.

Function 02H: Read Disk Sectors
Function 03H: Write Disk Sectors
Function 04H: Verify Disk Sectors
Function 05H: Format Disk Tracks

To call:

AH	= 02H	read disk sectors
	03H	write disk sectors
	04H	verify disk sectors
	05H	format disk track
AL	= number of sectors	
CH	= cylinder number	
CL	= sector number (unused if AH = 05H)	
DH	= head number	
DL	= drive number	
ES:BX	= buffer address (unused if AH = 04H)	

Returns:

CF	= 0	no error
	1	error
AH	= error code (*see* Interrupt 13H Function 01H)	

If AH was 05H on call:

ES:BX	= 4-byte address field entries, 1 per sector:		
	byte 0	cylinder number	
	byte 1	head number	
	byte 2	sector number	
	byte 3	sector-size code:	
		00H	128 bytes per sector
		01H	256 bytes per sector
		02H	512 bytes per sector (standard)
		03H	1024 bytes per sector

Function 08H: Get Current Drive Parameters

To call:

AH	= 08H
DL	= drive number

Returns:

AX	= 00H	
BH	= 00H	
BL	= drive type	
CH	= low-order 8 bits of 10-bit maximum number of cylinders	
CL	= bits 7 and 6	high-order 2 bits of 10-bit maximum number of cylinders
	bits 5–0	maximum number of sectors/track
DH	= maximum head number	
DL	= number of drives installed	
ES:DI	= address of floppy-disk-drive parameter table	

Function 09H: Initialize Hard-Disk Parameter Table

To call:

AH	= 09H

Returns:

Nothing

Function 0AH: Read Long

Reads 512-byte sector plus 4-byte ECC code.

To call:

See Interrupt 13H Function 02H.

Returns:

See Interrupt 13H Function 02H.

Function 0BH: Write Long

Writes 512-byte sector plus 4-byte ECC code.

To call:

See Interrupt 13H Function 03H.

Returns:

See Interrupt 13H Function 03H.

Function 0CH: Seek to Head

Positions head but does not transfer data.

To call:

See Interrupt 13H Functions 02H and 03H.

Returns:

See Interrupt 13H Functions 02H and 03H.

Function 0DH: Alternate Disk Reset

To call:

AH	= 0DH
DL	= drive number

Returns:

Nothing

Function 10H: Test for Drive Ready

To call:

AH	= 10H
DL	= drive number

Returns:

AH	= status

Function 11H: Recalibrate Drive

To call:

AH	= 11H
DL	= drive number

Returns:

AH	= status

Function 14H: Controller Diagnostic

To call:

AH	= 14H

Returns:

AH	= status

Function 15H: Get Disk Type

To call:

AH	= 15H
DL	= drive number

Returns:

AH	= drive type code:	
	00H	no drive present
	01H	cannot sense when floppy disk is changed

(more)

	02H	can sense when floppy disk is changed
	03H	fixed disk

If AH = 03H:

CX:DX	= number of sectors

Function 16H: Check for Change of Floppy Disk Status

To call:

AH	= 16H
DL	= drive number to check

Returns:

AH	= 00H	no change
	06H	floppy-disk change

Function 17H: Set Disk Type

To call:

AH	= 17H
DL	= drive number
AL	= floppy-disk type code

Returns:

Nothing

Interrupt 14H: Serial Port Services

Function 00H: Initialize Port Parameters

To call:

AH	= 00H		
AL	= serial port parameters (bit settings BBBPPSCC):		
	BBB	baud rate:	
		000	110 baud
		001	150 baud
		010	300 baud
		011	600 baud
		100	1200 baud
		101	2400 baud
		110	4800 baud
		111	9600 baud

(more)

PP parity code:
00 none
01 odd
10 none
11 even
S number of stop bits code:
0 one stop bit
1 two stop bits
CC character size:
00 unused
01 unused
10 7-bit character size
11 8-bit character size

DX = serial port number (0 = first port)

Returns:

Nothing

Function 01H: Send One Character

To call:

AH = 01H
AL = character to send
DX = serial port number (0 = first port)

Returns:

AH = error status (*see* Interrupt 14H Function 03H):
00H no error

Function 02H: Receive One Character

To call:

AH = 02H
DX = serial port number (0 = first port)

Returns:

AL = character received
AH = error status (*see* Interrupt 14H Function 03H):
00H no error

Function 03H: Get Port Status

To call:

AH = 03H
DX = serial port number (0 = first port)

Returns:

AX = serial port status:

8000H	time out
4000H	transfer shift register empty
2000H	transfer holding register empty
1000H	break detect
0800H	framing error
0400H	parity error
0200H	overrun error
0100H	data ready
0080H	received line signal detect
0040H	ring indicator
0020H	data set ready
0010H	clear to send
0008H	delta receive line signal detect
0004H	trailing edge ring detector
0002H	delta data set ready
0001H	delta clear to send

Note: Multiple conditions can be active simultaneously.

Interrupt 15H: Miscellaneous System Services

Function 00H: Turn On Cassette Motor
Function 01H: Turn Off Cassette Motor

To call:

AH	= 00H	turn on cassette motor
	01H	turn off cassette motor

Returns:

Nothing

Function 02H: Read Data from Cassette

To call:

AH	= 02H
CX	= number of bytes to read
ES:BX	= buffer address

Returns:

CF	= 0	no error
	1	error
AH	= error status (if needed):	
	01H	CRC error
	02H	bit signals scrambled
	03H	no data found
DX	= number of bytes read	
ES:BX	= location following last byte read	

Function 03H: Write Data to Cassette

To call:

AH	= 03H
CX	= number of bytes to write
ES:BX	= buffer address

Note: Blocking factor = 256 bytes/block.

Returns:

CX	= 00H
ES:BX	= location following last byte written

Interrupt 16H: Keyboard Services

Function 00H: Read Next Character

To call:

AH	= 00H

Returns:

If ASCII characters:

AH	= standard PC keyboard scan code
AL	= ASCII character

If extended ASCII codes:

AH	= extended ASCII code
AL	= 00H

Note: Does not return until character is read; removes character from keyboard buffer.

Function 01H: Report If Character Ready

To call:

AH	= 01H

Returns:

ZF	= 0	character ready
	1	character not ready
AH	= *see* Interrupt 16H Function 00H	
AL	= *see* Interrupt 16H Function 00H	

Note: Returns immediately; does not remove character from keyboard buffer.

Function 02H: Get Shift Status

To call:

AH	= 02H

Returns:

AL	= shift status:	
	01H	right shift active
	02H	left shift active
	04H	Ctrl active
	08H	Alt active
	10H	Scroll Lock active
	20H	Num Lock active
	40H	Caps Lock active
	80H	insert state active

Note: Multiple states can be active simultaneously.

Interrupt 17H: Printer Services

Function 00H: Send Byte to Printer

To call:

AH	= 00H
AL	= character to be printed
DX	= printer number

Returns:

AH	= status (*see* Interrupt 17H Function 02H)

Function 01H: Initialize Printer

To call:

AH	= 01H
DX	= printer number

Returns:

AH	= status (*see* Interrupt 17H Function 02H)

Function 02H: Get Printer Status

To call:

AH	= 02H
DX	= printer number

Returns:

AH = status:

01H	time out
02H	unused
04H	unused
08H	I/O error
10H	printer selected
20H	out of paper
40H	printer acknowledgment
80H	printer not busy (bit off, 0, = busy)

Note: Multiple states can be active simultaneously.

Interrupt 18H: Transfer Control to ROM-BASIC

Interrupt 19H: Reboot Computer (Warm Start)

Interrupt 1AH: Get/Set Time/Date

Function 00H: Read Current Clock Count

To call:

AH	= 00H

Returns:

AL	= midnight signal
CX	= high-order word of tick count
DX	= low-order word of tick count

Function 01H: Set Current Clock Count

To call:

AH	= 01H
CX	= high-order word of tick count
DX	= low-order word of tick count

Returns:

Nothing

Function 02H: Read Real-Time Clock

To call:

AH	= 02H

Returns:

CF	= 0	clock running
	1	clock stopped
CH	= hours in BCD	
CL	= minutes in BCD	
DH	= seconds in BCD	

Function 03H: Set Real-Time Clock

To call:

AH	= 03H	
CH	= hours in BCD	
CL	= minutes in BCD	
DH	= seconds in BCD	
DL	= 00H	standard time
	01H	daylight saving time

Returns:

Nothing

Function 04H: Read Date from Real-Time Clock

To call:

AH	= 04H

Returns:

CF	= 0	clock running
	1	clock stopped
CH	= century in BCD (19 or 20)	
CL	= year in BCD	
DH	= month in BCD	
DL	= day in BCD	

Function 05H: Set Date in Real-Time Clock

To call:

AH	= 05H
CH	= century in BCD (19 or 20)
CL	= year in BCD
DH	= month in BCD
DL	= day in BCD

Returns:

Nothing

Function 06H: Set Alarm

To call:

AH	= 06H
CH	= hours in BCD
CL	= minutes in BCD
DH	= seconds in BCD

Returns:

CF	= status:	
	0	operation successful
	1	alarm already set or clock stopped

Function 07H: Reset Alarm (Turn Alarm Off)

To call:

AH	= 07H

Returns:

Nothing

Indexes

Subject

Commands and System Calls

This index lists only primary command and system call entries. Please use the Subject Index for related entries.

Book Design by The NBBJ Group, Seattle, Washington

Cover Design by Greg Hickman

Principal Typography by Carol L. Luke

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Text composition by Microsoft Press in Garamond with display in Garamond Bold using the Magna composition system and the Linotronic 300 laser imagesetter.

Special Companion Disk Offer

In addition to the comprehensive technical information presented throughout *The MS-DOS Encyclopedia*, you'll find a wealth of programming examples, handy code fragments, and complete utilities that you'll turn to again and again — literally thousands of lines of code written to make your MS-DOS programming more efficient and more reliable. Included on the companion disks are:

■ a complete serial-communications program ■ two working TSR utilities ■ examples for each of the more than 100 system function calls ■ instructive debugging exercises ■ installable device drivers ■ two complete skeleton filters ■ replacement interrupt handlers ■ hundreds of working code fragments ■ a .OBJ Module Format Utility ■ and much, much more.

Save time, avoid those inevitable typing errors, and start using the code immediately. The disks are available in 5.25" or 3.5" format. To order, fill out the postpaid order card below. If the order card has already been used, refer to the ordering instructions on page xvi.

ORDER CARD

YES. . . please send me *The MS-DOS Encyclopedia* Companion Disks indicated below

_______ set(s) of 5.25" disks at $49.95 per set . . . . $ ____________

_______ set(s) of 3.5" disks at $49.95 per set . . . . $ ____________

Sales Tax (If applicable) . . . . $ ____________
California – 5% (plus local option tax), Connecticut – 7.5%, Florida – 6%, Massachusetts – 5%,
Minnesota – 6%, Missouri – 4.225%, New York – 4% (plus local option tax), Washington – 7.8%

Postage and Handling Charges: $5.50 per set for domestic postage and handling;
$8.00 per set for international postage and handling $ ____________

TOTAL (U.S. funds only) $ ____________

Name ____
(Please Print)

Address ____
(Please no P.O. Boxes)

____________________ Daytime Phone # (   ) ____________________

City ____________________ State __________ ZIP __________

Payment □ Check/Money Order □ VISA □ MasterCard □ American Express

Credit Card No. ________________ Exp. Date __________

Signature ____

□ Please send me [illegible]

Praise for
The MS-DOS® Encyclopedia:

"A superb, nearly inexhaustible reference work.... Anyone serious about programming for MS-DOS will not want to be without [THE MS-DOS ENCYCLOPEDIA]."
Online Today

"The ultimate authority."
Reference & Research Book News

"A splendid volume."
Dr. Dobb's Journal of Software Tools

"For those with any technical involvement in the PC industry, this is the one and the only volume worth reading." ***PC WEEK***

"If you like the idea of a one-stop DOS reference book, then this book is for you." ***PC Magazine***

"There's no doubting that this is a superb reference work on MS-DOS."
***EXE* magazine**

Here, from Microsoft Press, is the ultimate resource for writing, maintaining, and upgrading well-behaved, efficient, reliable, and robust MS-DOS programs. Covering all MS-DOS releases through version 3.2, with a special section on version 3.3, this encyclopedia is *the* standard reference for the working community of MS-DOS programmers and for anyone making strategic decisions about MS-DOS implementation. Included are version-specific technical data and descriptions for:

- More than 100 system calls — each accompanied by C-callable assembly-language routines and programmer's notes
- More than 90 user commands — the most comprehensive version-specific analysis ever assembled
- Key MS-DOS programming utilities and debuggers

THE MS-DOS ENCYCLOPEDIA has hundreds of hands-on examples and thousands of lines of great sample code plus in-depth articles on debugging, writing filters, installable device drivers, TSRs, Windows, memory management, the future of MS-DOS, and much more. There are also more than a dozen appendixes, an index to commands and system calls, and a subject index. THE MS-DOS ENCYCLOPEDIA was researched and written by a team of MS-DOS experts — many involved in the creation and development of MS-DOS — so you know it's accurate and authoritative.

U.S.A.	**$69.95**
U.K.	£48.95
Austral.	$104.95

(recommended)

ISBN 1-55615-174-8

56995

9 781556 151743