Master OS Development

FreeDOS Kernel

An MS-DOS Emulator for Platform Independence & Embedded Systems Development

Master operating systems development for PCs & embedded systems development

Achieve real platform independence with DOS compatibility

Provide embedded systems with full OS functionality

PAT VILLANI

CRC Press
Taylor & Francis Group

The FreeDOS Kernel

Pat Villani

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

First issued in hardback 2017

© 1996 by Miller Freeman
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

ISBN 13: 978-1-138-41230-9 (hbk)
ISBN 13: 978-0-87930-436-2 (pbk)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Bill, who tried to get me to write a book for 15 years.

To Kiddle, Mike, and Mom, for nodding and smiling.

And especially to Donna, whose love and confidence in me made this book possible.

Table of Contents

Chapter 1 Introduction 1

A Brief History of Desktop Operating Systems 2
MS-DOS 3
DOS-C 6
Why Roll Your Own 7
Educational Tool 8
Embedded Systems 9
Non-Intel Applications 9
DOS Clone 10
Extending DOS 10
Development Environment 11
Compiler 11
Assembler 12
Linker 12
Locator 13
Debugger 13
Getting Started 14

Chapter 2 DOS Basics 15

Where to Start 15
Basic MS-DOS Architecture 17
What Do Physical Disks Look Like? 21
Generic Disk Organization 22
Floppy Disk Drives 24
Logical Versus Physical 24
MS-DOS API 26
MS-DOS File System 30
Boot Area 33
FAT Area 39
Directory Area and Directories 44
MS-DOS Memory Management 48
MS-DOS Task Management 50
MS-DOS Device Drivers 52
What I Am Not Covering 54

Chapter 3 Bird's-Eye View of DOS-C 55

Basic DOS-C Architecture 56
Booting DOS-C 58
DOS-C Kernel Architecture 62
DOS-C API 63
DOS-C File System 64
DOS-C Memory Management 65
DOS-C Task Management 66
DOS-C High-Level I/O and Device Drivers 67
Final Notes 69

Chapter 4 ***DOS-C Kernel: File System Manager 71***

Device Driver Interface 72
Character Device Interface 75
Block Device Interface 84
FAT File System Functions 94
FAT Management 94
Directory Management 104
File Management 122

Chapter 5 ***DOS-C Kernel: Memory Manager and Task Manager 143***

Arena Management 145
Allocation Strategy Management 147
Arena Support Functions 160
Task Manager 166
Assembly Language Support 166
Process Environment 169
Program Loaders 182
Returning From a Process 199

Chapter 6 ***DOS-C Kernel: API 201***

System Call Support 203
DOS-C Personality Layer 216
Simple I/O Support 217
DOS Function Support 219
FCB Function Support 230

Chapter 7 Command Line Interpreter 239

Starting the Command Line Interpreter 240

`command.com` 242

Architecture 243

DOSLIB 253

Internal Commands 256

External Commands 259

Batch Commands 267

Other Options for Command Interpreters 274

Chapter 8 DOS-C Kernel: Putting It All Together 275

Organizing the Project 276

Building the DOS-C Kernel 278

Testing the Kernel (or Did I Really Want to Do This?) 285

Where Do You Go From Here? 287

Appendix A A Note about Portability 289

Appendix B The FreeDOS Project: Official FAQ File 299

Revision History 299

General Information 300

What Is the Goal of the FreeDOS Project? 300

What Is the Reason for FreeDOS? 300

What Will FreeDOS Not Be? 300

Who Owns FreeDOS? 301

How Is FreeDOS Project Development Organized? . . 301

Technical Information 301

What Exactly Is Included in FreeDOS? 301

Will FreeDOS Run on Some Particular Machine? 303

Will FreeDOS Run Some Particular Program? 303

Will FreeDOS Be Compatible with Some Particular Version of MS-DOS? 303

Specific Answers to Common Questions 303
Why Not Use DJGPP? 303
Why Not Add 32-bit Multitasking or Other Advanced Features? 304
Why Use MICRO-C? It Doesn't Support `floats` *or* `typedefs`*!* 304
But Wait! Why Is the DOS-C Kernel Written in Borland C? 304
I Can't Get Some Program to Work. Is FreeDOS Junk or What? 305
What Happened to FreeDOS? It Seemed Dead for a Long Time. 305
What Happened to James Hall, the Originator of FreeDOS? 305
How to Obtain More Information 305
Where Is the Official FreeDOS Web Page? 305
Where Is the Official FreeDOS FTP Site? 306
Is There a Mailing List for FreeDOS? 306
Is There a Usenet Newsgroup for FreeDOS? 307
Who Are the Contact People for FreeDOS? 307
What Other Documents Exist? 308
How Can I Participate in FreeDOS Development? . . . 308

Appendix C DOS-C Distribution Disk. 311

Before You Start. 312
Building DOS-C from the Diskette 312
Directory Structure 313
Organization in a Nutshell 313
Building a Bootable Floppy 314
DOS-C and FreeDOS 314
Please Help Us 315

Index 317

Chapter 1

Introduction

Ask anyone to define an operating system. I will bet that for every question you ask, you will receive a different answer. You may ask, "Is the definition of an operating system so vague?" I'm happy to answer — no. However, operating systems do mean different things to different people.

To many computer users, it is the full collection of utilities and the kernel that drives their computer to run their favorite applications. To engineers working with small microprocessor systems, it may be the kernel and drivers together that makes a complex control system work. No matter what the definition, an operating system of one form or another is an integral part of software engineers' and users' lives.

Also, operating systems are the subject of heated debates and argument. The debates range from the OS/2 versus Windows battles to the Berkeley versus System V UNIX debates. Lately, with the introduction of WindowsNT, the WindowsNT versus UNIX debate dominates discussions in many a workplace. These discussions typically result in heated debates that approach religious fervor over the choice of an operating system.

A Brief History of Desktop Operating Systems

Historically in the minicomputer and mainframe world, there were as many operating systems as there were computer vendors. Each vendor would design a unique operating system that took advantage of its unique hardware. Sometimes, the operating system changed drastically from one revision to another as the hardware changed, requiring major rewrites of application code. Fortunately, this was not the case for the desktop computing environment.

With the introduction of microprocessors, many individuals saw an opportunity to create small desktop computers. An 8-bit microprocessor, such as the Intel 8080 or Motorola 6800, was the central processing unit for the early desktop computers. Memory limitations were on the order of a few kilobytes, and some rather amazing applications ran in this small space. As the popularity of these computers grew, so did a small cottage industry serving hobbyists and small businesses.

Early users of these computers recognized advantages in sharing technology, and some early computer conferences attempted to standardize media formats and storage methods so programs be could exchanged. The storage medium for early desktop computers was audio cassette, with recording formats such as Tarbell and Kansas City Standard. Later, 8-inch and 5¼-inch floppy disks became affordable, and small operating systems such as CP/M and Flex further promoted standardization while still allowing room for experimentation.

The market for the desktop computer grew rapidly as did the complexity of microprocessors. Processors such as the Intel 8086 and the Motorola 68000 began promoting 16-bit and 32-bit capabilities, and companies such as Apple and IBM took advantage of these new processors. Operating systems developed for these new systems spawned a new generation of applications. These companies also spent considerably more time and money in planning markets than the earlier cottage industry that sprung up around desktop computers. Of particular interest to us is a case study of the joint IBM and Microsoft approach with MS-DOS. Its evolution will help shed some light on features and functions within MS-DOS.

MS-DOS

IBM's entry into the desktop market was with the IBM-PC. This desktop computer was the product of a skunk-works team within IBM that featured an Intel 8088 microprocessor as its CPU and 64Kb of RAM. Originally introduced with only cassette tape as standard storage, the IBM-PC also offered a 5¼-inch floppy disk as an option. To manage the diskette, IBM licensed the MS-DOS operating system from Microsoft.

The original design of MS-DOS evolved from an operating system developed by a Seattle company for its S-100-based desktop computers. Microsoft purchased all rights to it, because operating systems were new to them and IBM had approached them for software to run on its new PC. This system became the source code base for MS-DOS.

The MS-DOS precursor was similar to the CP/M operating system for the Intel 8080 and required little RAM, a distinct advantage because the 80x86 family is similar in design to the 8080. By design, the operating system provided a similar API (Application Programming Interface) to CP/M. As a result, MS-DOS 1.0 held very closely to the CP/M model. When combined with a familiar architecture, many developers were able to adapt their CP/M-based applications to MS-DOS. Favor quickly shifted toward Microsoft to the detriment of Digital Research, whose CP/M-86 operating system was the primary competition.

As the market matured, so did both MS-DOS and the PC. Demands for MS-DOS began to grow from outside IBM, with IBM leading the pack in technology. The next generation of desktop computers from IBM was the XT. This computer was similar to the PC but featured more memory and a 10Mb internal disk drive. Its motherboard also was capable of much more memory: 256Kb instead of 64Kb. IBM's target for this more powerful system was the business market. Previously, only large businesses could afford any kind of computing power. The XT addressed the needs of smaller businesses.

In order to address this new market, Microsoft developed a new strategy for their products as well. They adopted a dual approach to desktop computing environments. To manage the new internal disk drive, Microsoft adapted its XENIX operating system (an AT&T UNIX v7 derivative) for the XT. They also enhanced the MS-DOS operating system to support larger storage devices and added a XENIX-like hierarchical file system. Microsoft developers enhanced the MS-DOS API with many XENIX-like calls, which enabled easy migration of applications developed for a personal computer market based on MS-DOS to a business market based on XENIX. Microsoft also enhanced the MS-DOS command line interpreter, `command.com`, with I/O redirection and pipes similar to XENIX. This redesigned MS-DOS now became the basis for all new MS-DOS versions.

The lock-step product announcements continued with both IBM and Microsoft again adding new products in unison. IBM introduced the AT, based on an Intel 80286, and Microsoft released another version of MS-DOS to support it. At this time, Microsoft began to abandon its earlier XENIX product in favor of LAN connectivity and GUI interfaces. IBM also started a joint development with Microsoft for OS/2. This new operating system would take advantage of memory management and other advanced features that the new processor offered. The need for a new operating system was clear — MS-DOS lacked protection mechanisms, and it was very easy for a program crash to destroy the operating system. The Ctrl-Alt-Delete key sequence became very familiar to users, and a huge market for fixed-disk reconstruction programs was born.

For migration of MS-DOS applications to OS/2, IBM and Microsoft developed a strategy similar to their market entry strategy. They would provide all the functions that were previously available so that developers could quickly adapt existing applications to the new operating system. Virtual devices and API entries similar in functionality to BIOS calls eliminated the need for `int` xx-style real-mode device control, while retaining the functionality familiar to developers. With API calls defined in C, OS/2 encouraged greater developer productivity through the use of high-level languages and broke the traditional register call model that both CP/M and MS-DOS sported. Unfortunately, a design limitation of the 80286 forced some serious incompatibilities between OS/2 and MS-DOS.

The Intel 80286 featured two modes of operation, a real mode and a protected mode. The 80286 protected mode sported a new memory management scheme that modified the processor behavior so that memory access is very different from real mode. Any access to memory uses the contents of descriptors, in place of segment registers, to look up a new base address and add it to the offset address. The descriptor plus offset scheme expanded memory capacity from 1Mb in the 8086/8088 to 16Mb in the 80286. An unfortunate side effect of this scheme is that real-mode programs using segment plus offset addressing will not port easily to protected mode.

Additionally, a significant design oversight on Intel's part existed in the 80286. The processor could switch from real mode to protected mode in order to take advantage of features offered by the new operating system, but it could not switch back. Once in protected mode, the only way back was a hardware reset to the 80286. IBM designed new features into both the BIOS and hardware of the AT to make the switch back to real mode, but the context switch time was very high. Thus, new multitasking OS/2 applications worked in protected mode only, and MS-DOS real-mode applications were limited to a single task.

The OS/2 protected-mode environment was similar to environments found on larger computers but was incompatible with existing MS-DOS applications. This limited its popularity — users did not want to drop all their old applications and spend much more for special protected-mode applications — and generated bad press about OS/2. When combined with rumors of discord between the successful IBM/Microsoft team, users stayed with MS-DOS. Microsoft decided at this point to create a migration path that would always include MS-DOS compatibility for all its new operating systems.

New directions from Microsoft included a new GUI environment, Windows, that layered MS-DOS. For all subsequent releases after 3.3, MS-DOS and Windows became more tightly coupled so that both could cooperatively take advantage of 80386+ processors. These new processors featured improved memory management, linear addressing, and a V86 mode. The V86 mode allowed real-mode programs to function while taking advantage of operating system services for process protection and multitasking. Through it all, MS-DOS remained the foundation

for 80x86 desktop computing. This may change in the future, now that early machines are being retired from service, and Microsoft, feeling relief from supporting the older real-mode only computers, has many new, advanced operating systems in the wings.

DOS-C

DOS-C started as an experiment in writing device drivers using C for MS-DOS v3.1. Many articles and books that discussed the foundation of a DOS device driver had appeared on the scene; however, it seemed that each article described code written in assembly language only. In a discussion with some colleagues, I argued that this was unnecessary, citing UNIX as an example. As a result of these discussions, I began to write DOS device drivers in C. I developed both block and character device drivers, along with special C data structures to match the DOS request packet. This effort proved to be very successful, and I decided to continue with personal research to expand on these techniques. The goal: to create a full operating system by using the same techniques used to create the device drivers.

I began the design of a new architecture that would use assembly language only at the call interface and for process management, such as stack manipulation and context switching. The implementation of this operating system would take advantage of the C language features and require fewer resources to develop than the traditional assembly language designs. Although UNIX proved this concept for large multi-user operating systems, there were no documented instances for a traditional PC operating system.

Building on this knowledge, I developed a minimal operating system using the device drivers written earlier, along with a new 8086 interrupt API. Known as XDOS, it proved to be a functional operating system. This new operating system became the vehicle to explore booting techniques, and I developed a C library SDK (Software Development Kit) for it.

I later began to enhance XDOS and chose DOS as the new API. These enhancements included a more advanced architecture that uses

an IPL (Intermediate Program Loader) to set up the operating environment before loading the operating system itself. Additionally, reentrant system calls for real-time applications became part of the design. Using the name NSS-DOS (bearing the initials of the consulting firm I was part of in place of Microsoft's MS), I demonstrated this version to a few friends and business acquaintances. As a result of these demonstrations, a major defense contractor approached my consulting firm for a source license for this operating system. The only new requirement — it had to run on 680x0 processors.

This presented a new challenge. Because of the MS-DOS model used for the API, NSS-DOS relied heavily on a segmented architecture. To meet this challenge, I began a major redesign of NSS-DOS. Development of new portability techniques allowed compilation of the same source code base on a variety of hosts and with a wide range of compilers and target processors. The new version, DOS/NT, resulted from this work. The redesigned kernel used microkernel techniques with logical separation of the file system, memory, and task managers. The new design included a portable DOS API along with a new DOS SDK to guarantee portability. By removing all processor-unique code from the core functions, a highly portable operating system resulted.

The version described in this book, DOS-C, is derived from DOS/NT and is closer in design to MS-DOS: the kernel is one large program as opposed to many small programs. It is a nonmultitasking operating system that provides a large number of system calls similar to MS-DOS. This simplified design provides an operating system that readers can study without getting lost in the complexity that a microkernel entails. It also provides an excellent source code base to experiment on or with.

Why Roll Your Own

Although I discussed the reasons for the development of DOS-C, there are probably many readers who are wondering why bother at all. After all, a professional software company with a great deal of resources behind it wrote and maintains MS-DOS. Why would anyone develop a simple clone of a successful operating system?

Educational Tool

The first application for such an operating system is as an educational tool. Without a collection of reverse engineering tools, it is impossible to examine MS-DOS in any great detail. Anyone who has ever looked at maintenance of a software system will tell you that trying to get into the original programmer's head to figure out the reasons for a particular design, without proper documentation, is a monumental task — especially at the disassembled assembly code level. My compliments to all the folks who are pouring out reams of "undocumented documentation." Reverse engineering MS-DOS may also break any existing licensing agreements between you and Microsoft, so check your license before applying a disassembler to it.

My guess is that MS-DOS source code is almost entirely in assembly language, and the kernel itself is guaranteed to be assembly language. Even if you did see the source, it would be more difficult to follow than if coded in a high-level language. My goal of minimizing assembly language yields two benefits: the code is portable and easier to understand. Although a high-level language in itself does not necessarily guarantee portability, it is easier to port C source code than assembly source code. Also, conditional code and looping constructs are easier to understand in C than assembly language.

As an educational tool, you can replace any section of the tested operating system. This allows you to experiment with different algorithms. You may want to change the LRU (Least Recently Used) buffer algorithm for a different one and compare performance. You may also want to learn about the different algorithms in order to apply a similar technique to a different application. For example, loadable device drivers may be useful in a stand-alone application where the host uses different devices in different configurations, or the buffered keyboard input code may provide enough insight for an interactive display system.

Finally, you can test the code on a host MS-DOS or UNIX machine using a source-level debugger because the majority of DOS-C is C code. For example, performing DOS/NT code tests on a 680x0-based UNIX machine proved invaluable. Many of the "big-endian" versus "little-endian" issues surfaced quickly during the unit testing of individual components of the operating system.

Embedded Systems

Possibly one of the most useful applications is as an embedded operating system. Many times, an application requires the functionality of an operating system, such as file storage, embedded databases, or sophisticated device control. A general-purpose operating system used in an embedded system may not do the trick. Additionally, it may not be possible to develop the application on the host because of a lack of hardware or support tools.

You can simplify the design of a typical embedded system by using a DOS PC for application development and later linking to your DOS to create an application ROM. You can debug the application on the host PC and define all the DOS entry points used. Later, you can either replace the DOS functionality or use an embedded operating system with custom device drivers for your unique hardware. The embedded operating system approach eliminates work and becomes a tool for the developer. Additionally, the embedded operating system becomes an abstraction layer between the application and the embedded system hardware.

Non-Intel Applications

When the operating system design is portable, new opportunities open up for both traditional and embedded applications. One possibility in the area of embedded systems is to design your system and develop it as you would for a normal PC embedded system, then use cross-development tools to recompile the application. When combined with the cross-compiled operating system, software verification reduces to the level of regression testing on the new hardware.

Another reason for developing your own operating system is simply to adapt it to a new platform. There are many non-Intel processor single-board computers. To approach a company such as Microsoft to adapt MS-DOS to this custom hardware would be both time and cost prohibitive. However, with an operating system designed to be portable, moving the operating system to a new target is a question of cross-development tools.

DOS Clone

Another way to view DOS-C is as a declaration of independence. As the desktop computer market matures, Microsoft is moving away from MS-DOS as the foundation of its GUI products. Another example of the trend away from DOS is the decreasing size of the DOS clone market. Products such as DR-DOS are rapidly disappearing from the scene, which leaves older applications in an orphan state. Creating your own version of DOS solves this problem.

Extending DOS

Have you, as a developer, ever looked at the DOS API and felt that if it had just one additional feature, your professional life would be easier? Taking the challenge head-on and creating your own version of DOS is an excellent way of extending DOS.

A typical way to extend DOS is with the Novell approach. When they created their network product, it was as an extension to the existing MS-DOS `int 21h` call. They captured the call and searched their own system calls. If it matched one, the network product would service it. Otherwise, the processor was restored to its original state and the call was passed through to DOS. Microsoft also adopted this interrupt chaining with its standard `int 2fh` multiplex interrupt chain in v3 and above.

With your own DOS, you can extend system calls to cover your own special call by simply adding it to the function call handler. You write an extension, test it, and integrate it into your DOS. The simplicity behind this type of customization gives you, the developer, enormous power over your applications environment.

Development Environment

I expect to hear you now say: "Gee, this all sounds great, but what expensive tools do I need to do this?" As it turns out, you do not necessarily need any. By using a native 80x86 platform to develop for and test on, popular PC compilers from companies such as Borland and Microsoft are suitable as long as you follow special guidelines. Also, there is no need for special linkers if you design your architecture correctly. As I will describe in later chapters, the use of an IPL simplifies the files on disk, and you can use common `.com` and `.exe` file formats. Testing also becomes simpler through the use of some low-cost tools. With all this in mind, choose your tools.

Compiler

First, choose a C compiler. The version of DOS presented in this book compiles with Borland C (Borland International, Scotts Valley, Calif.), Microsoft C (Microsoft Corp., Redmond, Wash.) and other C cross-compilers. Special attention to the use of portable techniques pays off when using simple development tools. As it turns out, the only area that ties a given C compiler to its host operating system is the C runtime library. Avoid its use to ensure that your code will run stand-alone with the low-cost compilers.

Beware of floating point code used by the C compiler. Some compilers generate non-reentrant floating point code or code that uses non-reentrant library calls. Many times, a compiler's start-up code will take over a number of interrupt vectors. These vectors capture floating point exceptions generated when a call to a nonexistent coprocessor occurs. This is a common technique for automatic switching between coprocessor floating point calculations and software emulation of the coprocessor. For all these reasons, and more, avoidance of floating point code helps. DOS-C avoids all floating point code by avoiding the use of C library calls such as `printf()`, further simplifying the choice.

Assembler

Besides the C compiler, you will need an assembler. The need for this is twofold. First, the standard start-up code delivered with a native C compiler assumes a host machine and operating system. You will be working with a "naked" machine that does not have an operating system available (remember, you are creating it). As a result, you need to substitute your own start-up code that will take care of memory initialization, stack and register initializations, etc., and finally call `main()`. Second, you will use the de facto DOS standard. This means that you use the same registers for system calls, switch stacks during system calls, etc.

The support code must be written in assembly language, allowing us full control of the target machine. The `asm` shortcut provided by many C compilers simply does not suffice because we must move away from the C model for this support. You cannot afford stack register and frame pointer manipulations, and many times you will not generate code that resembles a C function.

Linker

Look at the linker provided with the C compiler you choose. For our 80x86 version, you can use the standard DOS linkers. However, for many platforms they may not be suitable. You may need some form of "strip" utility as typically supplied on many UNIX platforms so that you can adequately create an IPL. As you will see later, our architecture specifies that the kernel is an executable file for a known format. Our choice of the `.exe` file type simplified our decision, but you will need to be careful when examining other platforms and targets.

Locator

If you have decided to place the operating system into ROM, or if you have a special target machine architecture that requires you to fix the operating system in memory, you may also need a locator. This type of utility comes in various shapes and sizes and may be split between the linker and various other utilities, but it always serves the same purpose: to fix all memory references to absolute addresses and possibly to generate a special file format acceptable to various device programmers. There are two examples of this type of utility set. The first example is as a stand-alone locator. Multiple vendors in the 80x86 community have such utilities to convert DOS `.exe` files into "hex" files. UNIX users may find that their linker, `ld`, accepts special specification files and that other utilities will convert the resulting file into a hex file. Check your manuals because this is a function of the vendor whose UNIX you are running.

Debugger

With all your development tools in place, you will need to begin thinking about debugging tools. This choice is dictated by a number of factors. The most significant is your target architecture. If you plan to use your operating system in ROM, then a tool such as a ROM emulator that will accept a download from your development platform and emulate your target ROM(s) may be a good choice. This device will permit you to download your code directly, but you will need some other tool in combination with the ROM emulator to assist you in debugging.

If your budget permits, you may want to look into an in-circuit emulator. This tool replaces the processor on your target hardware and allows you to electronically look into your target. They typically provide all the favorite debugging methods available on host debuggers, such as single-step and break point. Additionally, they usually allow you to stop a processor that has stopped responding — a highly desirable feature for anyone who has experienced such a lock-up while running DOS-based native debuggers. Also, many other hardware features, such as trace buffers, are usually available for in-circuit emulators. This type of buffer allows you to look into the history of your program so

that if you wonder how you got to a certain point in your program, the trace buffer will point it out. Finally, if source-level debugging is an option, you should buy it. It will simplify your job by orders of magnitude. Just make certain that it supports your development tools.

If you are on a limited budget or you are using DOS-C as a learning tool, you will want to look into remote debuggers. Many of you are probably already familiar with the remote debugging features of debuggers such as Turbo Debugger (Borland International) or CodeView (Microsoft Corp.). These run on a remote DOS machine, but the type of remote debugger you will want to use is one that runs stand-alone. They sometimes require some work to adapt to the target hardware but are well worth the effort. Some will also work in combination with ROM emulators, making the combined tools almost as powerful as an in-circuit emulator.

Getting Started

Now that I have briefly touched on why and how to write your own operating system, I will begin to build DOS-C. I will approach its development in a logical fashion by first developing a set of requirements. I will examine MS-DOS from the outside in order to assist in the development of a set of general specifications. I will develop a general-purpose method of examining error recovery and apply it to round out our requirements. When completed, I will proceed to develop the operating system itself.

The full source code for this book is now available on the publisher's ftp site at `ftp.mfi.com/pub/rdbooks/FreeDOS.zip`; login as "anonymous" and download the file. Any references to the "companion code diskette" in the book now refer to the code available on the ftp site. See Appendix C for more details or check the FreeDOS home page at `http://sunsite.unc.edu/pub/micro/pc-stuff/freedos` or my personal web page at `http:/www.monmouth.com/user_pages/patv` for updates.

Chapter 2

DOS Basics

Where to Start

Many programmers have come to take MS-DOS for granted. Its API has become a standard in the industry and is known by developers world-wide. Many books, published by Microsoft and others, describe the interface. Many other books contain information about writing applications for DOS, and others talk about how to extend it. Still others describe how to write extenders, device drivers, etc. Additionally, periodicals dedicate space to and books are published about the "undocumented" system calls and data structures.

Although Microsoft produced a great deal of programmer and user documentation for MS-DOS, there is plenty of undocumented information about MS-DOS internals. Proof of this is the sheer volume of "undocumented" books and articles published about MS-DOS. However, even with the undocumented sources, the target audience for this documentation is application programmers who want to emulate some of Microsoft's private programming tricks, not for programmers who want to design their own version of DOS. Very little documentation exists about MS-DOS internals. You may need to provide the same interface as MS-DOS and in some cases emulate MS-DOS internals in your design in order to achieve any level of binary compatibility.

Whenever possible, I followed reference articles and texts during the design. These references are very good and cover a good portion of what you need to know. However, as pointed out earlier, questions will come up where you will need to perform little experiments to find out how DOS behaves. One reason is that many times, a programmer's references cover a subset of error responses; however, you will need to cover other undocumented instances. Another reason is that you can gain insight into the DOS design by running a test program that exercises a system call.

Exercising system calls through varying conditions is one of the best ways to study DOS. Many times, you don't know what algorithm the MS-DOS designers used. Quite often, it may be possible for many algorithms to perform certain functions. Unlike an open system where all functions of the operating system adhere to a well-known standard, Microsoft created the system functions and modified them as they went along. In some cases, the changes occurred as bug fixes or as the result of developer learning curves. In other cases, Microsoft created new system calls to take advantage of new features introduced with a release. As a result, many applications have become dependent on these features and any little quirks that they may exhibit. You need to emulate these features closely so that your version of DOS doesn't

break existing applications. Although more than one algorithm may achieve a given function, you must choose the algorithms for your DOS that closely emulate the original MS-DOS version.

This chapter contains a sample experiment whose purpose is to examine the minimum set of files and other objects that MS-DOS requires to run. The goal is to demonstrate a methodology that you can use whenever a design question arises. From there, the chapter covers the basic DOS boot procedure, physical disk organization, API, file system, memory management, and device driver.

Basic MS-DOS Architecture

The way I like to study any new subject is by starting with the overall picture. I like to peel away the different layers, uncovering new information with each new layer. I find this approach helps me comprehend a new subject better because I will not get lost in the details too early in the game. I will take this approach and begin by looking at the four basic MS-DOS files: boot, `io.sys`, `msdos.sys`, and `command.com`. You may ask, "How do I know that these are the basic MS-DOS files?" Perform a simple experiment to find out.

You will format a bootable floppy disk as the basis of this experiment. Take a blank disk, place it into drive A, and type `format a: /s` at the DOS prompt. MS-DOS chugs along and gives you a bootable disk. You know it's bootable because you can now hit Ctrl-Alt-Delete and up comes MS-DOS. You can copy files, delete files, execute other programs, etc. In short, every essential MS-DOS function is available at the new prompt.

Now take a closer look at this bootable disk and see what MS-DOS has done to it, aside from laying out track and sector structure to make it bootable. When you examine the directory, you find one file, `command.com`, on it. This file is the MS-DOS command line interpreter or shell. If you do a `dir a: /a:h`, you find two hidden files, `io.sys` and `msdos.sys` (or `ibmbio.sys` and `ibmdos.sys` in certain versions of MS-DOS and PC-DOS). It seems that Microsoft considers these three files to be the minimum components needed for MS-DOS to operate.

This is interesting, but you know from the references that one other component will not appear in any directory listing because it is not a true file: the boot sector. The boot sector is the first sector placed onto the disk by the `format` command after the disk physical format operation. The PC architecture defines its function, and it is a de facto standard. All PC-style computers load and execute the boot sector in order to boot any operating system; it is the final component to the minimum set of programs for MS-DOS.

Each of the four components has a specific role in starting MS-DOS. As you can see, your little experiment proves that only four components are necessary for a minimal DOS operating system. If you limit your design to only these components (or their equivalent functions), your design becomes easier to handle by eliminating the development of transient commands such as `attrib`, `format`, etc.

Having established the minimum set of programs necessary for MS-DOS to operate, you can examine the dynamics of these four programs. If your computer is a standard PC-class computer like mine, the BIOS performs a power-on self test (POST) the instant you turn your computer on. The basic tests your computer may perform are a memory test, followed by a keyboard test, disk test, and hard drive test. You may also see coprocessor tests and other varied tests interspersed among the basic tests. At the end of these tests, you will hear a beep. The question now is, "What happens after the beep?"

The answer is simple: Your computer's BIOS now attempts to boot the operating system from either the floppy disk or the hard disk. If it's booting from a floppy, it brings the boot sector into memory at location `0:7c00h`. This is an important address, because all BIOS guarantee to load the boot at this location for compatibility reasons. If the boot sector is not at `0:7c00h`, MS-DOS will not boot properly because a closer examination of the boot program reveals that its address modes require this start address (unlike a `.com` or `.exe` file that can load anywhere within the x86 real mode memory). If it's booting from a hard drive, the BIOS loads a master boot sector to determine the active partition and then loads the boot record into the same address. However, for educational purposes, you will continue to examine a floppy disk boot.

After loading the boot, the BIOS performs a `far` jump to `0:7c00h`. MS-DOS places a boot sector onto a disk or hard disk regardless of whether it is bootable or not. This protects the user from inadvertently attempting to boot from a nonbootable disk (unfortunately only for x86 processors). By design, the boot checks for a bootable hard or floppy disk by looking at the root directory for the presence of two required MS-DOS files, `io.sys` and `msdos.sys`. If boot does not find them, it issues the well-known "non-system diskette" message and waits for a new disk to retry. The hard drive boot does not wait but halts, since a boot failure means that something is seriously wrong with the active partition, whereas a floppy boot error may result from a user inserting the wrong floppy disk into the boot drive.

The boot next searches the root directory for `io.sys` and `msdos.sys` (or `ibmbio.sys` and `ibmdos.sys`). It reads each entry in the root directory in order to determine the filenames. If these files are present, boot proceeds to load `io.sys`. It does this by getting the starting disk sector from the matching `io.sys` entry, copying each sector into a fixed location in memory, and transfering execution control to it. One significant boot limitation is that it can only load sequential records. Boot's size limitation is significantly less than one sector, and the developers who created boot were very creative when it came to trading off functionality versus size.

Once the transfer of control takes place, `io.sys` proceeds to initialize its device drivers. These drivers are the fundamental device drivers that every DOS system has: floppy disk, hard drive, console, printer, and auxiliary serial port. Up until this point, boot communicated strictly through the BIOS functions in order to save space and achieve portability, but `io.sys` now uses its device drivers for control. `io.sys` continues to initialize its device drivers by loading `msdos.sys` and instructing it to initialize itself. It then uses `msdos.sys` to read `config.sys`, if present. `io.sys` parses each line of `config.sys` and sets up the environment space, loads and initializes the device drivers, and performs other functions necessary for a successful boot. When `io.sys` completes all this, it loads `command.com` and finally transfers control to it.

Notice that I have not mentioned the familiar DOS startup file, `autoexec.bat`. That is because it is really not part of the actual boot process but a convenience placed into `command.com`. The designers of MS-DOS knew that many people modified CP/M in order to execute a program on startup and that other operating systems, such as UNIX, had special files or directories of files that executed on startup (i.e., `/etc/rc`). These files are extremely useful in turnkey applications and tailoring environments. You can perform a short experiment to verify that `autoexec.bat` is a function of `command.com`. You can take the sample disk formatted earlier and add a simple `autoexec.bat` file to it. Next reboot your computer to verify that the new file executes. Then change `command.com` to a simpler or different command line interpreter. What you will see is the same as before, except without the execution of `autoexec.bat`. By verifying that `autoexec.bat` did not execute when the command line interpreter changed, you verified that its association is with `command.com`.

You now have enough data from this little experiment to generalize the MS-DOS boot procedure.

1. Four mandatory components of a basic MS-DOS system are: boot, `io.sys`, `msdos.sys`, and `command.com`. Two optional files are: `config.sys` and `autoexec.bat`.
2. To initiate a boot sequence, load the boot sector at `0:7c00h` and execute it.
3. Boot must examine the disk. If bootable, it loads the first stage, `io.sys` (or `ibmbio.sys`), and executes it.
4. Upon transfer of control, `io.sys` loads `msdos.sys` (or `ibmbio.sys` loads `ibmdos.sys`) and allows it to initialize. `io.sys` then uses `msdos.sys` to parse and execute `config.sys` if present.
5. When completed, `command.com` is loaded and control is transferred to it.
6. `command.com` executes `autoexec.bat` if present and displays the familiar DOS prompt, awaiting user input for commands.

You will use this information later when you design your architecture for DOS-C.

What Do Physical Disks Look Like?

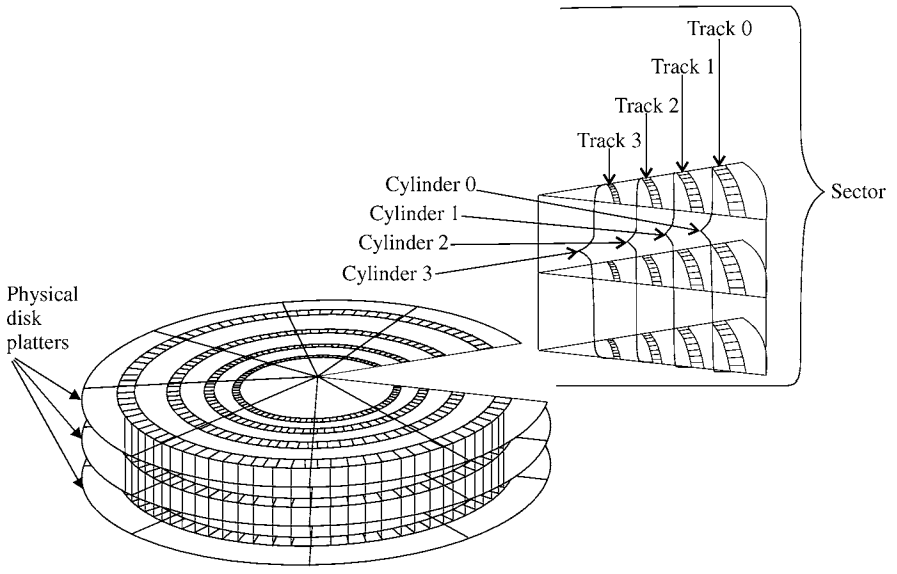
One area of mystery to many developers is the physical location of sectors stored on a disk. Unless you've developed disk drivers before, you've probably never seen this. Even if you did write a disk driver, odds are that you've never looked at the physical file system organization. I'll show you the physical and logical organization of the MS-DOS file system so that the mystery will finally be solved.

Generic Disk Organization

Disk drives are ubiquitous. Nearly every desktop computer, workstation, server, and mainframe has at least one. They are the preferred secondary storage for computers. Typically, their size is many times the primary RAM and ROM storage contained within the computer itself. They're also taken for granted. Why? Because storing data onto the disk is done without regard to either physical or logical organization. Just pause for one moment and ask yourself when was the last time you thought about which sectors the file you just created occupies?

Before I get into such details, you need to understand the terms used to describe how the disk stores data and how the disk actually works. Figure 2.1 is a view of a generic disk. This disk has one or more physical *platters* that hold the data. Each platter allows data to be read and written from either one or both surfaces due to a coating capable of being magnetized. This coating is similar to that used on mylar tapes

Figure 2.1 Diagram of a generic disk architecture.



that store data or audio. The data is stored by magnetizing a portion of this surface as the surface moves beneath a read/write head. Unlike a tape, where the tape is spooled from one reel to another, a disk rotates so that the same physical location is available once per revolution and data can be written and repeatedly accessed by waiting for the next revolution. This stripe of recorded data is called a *track.*

This track contains all the data. You need to access this data much the same way you access internal memory — you need to be able to modify it at random intervals. In order to facilitate this, the track is broken down into *sectors* that have prerecorded markers (laid down by a formatting program) that identifies where a sector is physically located on the track. By prior agreement, this sector is defined as a fixed number of bytes designed to fit within the physical space between the prerecorded markers. With a fixed number of bytes contained within each sector, you can store the number of bytes equal to the size of the sector times the number of sectors recorded on each track.

There is more than one track per disk. In fact there are two directions in which you can distribute the tracks: on different surfaces of the platter and across the surface. Typically, a disk has more than one surface. It usually has two surfaces and multiple platters. By placing a read/write head in close proximity of each surface, you can access this data by electrically switching heads and reading or writing data on this surface. This organization is known as a *cylinder* because you can visualize the track as physically extending through each platter forming a three dimensional cylinder of data. You can now store the number of bytes in each track times the number of tracks in each cylinder. By moving the heads, you can take advantage of the surface area by forming concentric rings or tracks on each surface and concentric cylinders across the platters. With this additional dimension of storage, you can store the number of bytes in each cylinder times the number of cylinders on each disk drive.

Floppy Disk Drives

To better understand how the logical MS-DOS file system organization relates to the physical disk, we need to select a specific disk type to study. Hard drives vary in architecture from vendor to vendor. Just look at the number of drive types in your computer's BIOS. Floppy disks are much more standard and easier to study. I'll discuss a 360Kb floppy disk.

Like the generic disk, the data organization on a floppy disk is broken into sectors, tracks, and cylinders. The platter on a floppy disk is made from mylar as opposed to aluminum or some other alloy. Because the mylar is very flexible when compared to the metal platter, the term "floppy" was applied to it. There is only a single platter with one or two surfaces, so a maximum of two tracks compose a cylinder. The number of cylinders vary, but most common is 40 or 80.

For a 360Kb floppy, the organization is nine sectors per track, two tracks per cylinder and forty cylinders per disk. Other architectures are possible, such as 720Kb and 1.44Mb floppy disks. These disks are also different physical sizes, typically 5 1/4-inch and 3 1/2-inch, but other sizes such as 8-inch and 3-inch existed in the past. With MS-DOS, the sector size is typically 512 bytes, but other sector sizes were also used in the past.

Logical Versus Physical

Both MS-DOS and DOS-C internally reference the disk drive as an array of fixed-size blocks of data. Each block of data in this array is sequentially numbered from zero to one less than the maximum size of the disk (in sectors). DOS-C and MS-DOS impose a structure upon this array of sectors in order to easily access data and organize files. This logical file system organization is shown in Figure 2.2. DOS-C and MS-DOS place a boot sector at the first sector on the disk because this is the best defined sector on the disk. This sector contains the code to load the operating system and a data structure that contains all the pertinent variables that define the disk geometry. For a 360Kb floppy disk, the next sector is a reserved sector and it's followed by two FAT areas,

each two sectors long. The last area reserved by the file system is the root directory and it's 7 sectors long. The remainder of the disk is dedicated to data and subdirectories.

Mapping this logical structure into the physical disk requires some thought. In order to map the disk, you need to standardize how the logical sector numbers map into physical sector, cylinder, and track numbering. For MS-DOS and compatible operating systems, the first block is mapped into sector 1 of cylinder 0, track 1. The track number increases for every nine sectors on a 360Kb floppy, followed by

Figure 2.2 Diagram of logical file system organization for MS-DOS and DOS-C.