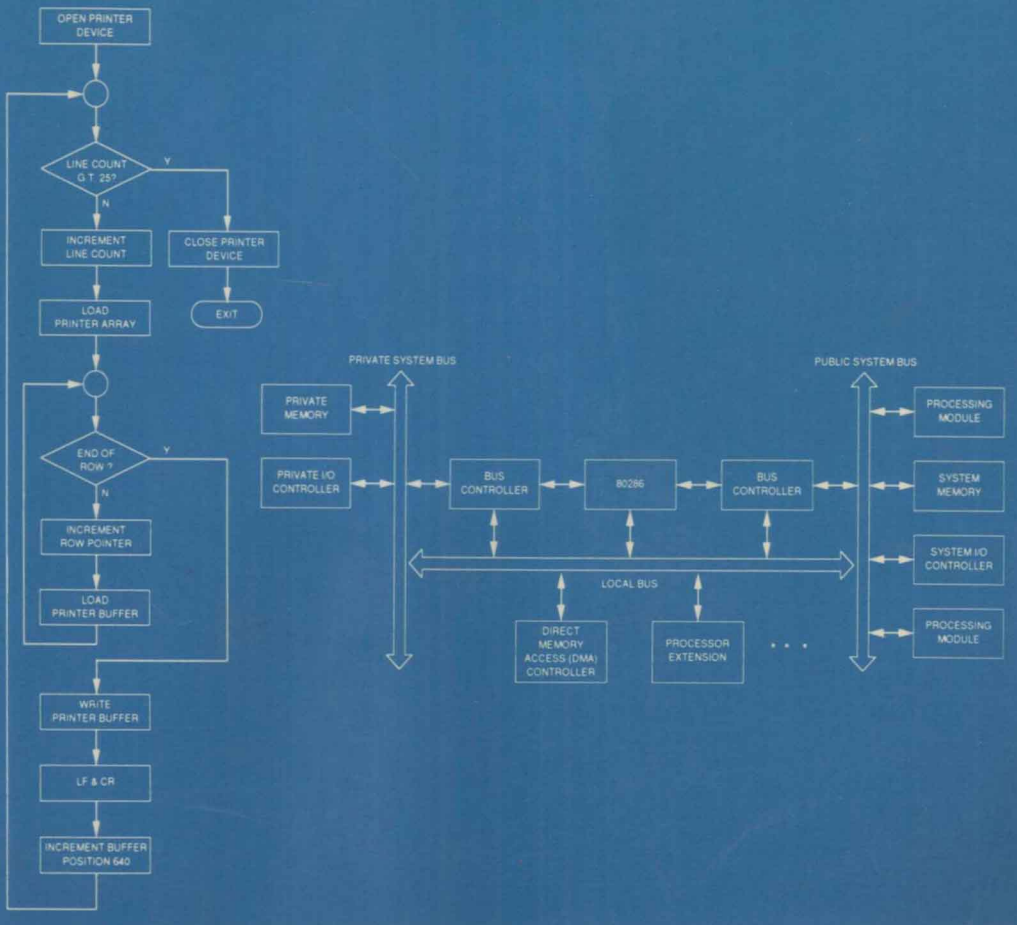
J. TERRY GODFREY

Programming the OS/2 Kernel



Programming the OS/2 Kernel

J. Terry Godfrey
President, JTG Associates

PRENTICE HALL
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Godfrey, J. Terry
Programming the OS/2 kernel / J. Terry Godfrey.
p. cm.
ISBN 0-13-723776-6
1. OS/2 (Computer operating system) I. Title.
QA76.76.063G63 1991
005.4'469--dc20 90-7518
CIP

Editorial/production supervision and
interior design: *Kathleen Schiaparelli*
Cover design: *Wanda Lubelska*
Manufacturing buyer: *Lori Bulwin/Linda Behrens/Patrice Fraccio*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-723776-6

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Programming the OS/2 Kernel

To Judy, Ray, Agnes, and Emma

Preface

This book has been developed for teaching programming using the IBM Operating System/2 (OS/2). It is suitable for a one-semester course in OS/2, as an adjunct to a course in operating system design, or as a vehicle for self-study on OS/2 programming. The emphasis in the book is on programming techniques for an advanced multitasking microcomputer operating system. Both Macro Assembler/2 and the C language are supported in the text. The OS/2 Application Programming Interface (API) services can be understood in either context.

The text addresses the basic OS/2 kernel services: the video (Vio), Disk Operating System (Dos), keyboard (Kbd), and mouse (Mou) API functions. The latter service is most useful in a windowed display such as the Presentation Manager, which is omitted from this text. The book concentrates on the OS/2 Full-Screen Command Mode, which utilizes the entire display for presentation of a single program, making no other programs visible. Similarly, input and output under program control is implemented through the standard assembler or C syntax, such as printf() or scanf(). These operate in Protected Mode as well as Real Mode. Consequently, there is little need to incorporate specific keyboard API services into the program examples. Keyboard and mouse functions are discussed briefly in Appendix D. Some use is made of the keyboard services, for example, to pause the graphics screen.

The Presentation Manager windowed interface is not developed in this book. Although this is a rich and complex interface, it is not considered suitable for a one-

semester course on OS/2 programming. The services at the level of IBM's OS/2 Standard Edition 1.0 are assumed as sufficient material for such an introductory course. When object-oriented programming tools become available for the Presentation Manager and the burden for programming this interface is eased, it will be appropriate in a beginning course in OS/2 programming.

During the late 1980s when OS/2 was developed, the principal major competing operating system for advanced microcomputer applications was UNIX. OS/2 follows IBM's earlier microcomputer operating system, Disk Operating System (DOS), and runs DOS as a subset. UNIX has tended to be used more within the scientific and engineering community and is generally optimized for larger machines than the baseline microcomputers developed during this time frame.

What are the advantages afforded by OS/2? OS/2 is predominantly a multitasking operating system capable of extensive memory management. It accomplishes these activities through hardware intervention based on the Intel 80286 chip set. (Hardware compatibility exists at the 80386 and 80486 levels.) There are four levels of protection provided (unlike the Motorola 68020 and 68030, for example, which have two); hence OS/2 can be tailored to handle the multitasking problem. The protection mechanisms provide coarse-grained through fine-grained memory management. This allows a detailed dynamic memory allocation at any given time.

If we examine OS/2 in the framework of the near-term evolution of microcomputer systems (1990s), it is apparent that changes in software development and applications will dictate about an order-of-magnitude increase in software complexity. It is clear that many efforts will give way to multithread and multiprocessor programming. The OS/2 multitasking features make it a good candidate for major microcomputer applications during the 1990s time frame. Also, the hardware protection mechanisms mentioned above are suited for minimizing operational errors in such multitasking situations. Hence OS/2 is positioned to become the operating system of choice for high-end personal computer applications based on the Intel chip sets.

OS/2 is particularly suited for user-friendly operation and programming. The API services are readily programmed in a fashion similar to the now-more-familiar Basic Input Output System (BIOS) interrupt calls. The Presentation Manager represents a large-scale object-oriented interface. It is programmed in an almost identical manner to the Microsoft Windows Software Development kit (SDK) programming. OS/2 is moving rapidly toward widespread acceptance as the IBM microcomputer operating system for the early 1990s, just as DOS was for the 1980s.

This book is intended to teach techniques on how to program in an advanced multitasking environment. The approaches required for software development reflect the solutions and compromises that exist in the 80286 hardware and the OS/2 Protected Mode software. The power of OS/2 lies in its potential to run a number of large-scale applications simultaneously, with asynchronous and synchronous sharing of data. The use of pipes, queues, and semaphores (as well as shared memory blocks) ensures that intertask communication minimizes errors and follows well-established guidelines.

OS/2 is large, but experience has demonstrated a rather elegant superstructure that combines Microsoft Windows, DOS, multitasking, and memory management. Even in the scaled-down 80286 environment, OS/2 presents a very user-friendly interface to the hardware. Finally, all the programming skills developed for the earlier DOS framework are applicable when writing software for OS/2. IBM and Microsoft have maintained many philosophical features of DOS while incorporating the Apple MacIntosh-like graphical interface in PM. OS/2 is truly an order-of-magnitude change in microcomputer operating systems. The potential for large-scale object-oriented applications is intrinsic to the PM definition.

This, then, is the world of OS/2 as we move through the 1990s. The reader can expect a programming arena in which multitasking is important. This is a precursor to the parallel processing systems coming toward the end of the decade. At the same time, implementation of segmented large-scale applications becomes a reality through interprocess communications and memory management. Thus efficient use of microcomputer resources becomes feasible. Finally, graphical interface techniques lead to very user-friendly application environments. OS/2 promises to be at the forefront of microcomputer operating systems because of all these features.

One comment about the style used in this book. The IBM macro calls to the Application Program Interface (API) are used throughout. This is in keeping with the trend toward higher-level-language constructs and structured code when developing assembler programs. It does have the effect of obscuring the stack loading during an API call and assumes that the reader has access to the IBM API macros (i.e., the IBM Toolkit include files). The trade-off, however, is that fewer lines of program code need to be understood, and for someone familiar with the calls, the inferences are clear. This has implications for maintenance as well as debugging.

This text is practically oriented. The examples are somewhat lengthy, by intention and as a real-world case would be. They are intended for the serious student who is interested in programming under OS/2. The Color Graphics Adapter mode (CGA) is illustrated because of its relative simplicity and ease of programming. Also, it is a readily testable feature that can easily be programmed using C or assembler. The book assumes that the student has a basic familarity with C and assembler.

ACKNOWLEDGMENTS

As is to be expected, a great many people contributed to this book both knowingly and unknowingly. It is impossible to give credit in all cases; however, a few notable exceptions are my wife, Judy, who did all the typing and much of the editing; Marcia Horton, Editor-in-Chief at Prentice Hall, who was always available to answer questions and provide inputs; Ray and Agnes, my parents, who laid the groundwork for this book years ago, and Emma, Judy's mother, who provided both of us with a sense of stability. Thanks to Kathleen Schiaparelli and her staff for their excellent job producing the book.

Finally, special mention should be made of the help I received on BIX, *Byte Magazine's* bulletin board, for those unanswerable questions that plague every programmer and can be answered only by someone else. Like many other forums, BIX is an excellent place to go for answers because of the depth and breadth of experience displayed by its membership. Also, the thoughtful comments provided by Margaret Mooney added a new perspective.

Contents

PART I
Introduction to OS/2

The OS/2 Environment

During the 1980s, IBM developed (in conjunction with Microsoft, Incorporated) the Disk Operating System (DOS) [1] as a primary operating system for its family of microcomputers: the IBM PC, XT, XT286, AT, PS/2 Models 25, 30, 50, 60, 70, and 80. These systems were developed using the Intel family of central processor unit (CPU) chips, including the 8086, 8088, 80286, and 80386 [2–4]. DOS is a single-thread single-user system and hence is capable of executing only one task at any given time. Intel, however, provided the 80286 and 80386 with architectures that ensure hardware protection for multiple applications. This prevents code segments from being mixed during execution of multiple separate tasks. Such multitasking is the framework required by the advanced applications in existence and slated to arrive throughout the decade of the 1990s.

Toward the end of the 1980s a clear need developed for an operating system that was capable of supporting and utilizing these advanced microcomputer hardware architectures. In response to this need, IBM and Microsoft developed Operating System/2 (OS/2) as their candidate to run on the Intel 80286-based (and 80386) machines [5,6]. There are many facets to OS/2. Both IBM and Microsoft have provided information needed to be able to program in the OS/2 environment through their Toolkit (IBM) [7] and Software Development Kit (Microsoft) [8]. Initially, following an early issue by Microsoft in 1987, IBM released OS/2 Standard Edition Version 1.0 in December 1987. This early version employed the full-screen command prompt mode only, which initially displays a menu followed by a screen with

header. Basically, two modes were allowed: DOS compatibility mode, which runs from a screen with a typical prompt such as

(C:\>)

and runs DOS programs, and OS/2 Protected Mode, which runs from a screen with a typical prompt such as

[C:\>]

In the fall of 1988 IBM released Version 1.1 of the Standard Edition, which included the Presentation Manager (PM) [9]. This provided a full Windows-like graphical interface to the user. This graphical interface is very similar to that found with the Apple MacIntosh operating system [10].

In addition to the Standard Editions, IBM and Microsoft have developed an Extended Edition, which has a local area network (LAN) interface and a database manager with support for Structured Query Language (SQL). The later editions of OS/2 (Extended Edition 1.0-10/88 and 1.1-11/88) function in essentially the same fashion as the Standard Edition; hence we will focus on the Standard Edition and not address the LAN and database features in this book. Basically, we are interested in programming highlights rather than specialized application packages.

IBM recommends a minimum of 2 megabytes (MB) of random access memory (RAM) for running Standard Edition 1.0, 3 MB of RAM for Version 1.1, and 3 MB of RAM for the Extended Edition (EE). Also, the EE may completely consume a 20-MB hard disk drive [11]. Most versions of OS/2 come complete with the CodeView debugger, which is capable of debugging both assembler and C code. These are the two languages considered in this book. The language support for OS/2 is extensive with assembler, FORTRAN, BASIC, C, Pascal, and COBOL compilers existing. As indicated, we will focus on C [12] and assembler [13] for the OS/2 environment. Although IBM provides a Protected Mode editor with Version 1.1, in the program development for this book, VEDIT PLUS [14] was used as a full-screen editor run from the DOS compatibility box. This process was quite smooth and allowed for early development when only Version 1.0 was available. Context switching between Real (DOS compatibility box) and Protected Mode was accomplished rather efficiently in the OS/2 implementation. Programming the Presentation Manager graphical interface is very much a Windows-like exercise [15].

With these introductory remarks in mind, where are we going with this book? The goal is to establish for the reader the capability to write programs in the OS/2 kernel environment. We address code development in assembler (IBM Macro Assembler/2) and C (Microsoft C Compiler Version 5.1).

What is so unusual about OS/2 in relation to conventional Real Mode (Intel 80286 Real Mode) programming? In OS/2 the major achievement is the definition of API services for access of the Protected Mode multitasking and memory management features. Typically, an entire new class of function calls is added to the usual assembler or C code. These functions (the API) constitute the system interface and have syntax (in ASM) like

```
@DosExit action, result
```

instead of the normal return instruction, ret, or

```
...
@VioScrLock waitf,iostat,viohdl
@VioGetPhysBuf PVBPtrl,viohdl
...
@VioScrUnLock viohdl
...
```

instead of int 10H. Hence it is apparent at a glance that OS/2 function calls tend to require more parameters (versus register setup) than conventional assembler. They have the added attribute, however, of being a symbolically elegant interface. By the latter reference, we mean that the API services appear as a natural extension of assembler or C code in modular and complete fashion.

OS/2 is a model operating system for illustrating advanced features in a systems software framework. As discussed, it is somewhat RAM intensive, although it will run comfortably with 2 MB as an installed base. The principal accomplishment is the segregation of services for operation in the multitasking environment. How this segregation is accomplished is reflected in the programming techniques used to write code for OS/2. OS/2 is a good example of how multitasking should be implemented.

HARDWARE CONSIDERATIONS

OS/2 is written primarily for the architecture of the Intel 80286 (and is compatible with the 80386) as it exists in Versions 1.0 and 1.1 of the Standard and Extended Editions. The manner in which the hardware and software coexist depends largely on the Intel concept of segmented memory and the notion of levels of protection. We examine these aspects of OS/2 and attempt to correlate the register-level hardware with OS/2 address allocation. It is important to recognize, however, that keeping with the Intel philosophy of downward compatibility, subsequent microprocessors in the 8086 family run code intended for the earlier chip sets. Hence the 80386 architecture, although more advanced than the 80286, will support 80286 Protected Mode software. This means that OS/2 runs on 80386 machines as well.

1.1.1 The 80286 and 80386 Architecture

It is worthwhile examining the Intel 80286 (and 80386) architecture at this point because this implementation serves as the basis for development of programs such as OS/2. Once we have touched on this hardware foundation, we can forever assume that a starting point exists from which to explore the features of 80286 systems software.

Intel started the 8086 family of microprocessors with initial entries that have 16-bit addressing. This includes the 8086, 8088, and 80286 chips. The 80386 has

32-bit addressing and represents a major step forward, in keeping with the increased speed of these integrated circuits. What is the major limitation of the 16-bit architecture? In a physical sense (based on the actual wiring of circuits and memory) 16 bits provides only 2^{16} or 65,536 possible individual references. This is the usual 64K segment. Recognizing that this constituted a very limited memory capability, Intel expanded the addressing concept to allow for multiple segments by providing a set of segment registers used to hold segment addresses. (This was in addition to the 16-bit instruction pointer that held an offset into the code segment, for example.) When IBM implemented the Real Mode operating system DOS, a 1-MB address limit was built into the architecture which was based on a 20-bit address. Addressing was accomplished by shifting the segment address left 4 bits, appending a zero (hexadecimal) to the segment address, and adding the offset to get the five-digit hexadecimal physical address. For example, assuming a segment address 10AF and an offset F0FF this physical address is

```
1 0 A F 0    (segment address)
F 0 F F      (offset address)
1 F B E F    (physical address)
```

where the usual notation would be 10AF:F0FF. What are the register structures used to support this addressing scheme? In the 8086 and 8088 the following registers exist:

Data

AX the Accumulator: This register can be used for general programming storage.

BX the Base Register: This register is frequently used to hold address values when accessing memory.

CX the Count Register: During loop operations this register holds the count index.

DX the Data Register: This register is used for general storage.

Segment

CS the Code Segment Register: This register points to the beginning of the code segment block.

DS the Data Segment Register: This register points to the beginning of the data segment block.

SS the Stack Segment Register: This register points to the beginning of the stack segment block.

ES the Extra Segment Register: This register points to the beginning of the extra segment block.