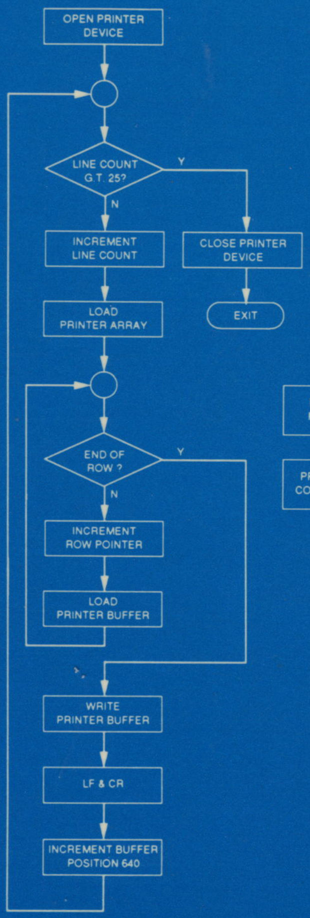
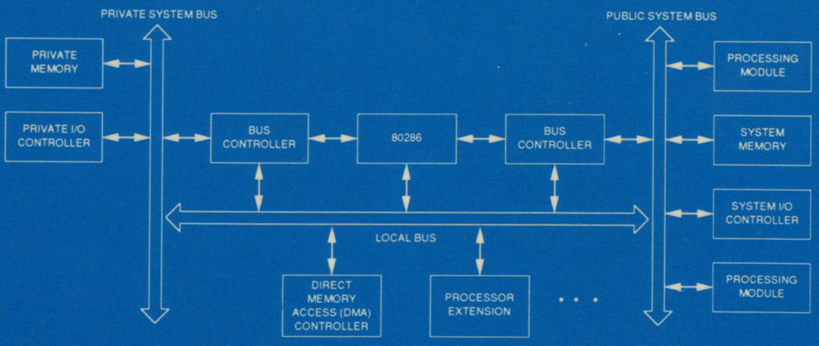
Programming the OS/2 Kernel

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*

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

UNIX is a registered trademark of AT&T (Bell Laboratories).
Apple and MacIntosh are registered trademarks of Apple Computer, Inc.
Intel is a registered trademark of Compuview Products, Inc.
Microsoft Window is a trademark and Microsoft is a registered trademark of the Microsoft Corporation.
IBM and IBM PC/XT/AT are registered trademarks of International Business Machines Corporation.

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*

To Judy, Ray, Agnes, and Emma

Contents

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.

Programming
the OS/2 Kernel

PART I
Introduction to OS/2

1 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  PVBPtr1,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.

1.1 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.

Pointer

SP the Stack Pointer: This register holds offset values for the stack.

BP the Base Pointer: This register holds offset values into the data segments.

Index

SI the Source Index: This register holds an index offset in memory and frequently references the instruction source.

DI the Destination Index: This register holds an index offset in memory and frequently references the instruction destination.

Added to these 12 registers are the instruction pointer (IP) and flags registers, yielding a total of 14 16-bit registers for the 8088 and 8086.

The 80286 enhances this register set with the addition of five registers:

GDTR the Global Descriptor Table Register: This register points to system resource segments.

IDTR the Interrupt Descriptor Table Register: This register points to interrupt service routine segments.

LDTR the Local Descriptor Table Register: This register points to the active local program code segment.

TR Task Register: This register holds the code segment address for the current task.

MSW the Machine Status Word Register: This register sets up the processing for real or Protected Mode.

These Protected Mode registers plus some others are used by the operating system to provide proper address allocation during execution of an active Protected Mode task.

Figure 1.1 illustrates a typical 80286 central processor unit (CPU) environment. Here the parallel external bus structure is apparent. The 80286 control of both the private and public system buses is via bus controllers (typically, an Intel 82288 and 82289 combination). In the IBM AT and PS/2 Model 50 and 60 such a bus structure exists with a representative architecture as depicted in Figure 1.2. Both these figures are similar and illustrate the parallel bus structure typical of 80286 systems. The private system bus contains localized private input/output (I/O) processing and buffering such as RS-232C adapters and video adapters, which constitute external physical entities. These are frequently accessed using direct memory access (DMA) controllers. The 8259A programmable interrupt controller (PIC) interfaces external hardware interrupts to the CPU. Both the private and public system buses have a three-part architecture: a control, address, and data bus subset. Control bus interfaces are handled by an 82288 bus controller with associated address decode logic (this is typically implemented using LS138 decoder/demultiplexers) [16,17]. The address I/O is handled using latches, which hold and strobe address data onto the system buses in response to enable signals. Finally, data is placed on the data bus using transceivers (typically, the LS245 transceiver).

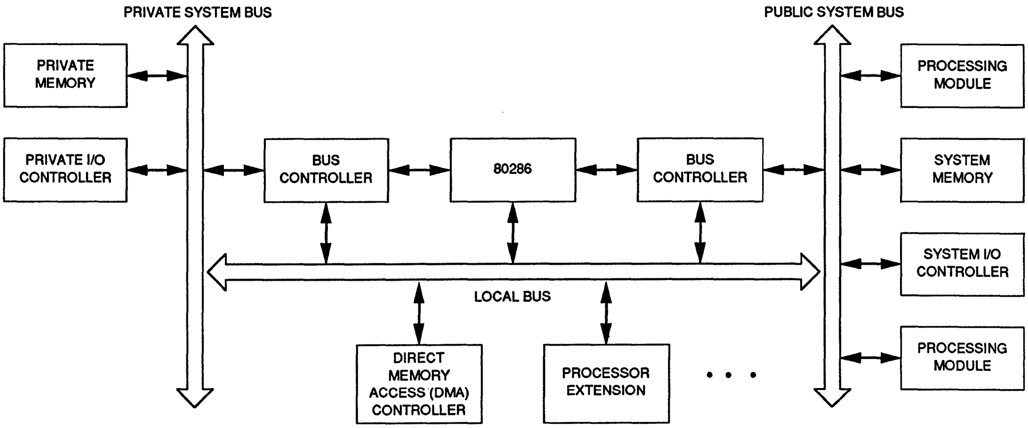


Figure 1.1 Representative 80286 system environment illustrating local and system buses.

Figure 1.3 illustrates a representative memory and port allocation of addressing among public and private spaces. For low memory, a 1-MB partition is illustrated from 0H to 0FFFFFH (here H indicates hexadecimal). This corresponds to the DOS partition in the IBM microcomputer address space. Local firmware is illustrated at the top end of the 16-MB physical memory space (OFFF000H to 0FFFFFFH). This would be in the OS/2 extended memory area. In the IBM systems, erasable programmable read-only memory (EPROM) exists between portions of the 640K and 1-MB address area, in what is designated as private system memory in Figure 1.3. Finally, the bulk of the public system memory resides above 1 MB. In OS/2 implementations, this extended memory exists from 1 MB to 16 MB.

The 80386 uses double-word arithmetic. The eight general registers are 32-bit: EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP. The prefix E indicates that the familiar 16-bit general registers (AX, BX,...) have simply been extended to 32 bits. In fact, the low-order word of each of these eight registers can be treated as the equivalent 16-bit register with all the reserved name definitions applied to these 16-bit quantities. (The data registers further subdivide into byte-length register halves AH, AL, BH, BL, ...) Clearly, this implies a downward compatibility for running 16-bit microprocessor (8088, 8086, and 80288) code.

The instruction pointer (EIP) and flags register (EFLAGS) have similar downward compatibility features. Finally, there are six segment registers: CS, DS, ES, SS, FS, and GS. The last two are new and provide for additional independent data segment access using overrides. These segment registers are each of word length. In addition to the registers specified, we have the following new system register types (plus the memory-management registers specified above):

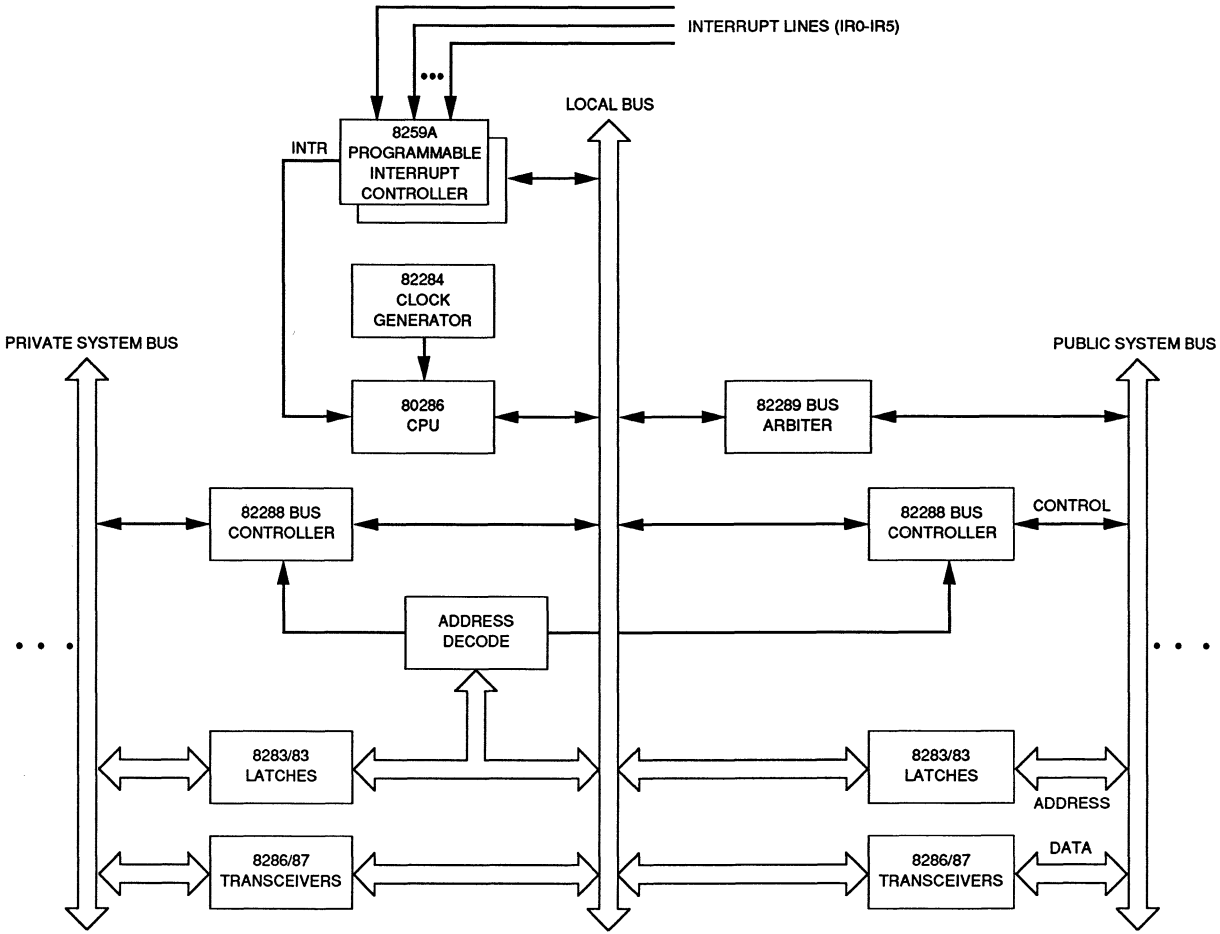


Figure 1.2 Expanded view of the 80286 bus environment.

1. Control registers (four): CR0, CR1, CR2, CR3
2. Debug and test registers (eight): DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7

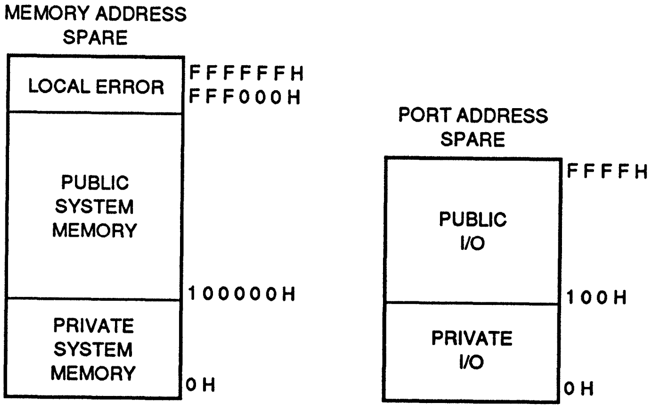


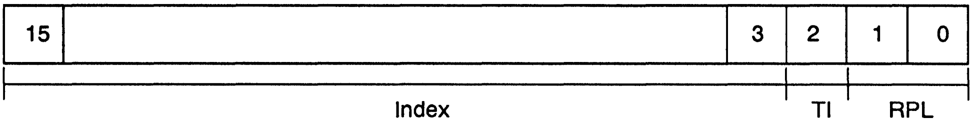
Figure 1.3 Representative memory and port allocation among private and public address spaces.

What does all this mean in terms of OS/2? Basically, the hardware manipulation of addressing under OS/2, and established by the link/locate operation in response to programmed instruction sequences, must occur so that no segment violations take place in system memory. This is the topic of the next two subsections, where operation is described for a Protected Mode installation. We will observe that the 80286 registers are the primary vehicle for ensuring Protected Mode isolation of tasks.

1.1.2 Hardware Operation for Protected Mode

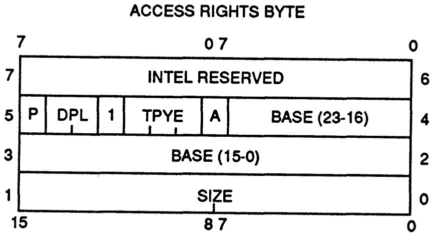
The main system memory of an 80286 system is organized as a sequence of 8-bit addressable quantities called bytes. The addressing spans the range 0 to 2^{20} (1 MB) in Real Address Mode and up to 2^{24} (16 MB) in Protected Mode. In Protected Mode no direct access to physical memory is allowed. The physical address space, for example, is controlled by 24 address pins from the 80286 chip itself. This dictates the 16-MB physical limit. Composition of the address space in Protected Mode, however, indicates a virtual address capability that is much larger. Basically, the 80286 can access a collection of roughly 16,384 linear subspaces or segments each with a maximum size of 64 KB. This translates to a virtual memory size of 2^{30} bytes or 1 gigabyte (GB). The virtual memory allocation must map to physical memory for actual operation, using extended storage where needed. The notion of segmentation as described here allows programs to execute faster and requires less space, than does nonsegmented bulk linear addressing.

How does this protected virtual address mode manage memory? The segment selector is used. A particular segment is uniquely referenced by its selector, a 16-bit address with the following form:



Here TI is the table indicator, which references a global space when set or a local space when zero. The global address space is used for systemwide data and code. The local address space is for general code and data applications such as user tasks. The first two selector bits are the requested privilege level (RPL) bits and relate to protection. This leaves 13 bits, which when coupled with the TI bit allow a segment address space of 2^{14} segments, as discussed above. We will see the impact of protection shortly, but it will be useful, briefly, to explore these segment descriptors further. Note that the descriptor table registers point to tables that provide a complete description of the global address space (GDTR), one or more local address spaces defined dynamically by the LDTR, and an interrupt address space (IDTR).

Within a descriptor table two main classes are recognized: segment descriptors and special-purpose control descriptors. Figure 1.4 illustrates a descriptor. They provide the physical memory base address, segment size, transfer data, and access data. The special-purpose control descriptor is very similar. There is a global and several local descriptor tables as alluded to earlier. (It is these tables that are pointed to using the GDTR and LDTR.)



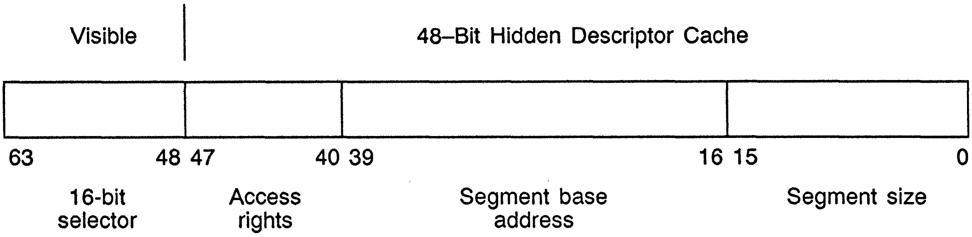
ACCESS RIGHTS BYTE

P = Present
DPL = Descriptor Privilege Level
S = Segment Descriptor
TYPE = Segment Type
A = Accessed

Figure 1.4 80286 segment descriptor.

The 16-bit selector is mapped to a descriptor table entry with its subsequent 24-bit base address. The TI bit determines whether the GDT or a LDT is to be selected. The INDEX field specifies the particular descriptor entry within the chosen table. To get this descriptor entry the processor simply multiplies the index value by 8 in hardware and adds the result to the descriptor table base address.

The segment address translation registers can be depicted as follows:



Here the last 16 bits (bits 48–63) comprise the CS, DS, SS, or ES register values. These bits are the visible portion of the translation register. By loading a segment selector into one of these registers, the program makes the associated segment one of its four currently addressable segments. Note that the definition of the segment base address, the physical address associated with the 16-bit segment selector, must be correlated with the selector by the system software. It is this correspondence between the 16-bit selector and the segment base address that permits virtual addressing to function properly. Both of these addresses, along with the access rights byte and segment size (the translation register contents), permit the correct mapping of virtual memory to physical memory by OS/2, for example. It is here, then, that the algorithms developed in the systems software effect the actual mapping of memory, and the content of this segment address translation register serves as the basis for this mapping.

Figure 1.5 illustrates the protection levels permitted by the 80286 RPL selection. The two bits provide for four levels of protection. In this figure level 0 is the most trusted and level 3 the least. Privilege level is a protection attribute assigned to all segments. Privilege checks are made automatically by the CPU hardware. Programs at level 0 may access data at all other levels, while programs at levels 1–3 may access data only at the same or a less trusted level.

How does OS/2 make use of these levels? Typically, software at level 0 includes services such as memory management, task isolation, intertask communications, and I/O resource control. Level 1 is designated system services and provides functions such as file access scheduling, character I/O, and data communications. Level 2 corresponds to reserved space for customized applications such as database managers, spreadsheets, and word processors, as well as background tasks. Finally, level 3 contains general-purpose user application of the sort written about in the examples in this book. Privilege applies to tasks and affects three different categories of descriptors:

1. Main memory segments
2. Gates (used to change code segments)
3. Task state segments

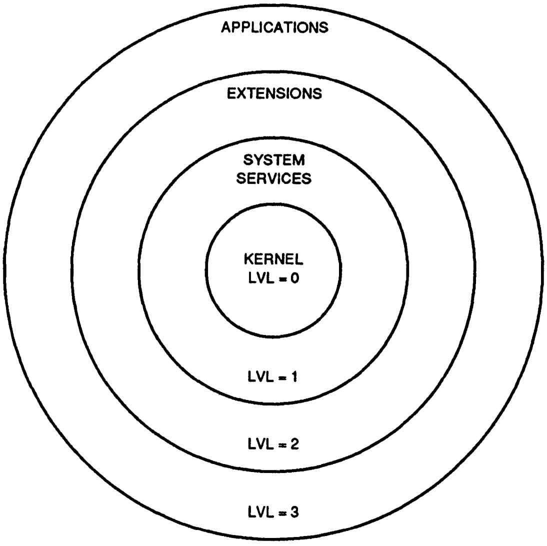


Figure 1.5 Protection-level software allocation and priority, based on level hierarchy, for the Intel 80286.

Descriptor privilege is assigned when the descriptor is created. We have seen, for example, how segment descriptors are formed with their access rights byte and RPL bits controlling protection.

Three kinds of control transfers can occur within a task:

1. Intrasegment transfers
2. Intersegment transfers at the same privilege level
3. Intersegment transfers to different privilege levels

The interlevel transfers must check for access permission and must ensure that a correct entry address is used. To achieve these control transfers, the gates indicated earlier must be used.

A gate is a four-word control descriptor used to redirect a control transfer to a different code segment in the same or a more privileged level or to a different task. There are four types of gates: call gates, task gates, interrupt gates, and trap gates. All four gates define a new destination selector (16-bit), and offset (16-bit), which specifies the physical address to which the transfer is to take place. Call gate descriptors are established for call and jump instructions in the same manner as a code segment descriptor. Task gates specify intertask transfers for the initialization and establishment of child tasks, information exchange, and synchronization. Interrupt gates permit system-level access of low-level hardware-driven interrupt services, and trap gates transfer control to system exception-handling services.

Finally, task state segments are a special control segment defined uniquely for each task. They include the definition of the task address space and execution state. This segment has a special descriptor whose selector is contained in the Task Register. Each task state segment contains 22 words, including the current general-purpose register values and the SS and SP values for each current protection-level stack (there are four such stacks). The descriptor contains the task descriptor privilege level and the usual segment base and limit values. Hence protection mechanisms are extended to intertask control transfers using task state segment descriptors. Task switching is accomplished by loading the Task Register with the new task selector, marking the new task's descriptor type as busy, and setting the Task Switched bit in the Machine Status Word (MSW). This MSW bit signals that the context of the processor extension (80287, for example) may not belong to the current task.

So far we have looked at the 80286 hardware for Protected Mode operational features. These hardware features allow system software such as OS/2 to develop strong protection mechanisms in multitasking environments. Without this protection implementation of the software would necessarily be much more difficult to develop and software protection mechanisms would be required rather than a reliance on register bit monitoring. The room for error in such systems software becomes substantially greater as application complexity increases.

1.1.3 Software Operation for Protected Mode

The flag word for the 80286 is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	NT	IOPL		OF	DF	IF	TF	SF	ZF		AF		PF		CF

where

CF	Carry Flag (set on high-order bit carry or borrow)
PF	Parity Flag (set if low-order byte contains an even number of 1's)
AF	Auxiliary Flag (set on carry or borrow to low-order nibble)
ZF	Zero Flag (set if result 0)
SF	Sign Flag (0 if positive or 1 if negative: high-order bit of result)
TF	Trap Flag (single-step mode)
IF	Interrupt Flag (causes CPU to transfer control to a vector location)

DF	Direction Flag (causes string instructions to auto-decrement when set)
OF	Overflow Flag (set if result is too large)
NT	Nested Task Flag (when set causes a return to the calling task for IRET)
IOPL	IO Privilege Level (specifies current task privilege level)

The Machine Status World (MSW) has the following format:

15		3	2	1	0
	Used by 80386	TS	EM	MP	PE

where

PE	Protected Enable (places 80286 into protected mode)
MP	Monitor Processor (allows WAIT states to be introduced for the 80287)
EM	Emulate Processor (allows the 80287 to be emulated)
TS	Task Switched (allows test to determine if 80287 context belongs to current task)

It is clear that modifications to the flag word (IOPL and NT) permit added software checking for Protected Mode status based on hardware. The MSW provides a software mechanism for checking coprocessor status with regard to the current 80286 program context. In addition to these changes, a number of instructions have been added to the basic Real Address Mode set which reflect privileged and trusted operation. Instructions such as the following fall in this category of enhanced software instruction capability to support Protected Mode.

SMSW	Store MSW
LIDT	Load interrupt descriptor table register
LMSW	Load MSW
CLTS	Clear task switch flag
LGDT	Load global descriptor table register
LLDT	Load local descriptor table register
LTR	Load task register
SGDT	Store global descriptor table register
SLDT	Store local descriptor table register
SIDT	Store interrupt descriptor table register

. . .

These instructions are generally available only to system software. Once an operating system is established (such as OS/2) for the system, privilege-level access must be regulated within the constraints of this system. The user, for example, cannot arbitrarily insert code at privilege level 0. During software implementation under OS/2, privilege-level access is regulated by the OS/2 system software and API service calls for normal application development. This sort of application development is the type treated in this book.

1.2 A BRIEF LOOK AT OPERATING SYSTEM/2

Figure 1.6 presents a simplified OS/2 architecture. This figure reflects the component parts needed to define operating systems software completely [18–21] and illustrates both the multitasking and memory management functions. As with most architectures, Figure 1.6 is a mix between static entities, such as loaders, and dynamic activities, such as intertask communications. The intent of the figure is to portray hierarchical relationships [22–25] for OS/2.

Before we examine the major individual features of the IBM OS/2 implementation, it will be useful briefly to discuss Figure 1.6. This architecture attaches equal importance to the 80286 (or 80386) hardware protection implementation and OS/2 itself. Without the hardware mechanisms OS/2 would be a much bulkier and more sluggish operating system. Hence the Intel chip architecture deserves a substantial amount of credit for yielding an optimized multitasking executive. The dominant mechanism for installing multitask protection is the definition of privilege based on hardware monitoring and checking. This mechanism is put in place in the software during system initialization. The loader and kernel act at level 0 to define the baseline operating system. Device service is handled through the device drivers, which consist of three parts:

1. An initialization routine
2. A strategy routine
3. An interrupt routine

The initialization routine initializes the device by setting the device registers with proper data to establish mode of operation, and this routine also establishes any data structures (buffers and parameter data) needed by the device during operation. The strategy routine receives I/O requests from the kernel and initiates I/O. The Interrupt Service Routine (ISR) is the only OS/2 program privileged to accept and process hardware interrupts (signals on IR0–IR7 of the two 8259A PICs in the IBM PC AT, for example). The device service routines call DevHlp functions in order that the ISR can access application buffers from the kernel. I/O Protection Level checking is continuously monitored to validate what buffers may be accessed by the drivers. Other API functions are used to ensure proper physical memory segmentation, for example, as is the case with screen buffer access.

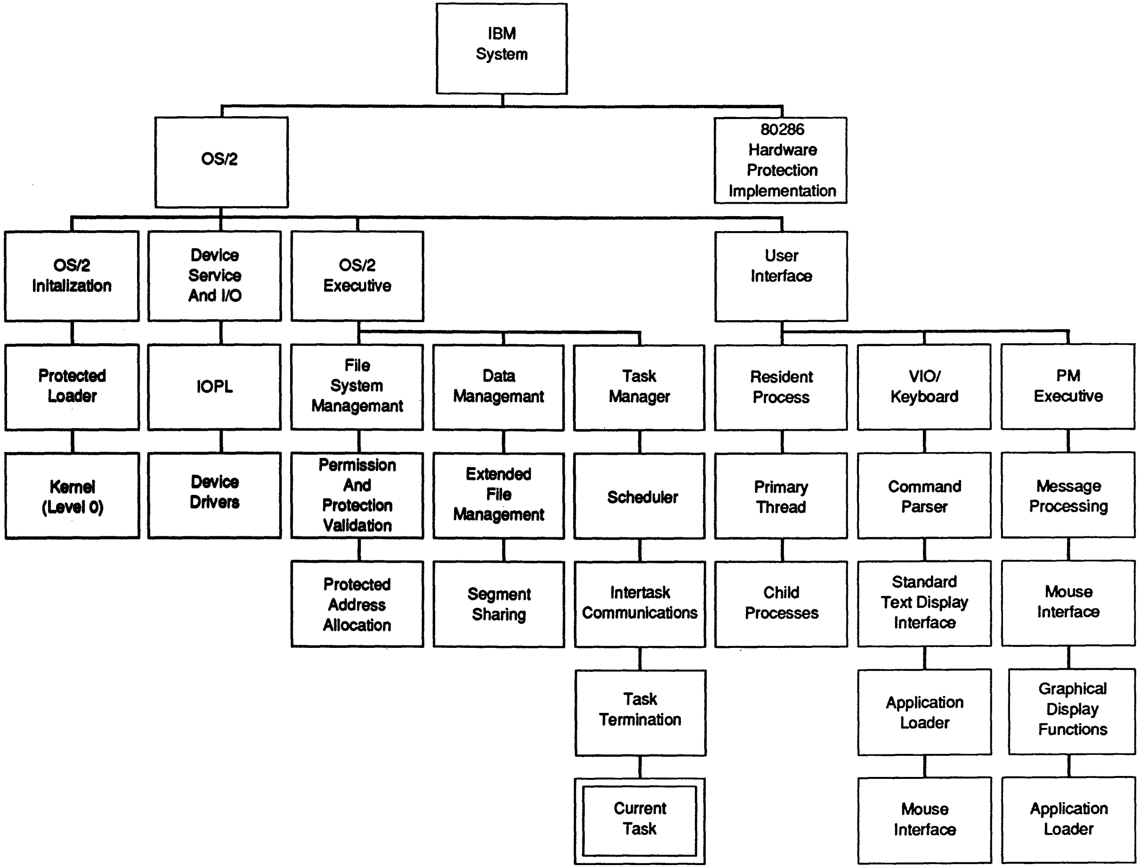


Figure 1.6 IBM OS/2 architecture.

The OS/2 executive is the main processing element of the architecture. This routine contains the core loop, which is the infinite loop sequence representing the idle system state and which may only be affected by a hardware interrupt such as the striking of a key on the keyboard. The executive has four components: file system management, data management, task management, and data segment definition. All objects in the system represent named entities and are subject to file management rules. File permission and protection validation is conducted at a relatively high level in software. At a lower level, the protected address allocation requires privilege-level authority.

Data management is an extension of file system management. This component of the executive ensures that segmentation violations do not occur because proper allocation is implemented. It manages extended files (disk and diskette files, for example). It handles the movement of data in a virtual memory context where the amount of virtual memory required for data, for example, can exceed the total available physical memory in the system. (This is also true for other memory types, such as program memory.) Segment sharing among tasks and other interprocess communication activities involving the access of data segments are managed by this component.

The task manager is the primary dynamic activity element during program execution. We use the terms *thread* and *process* assuming some familiarity with these abstract concepts. Briefly, a thread provides program instructions and data in an execution environment that consists of registers, stacks, and CPU dynamics. Threads do not have system resources allocated to them, but may call or access such resources under the umbrella of a process. Here a process is a collection of threads and system resources allocated to a program. OS/2 task management includes scheduling with a preemptive time-slicing dispatcher. This type of dispatcher is operated on a polling model concept where each thread has a fixed period of time during which it may execute (the time slice) before control is taken away (preempted) and the next thread rolled in and allowed to execute. The process continues in wraparound fashion, and each time slice executes for an approximately equal length of time (at a given privilege level), which is dynamically allocated based on the number of threads active in the system. Task initialization is established by the task manager at level 0 and involves service calls of the type DosCreateThread or DosExecPgm.

Intertask communications typically involve semaphores, pipes, and queues. Also, shared memory segments can be used to pass data among threads and processes. A semaphore is a data structure that OS/2 passes to only one thread at a time. This provides for a rudimentary serialization when two threads, for example, need to access a common data area. OS/2 implements two kinds of semaphores: system and RAM semaphores. System semaphores are used for communication between processes and are implemented by DosCreateSem, for example. RAM semaphores, on the other hand, are used between threads in a single process.

Pipes are buffer areas created by DosMakePipe and are accessed using pipe read and write handles in much the same fashion that conventional file handles are used to access files. To use a pipe to communicate with another process, the pipe is first created and then one of its two handles (read or write) is passed to the second process using a common data structure. In accessing buffers via pipes the data is treated as a continuous stream. An alternative interprocess communications tool is the queue generated by the DosCreateQueue service. Here the data is viewed as an individual collection of finite-length elements that may be separately addressed or accessed. The major feature of the queue is the variability with which data elements may be accessed.

Termination of a communication or task is typically initiated by API calls of the type DosCloseQueue, DosCloseSem, or DosExit. Finally, the task manager is responsible for maintaining all thread parameters on the local thread stack during roll-out by the dispatcher. When a task has been preempted it must maintain pointers to entry points in the code and associated data segments that reflect its current execution position. This is what the task state segment contains, and the task manager is responsible for establishing and maintaining this segment. Data segments must, of course, be defined at the proper privilege level and accessed only with the proper privilege authority. This is also a function of the OS/2 executive.

The user interface represents OS/2s interaction with the outside world. Two visual displays are possible: the VIO or full-screen command mode and the Presentation Manager (PM), a windowed mode where visual data from multiple tasks can be presented. The VIO has a familiar DOS-like screen. The PM is a dynamic display environment capable of simultaneously illustrating multiple program execution through overlapping windows of the type found in the Microsoft Windows program and similar to the Apple MacIntosh display. The PM coordinates activity internally by exchanging messages among tasks and the PM executive. A preferred interface technique is the mouse, which moves a cursor around the screen. When the cursor is placed on a system or menu operation and the mouse single- or double-clicked, the command is executed. This requires an extensive command interpreter based on messages returned to the PM executive.

1.2.1 Protected Mode

In the discussion so far we have alluded to the API service routines. These are the topics of the next section, but it must be emphasized that they are a consequence of the Protected Mode implementation. Before examining the API, however, it is useful to look briefly at some of the Protected Mode operational considerations.

When Real Address Mode is initialized and executes, the boot record causes the code segment register eventually to load with F000 and the instruction pointer to load with FFF0. This address points to a private memory segment that provides 64K bytes of code space for initialization code without changing CS. During initialization of Protected Mode, several registers must also be initialized: Both the GDT and IDT base registers must point to a valid global descriptor table and interrupt descriptor table, respectively. The 80286 next executes the LMSW instruction to set the protected enable (PE) bit and must follow this with an intrasegment JMP (unconditional jump) instruction to clear its CPU instruction queue of instructions decoded in Real Address Mode. To initialize the 80286 registers for the initial Protected Mode state the JMP instruction executes with a selector referring to the initial task state segment address used in the system. This loads the task register, LDTR, segment registers, and remaining general registers with the initial Protected Mode parameter data. The TR should point at a valid task state segment.

We have seen a general description of Protected Mode features. The 80286 mechanisms to protect critical instructions (that affect CPU execution states such as HLT, halt) have three attributes:

1. They involve restricted usage of segments, with the only segments available for use by applications defined by the LDT and GDT.
2. They involve restricted access to segments via privilege.
3. They include privileged instructions or operations that may be executed only at privilege levels determined by the current privilege level and I/O Privilege Level (IOPL).

These mechanisms yield checks that are performed for all instructions and include segment load checks, operand reference checks, and privileged instruction checks. Operand reference errors would include writes to the code segment or segment limit exceeded, for example. Finally, an example of a privileged instruction exception or error would be the execution of an IN or OUT (port input or output instruction, respectively) instruction when the current protection level for the executing task is less trusted than the required IOPL.

Four types of control transfer can occur when a selector is loaded into CS: intersegment with the same privilege level, intersegment to the same or higher-privilege-level interrupt, intersegment to a lower privilege level, and a task switch. We have already briefly considered these transfers, but what are the privilege rules associated with these transfers? The rules are as follows:

1. JMP or CALL direct to a code segment can only be to a segment with descriptor privilege level greater than or equal in privilege to the current privilege level.
2. Interrupts within a task or calls that may change privilege levels can only transfer control through a gate at the same or a less privileged level (than the current privilege level) to a code segment at the same or more privileged level (than the current privilege level).
3. Return instructions that do not switch tasks can only return to a code segment with the same or less trust.
4. A task switch can be performed by a call, jump, or interrupt that references a task gate or task state segment of the same or less trust.

Any violation of these descriptor privilege-level rules will result in exception 13, indicating a segmentation violation.

A task has a current privilege level (CPL) defined by the lower two bits of the CS register (in the selector), as we have seen. CPL can change only when CS changes, using a control transfer through gate descriptors to a new code segment. Tasks begin executing at the CPL specified by the task switch's resulting code segment. Tasks executing at level 0 can access all data segments defined in both the GDT and the task's LDT. Any control transfer that changes CPL within a task requires a change of stacks. Initial values of SS and SP for privilege levels 0, 1, and 2 are maintained in the task state segment. The values for level 3 are defined by the application and affect such transfers as establishing a new thread using DosCreateThread.

These brief remarks about the Protected Mode are intended to put the hardware implementation in perspective. We have seen how privilege-level monitoring, for example, serves as a basis for segment access during execution. The 80286 hardware features include rules and rule checking to maintain privilege integrity, and this preserves proper task operation in a multitasking environment.

Generally speaking, data such as the GDT, initial task state segment, and system services will be located in erasable programmable read-only memory as part of the system build. These are all loaded during the bootstrap process and precede the actual OS/2 executive load and initialization. The foregoing discussion presents the salient 80286 features applicable to a consideration of how OS/2 avoids segment violations. This is the cornerstone for multitasking and memory management. The 80286 Protected Mode attributes are summarized briefly in Table 1.1.

1.2.2 API Services

The Application Program Interface services have been mentioned a number of times so far in the discussion, and Table 1.2 lists these functions (for Version 1.0), indicating whether or not they are conventional API (all the OS/2 services comprise the API, or a subset of the API, the DOS family API, which corresponds roughly to the BIOS and DOS services and runs under the DOS compatibility box as well as the Protected Mode).

The API services are based on the CALL instruction rather than the INT instruction. These API functions act in similar fashion to conventional higher-level-language (HLL) routines with their individual stacks and local parameter spaces. For OS/2 programs written in assembly language, the API service request can be cumbersome. On the other hand, these OS/2 service implementations add an elegance to the resulting code that with just a few exceptions enhances modular development. In a higher-level-language context the API services improve modularity and structure. We will see examples of the use of these services in an assembly language and C context throughout the remainder of this book. Basically, however, the API is most desirable in a HLL environment.

TABLE 1.1 80286 PROTECTED MODE FEATURES

Feature	Discussion
I/O protection	To help manage I/O activities such as setting/clearing interrupts and port read/writes, the 80286 implements an I/O Protection Level (IOPL). This flag defines the minimum protection level at which a program must execute to perform I/O. This provides operating system control of the hardware.
Privilege levels	The 80286 provides for four levels of protection: 1. PL0 (Privilege 0): Most trusted; can access data at levels 0, 1, 2, and 3. 2. PL1 (Privilege 1): Can access data at levels 1, 2, and 3. 3. PL2 (Privilege 2): Can access data at levels 2 and 3. 4. PL3 (Privilege 3): Least trusted; can access data at 3.
Address protection	Through use of LDTs each application program is allocated a private memory space. No other tasks are allowed to enter or use a given task's LDT area. Any common memory elements must be shared using the GDT.
Memory attributes	These attributes are specified in the descriptor table access byte. They include such features as read/write access and descriptor privilege level as well as a flag to indicate execution only (versus addresses associated with variable allocation).

TABLE 1.2 APPLICATION PROGRAMMING INTERFACE ROUTINES

Name	API	FAPI	Description
Tasking			
DosCreateThread	x		Creates asynchronous thread
DosCWait	x		Places current thread in wait state
DosEnterCritSec	x		Disables thread switching
DosExecPgm		x	Allows another program to execute a child
DosExit		x	Issued at completion of execution
DosExitCritSec	x		Reenables thread switching
DosExitList	x		Maintains an exit list for routines
DosGetInfoSeg	x		Returns the address of a data segment
DosGetPrty	x		Gets the priority of the current thread
DosKillProcess	x		Terminates a process
DosPtrace	x		Interfaces to kernel for debugging
DosSetPrty	x		Changes priority of child process
Asynchronous Notification			
DosHoldSignal		x	Changes signal processing

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
DosSetSigHandler		x	Notifies OS/2 of a handler for a signal
Interprocess Communication			
DosCloseQueue	x		Closes a queue
DosCloseSem	x		Closes a semaphore
DosCreateQueue	x		Creates a queue
DosCreateSem	x		Creates a semaphore
DosFlagProcess	x		Allows a process to set an "event" flag
DosMakePipe	x		Creates a pipe
DosMaxSemWait	x		Blocks until semaphore clears
DosOpenQueue	x		Opens queue
DosOpenSem	x		Opens semaphore
DosPeekQueue	x		Examines element in queue
DosPurgeQueue	x		Purges a queue
DosQueryQueue	x		Finds the size of a queue
DosReadQueue	x		Reads an element from a queue
DosResumeThread	x		Restarts a thread
DosSemClear	x		Clears a semaphore
DosSemRequest	x		Obtains a semaphore
DosSemSet	x		Sets a semaphore
DosSemSetWait	x		Blocks a thread until a semaphore
DosSemWait	x		Waits for a semaphore to clear
DosSuspendThread	x		Temporarily suspends thread execution
DosWriteQueue	x		Adds an element to a queue
Timer			
DosGetDateTime		x	Gets the current date/time
DosSetDateTime		x	Sets the date/time
DosSleep		x	Suspends the current thread
Memory Management			
DosAllocSeg		x	Allocates a segment of memory
DosAllocShrSeg	x		Allocates a shared segment
DosAllocHuge		x	Allocates multiple memory segments
DosCreateCSAlias		x	Creates a code segment descriptor
DosFreeSeg		x	Reallocates a memory segment
DosGetHugeShift		x	Returns a shift count for deriving selectors

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
DosGetShrSeg	x		Accesses shared memory
DosGetSeg	x		Accesses shared memory
DosGiveSeg	x		Yields shared access to another process
DosLockSeg	x		Locks a discardable segment
DosMemAvail	x		Returns size of largest free block
DosReallocHuge		x	Changes huge memory size
DosReallocSeg		x	Changes segment size
DosSubAlloc		x	Allocates from a previous allocated segment
DosSubFree		x	Frees from a previous allocated memory
DosSubSet		x	Initializes a segment
DosUnlockSeg	x		Unlocks a discardable segment
Dynamic Linking			
DosFreeModule	x		Frees a dynamic link module
DosGetModHandle	x		Returns handle for dynamic link module
DosGetModName	x		Returns pathname for dynamic link module
DosGetProcAddr	x		Returns FAR procedure address
DosLoadModule	x		Loads a dynamic link module
DosGetMachineMode		x	Returns current CPU mode
BadDynLink		x	Error on dynamic link
Device Monitors			
DosMonClose	x		Terminates character device monitoring
DosMonOpen	x		Accesses a character device
DosMonRead	x		Moves data
DosMonReg	x		Establishes I/O buffer
DosMonWrite	x		Writes to the monitor's buffer
Session Management			
DosStartSession	x		Starts a session
DosStopSession	x		Stops a session
DosSelectSession	x		Allows a parent to switch to a child
DosSetSession	x		Sets child session status

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
Device I/O Services			
DosBeep		x	Beeps speaker
DosCLIAccess	x		Requests privilege for enabling/ disabling interrupts
DosDevConfig		x	Gets information about attached devices
DosDevIOCtL		x	Sets up control functions for a specified device
DosGetPID		x	Returns current process ID
DosPFSActivate	x		Specifies the code page and foot to make active
DosPFSCloseUser	x		Indicates the spool file is closed
DosPFSInit	x		Allows initialization of the code page and font
DosPFSQueryAct	x		Queries the active code page and font
DosPFSVerifyFont	x		Indicates validity for the specified code page and font
DosPhysicalDisk	x		Obtains disk information
DosPortAccess	x		Requests or releases port I/O privilege
DosSendSignal	x		Sends a Ctl/c or Ctl-Break to process
KbdDeRegister	x		Deregisters a keyboard
KbdCharIn		x	Reads a character
KbdClose	x		Ends the existing logical keyboard
KbdFlushBuffer		x	Clears the keyboard buffer
KbdFreeFocus	x		Frees the logical to physical keyboard bond
KbdGetCp	x		Allows access to the current code page
KbdGetFocus	x		Binds the logical to physical keyboard
KbdGetStatus		x	Gets the state of the keyboard
KbdOpen		x	Creates a new logical keyboard
KbdPeek		x	Returns the last character without clearing the keyboard buffer
KbdRegister	x		Registers a keyboard
KbdSetCp	x		Sets the code page
KbdSetCustXt	x		Installs a code page and calling handle

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
KbdSetFgnd	x		Raises the priority of the foreground keyboard's thread
KbdSetStatus		x	Sets the keyboard characteristics
KbdStringIn		x	Reads a character string
KbdSynch	x		Synchronizes access for a keyboard to device driver
KbdXlate	x		Translates scan codes to ASCII
MouClose	x		Closes the mouse driver
MouDeRegister	x		Deregisters a mouse device
MouDrawPtr	x		Opens a mouse pointer image to the mouse
MouFlushQue	x		Empties the mouse queue
MouGetDevStatus	x		Returns status flags for the mouse driver
MouGetEventMask	x		Returns event mask for mouse
MouGetNumButtons	x		Returns number mouse buttons supported
MouGetNumMickeys	x		Returns number of mouse movement units per centimeter
MouGetNumQueEl	x		Returns status for mouse device drive event queue
MouGetPtrPos	x		Gets row and column position of mouse
MouGetPtrShape	x		Gets the pointer shape
MouGetScaleFact	x		Gets the scaling factors for the mouse
MouInitReal	x		Initializes the DOS mode mouse
MouOpen	x		Opens the mouse device
MouReadEventQue	x		Reads an event from the mouse device event queue
MouRegister	x		Registers a mouse
MouRemoves Ptr	x		Clears a pointer area from mouse use
MouSetDevStatus	x		Sets mouse status
MouSetEventMask	x		Assigns a new event mask
MouSetPtrPos	x		Resets the row and column position for the mouse
MouSetPtrShape	x		Sets the mouse shape
MouSetScaleFact	x		Assigns the mouse a new pair of scaling factors
MouSynch	x		Synchronizes the mouse
VioDeRegister	x		Deregisters a video subsystem
VioEndPopUp	x		Closes a temporary screen

TABLE 1.2 (*Continued*)

Name	API	FAPI	Description
VioGetAnsi	x		Returns the current ANSI ON/OFF state
VioGetBuf	x		Returns the address of the logical video buffer
VioGetCp	x		Allows a query of the code page
VioGetConfig		x	Returns the display configuration
VioGetCurPos		x	Returns the cursor position
VioGetCurType		x	Returns the cursor type
VioGetFont		x	Returns font
VioGetMode		x	Returns display mode
VioGetPhysBuf		x	Gets addressability to physical display buffer
VioGetState		x	Gets display state
VioModeUndo	x		Changes mode
VioModeWait	x		Allows notification when display must be restored
VioPopUp	x		Allocates a temporary screen
VioPrtSc	x		Copies the screen to printer
VioPrtScToggle	x		Called when Ctrd-PrtSc is entered
VioReadCellStr		x	Reads character-attribute pairs (cells) from screen
VioReadCharStr		x	Reads a character string from the display
VioRegister	x		Registers an Alternate Video subsystem
VioSaveRedrawUndo	x		Cancels a VioSavRedrawWait
VioSavRedrawWait	x		Notifies a redraw must be performed
VioScrLock		x	Locks the physical display
VioScrollDn		x	Scrolls down
VioScrollUp		x	Scrolls up
VioScrollLf		x	Scrolls left
VioScrollRt		x	Scrolls right
VioScrUnLock		x	Unlocks the physical display
VioSetAnsi	x		Activates or deactivates ANSI support
VioSetCp	x		Sets the code page
VioSetCurPos		x	Sets the cursor position
VioSetCurType		x	Sets the cursor type
VioSetFont		x	Downloads a display font
VioSetMode		x	Sets display mode
VioSetState		x	Sets the display state

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
VioShowBuf	x		Updates the physical display with the logical
VioWrtCellStr		x	Writes a string of character-attribute cells to display
VioWrtCharStr		x	Writes a character string to the display
VioWrtCharStrAtt		x	Writes a repeated attribute string to the display
VioWrtNAtt		x	Writes an attribute M times to the display
VioWrtNCell		x	Writes a cell M times to the display
VioWrtNChar		x	Writes a character M times to the display
VioWrtTTY		x	Writes a character string to the display
File I/O			
DosBufReset		x	Flushes a requesting process cache buffer
DosChDir		x	Defines the current directory
DosChgFilePtr		x	Moves the read/write pointer
DosClose		x	Closes a file handle
DosDelete		x	Removes a directory entry
DosDupHandle		x	Returns a new file handle for an open file
DosFileLocks		x	Locks and unlocks a range in an open file
DosFindClose		x	Closes the association between directory handles and search functions
DosFindFirst		x	Finds the first set of names that match a directory specification
DosFindNext		x	Locates the next set of matching directory entries
DosMkDir		x	Creates specifies directory
DosMove		x	Moves a file
DosNewSize		x	Changes a file size
DosOpen		x	Opens a file
DosQCurDir		x	Gets full path name for current directory
DosQCurDisk		x	Gets the current default drive
DosQFHandState		x	Queries the state of the specified files

TABLE 1.2 *(Continued)*

Name	API	FAPI	Description
DosQFileInfo		x	Returns information for a specific file
DosQFsInfo		x	Queries information from a file system device
DosQHandType		x	Determines whether a handle references file/device
DosQVerify		x	Returns the value of the verify flag
DosRead		x	Reads from a file to a buffer
DosReadAsync	x		Transfers from a file to a buffer, asynchronously
DosRunDir		x	Removes a subdirectory
DosScanEnv	x		Searches an environment for a value
DosSearchPath	x		Searches a path for a filename
DosSelectDisk		x	Specifies the default drive
DosSetFHandState		x	Sets the state of a file
DosSetFileInfo		x	Specifies information for a file
DosSetFileMode		x	Changes the attributes of a file
DosSetFsInfo		x	Specifies information for a file system device
DosSetMaxFH	x		Defines a maximum number of file handles
DosSetVerify		x	Sets a verify switch
DosWrite		x	Transfers from a buffer to a file
DosWriteAsync	x		Transfers from a buffer to a file, asynchronously
Errors and Exceptions			
DosErrClass		x	Returns error code options
DosError		x	Allows the disabling or user notification on errors
DosSetVac		x	Allows address registration for machine exceptions
Messages			
DosGetMessage		x	Retrieves a message from a message file
DosInsMessage		x	Inserts text into message body
DosPutMessage		x	Outputs a message
Trace/Program Startup			
DosGetEnv		x	Returns a pointer to the environment string

TABLE 1.2 *(Concluded)*

Name	API	FAPI	Description
DosGetVersion		x	Returns the OS/2 version number
Code Page Support			
DosGetCp		x	Gets the current code page
DosSetCp		x	Sets the current code page
DosSetProcCp	x		Sets the current code page
Country Support			
DosCaseMap		x	Case maps country codes to a binary string
DosGetCollate		x	Obtains country information
DosGetCtryInfo		x	Obtains country information
DosGetDBCSEv		x	Obtains country environment vector

Most microcomputer system software involves intersegment references between segments contained in the program file obtained from the linker, the .EXE file in DOS. This reference mechanism is referred to as static linking because it is implemented prior to run-time loading. Loading merely brings the segments into memory and modifies fix-up points to reflect the correct intersegment references.

OS/2 allows the loader (not the linker) to reference segments included in special dynamic-link libraries (DLL). The entire APT is based on DLL programming. How does a dynamic-link reference function? Basically, any program can reference DLL routines by indicating that they are externally defined (using the EXTRN pseudo-op, for example, in an assembler program). At link time the system matches external references with other object modules (.OBJ files) and libraries (.LIB files) specified. Since the DLL routines are an .EXE file and suitable for run-time loading, they do not fall in the .OBJ or .LIB category. A new type of library file is required, the dynamic-link definition library file. This file simply satisfies the external reference by indicating to the loader the location of the DLL routine involved. At run time the loader then adds the DLL code from storage to the executable module.

The API call interface employs dynamic linking. The major advantages to this approach are that:

1. The API code can easily be modified at the system level
2. The API call can be satisfied with in-line code instead of the DLL code if desired by proper loading
3. The API can include some services not essential to kernel-level privilege, and these services can be implemented with less protection
4. The API call is direct, not via vector table routing
5. The API library can easily be expanded

In Chapter 2 we begin to develop the programming techniques needed to access properly the services outlined in Table 1.2.

1.2.3 Memory Management

OS/2 provides a significant memory management capability by using the hardware features of the 80286, together with its system architecture. OS/2 provides the capability to move segments around and free memory in response to DLL requirements. Also, using the P bit of the descriptor, OS/2 can determine when a referenced segment is needed and dynamically roll these segments in or out of memory from extended storage, in response to program execution. Such segment swapping is the basis for allowing large-scale access to the virtual address space in a given physical memory implementation. Provision exists to:

1. Create or close new segments
2. Create or close huge segments (greater than 64 KB)
3. Suballocate segments

This corresponds to the demand loading philosophy, which OS/2 supports, and allows dynamic reallocation and subdivision of memory in response to changing requirements.

1.2.4 Multitasking

Just as memory management has been addressed earlier in the chapter, multitasking has been covered in Sections 1.1 and 1.2.1. We have mentioned the notion of threads (a dispatchable unit), processes (a collection of threads and system resources), and a session is a collection of processes run in a virtual context. Under OS/2, for example, a given element of program code comprises a thread's executable context. This may run as multiple instances in which multiple copies of the thread are executed as individual tasks, each task running the same code. Based on this interpretation the meaning of an instance is clear: an executing entity dynamically different from all others.

The reentrant nature of OS/2 threads requires that if multiple threads access the same data block, the threads must synchronize access to this data. This synchronization can be accomplished using a number of OS/2 features already discussed (semaphores, queues, pipes, flags, and shared memory). Interprocess communication requires the use of these facilities, and threads desiring to access such common data blocks must serialize their access. In general, when no common access between processes is required, OS/2 will asynchronously execute the processes in a multitasking situation.

A simple example of process synchronization is presented in Chapter 2, where a common data area (shared segment) is established using DosAllocShrSeg and the first few bytes are used to establish a handshake. The creating process sets the flag byte to zero and turns on the child process, which also has access to the segment. Once the child process completes its generation of data (to be used by the parent), it sets the flag to 1. The parent, sensing a 1, then accesses the segment.

Semaphores, pipes, and queues have much the same functional behavior except that they represent tools specifically designed for interprocess exchanges. These OS/2 objects represent a formal extension of interprocess communications (compared with the shared memory flag above, for example). DosSetSigHandler and DosFlagProcess are examples of formal flag implementation services. We examine those resources in later chapters.

Processes are created with the API service DosExecPgrm, as we shall see in Chapter 2. They are hierarchical in that the creating process serves as the parent, with the created process the child. The API DosKillProcess can be used to terminate a child process. At creation a process can be established asynchronously, during which the parent continues to execute in normal time-slice fashion, or synchronously, where the parent is suspended until the child completes execution. When a thread is created it assumes the priority level of its creator. Using DosSleep a thread may stop execution for a fixed period. During this period the thread is not allowed to access system resources.

We have considered dynamic linking, in which a DLL is created and an associated definition file containing pointers to the DLL entries. At run time the definition file has already been linked with the main calling routine, so the loader simply brings the DLL into memory and completes its entry-point fix up. A second type of dynamic linking exists called run-time dynamic linking. In the latter procedure the API DosLoadModule can actually be used to load a DLL after execution begins. The difference between run-time dynamic linking and load-time dynamic linking is that loading the DLLs and entry point fix-ups can occur after execution begins in the former if needed, whereas they must occur during loading in the latter.

Finally, we look briefly at input and output (I/O) in the privileged multitasking environment. I/O occurs from level 2, whereas applications execute from level 3; hence OS/2 must build a call gate for access to segments that accomplish I/O– I/O-protected segments (IOPS). Such segments are created by the loader, and typically, API calls such as DosOpen or DosClose establish generation of an IOPS (see Table 1.2).

1.2.5 Version 1.0 and 1.1 Differences

Earlier we saw the API functions described (Table 1.2). In the IBM OS/2 Standard Edition 1.0 these functions comprised the bulk of the services afforded by OS/2 and were intended for use by programmers desiring to access these services. The Toolkit routines (reference 7) provide a collection of include files (for both C and assembler) that make use of the API services relatively easy.

With the development of Standard Edition 1.1 (aside from some relatively minor enhancements) the addition of the Presentation Manager (PM) graphical interface, and its associated 300 plus function library, is the major improvement over Version 1.0. Essentially, OS/2 under Version 1.0 employs a DOS-like full-screen command mode for the user interface. This display mode is capable of addressing only one screen at a time. Under the PM a Windows-like interface is presented and each executing context can be visualized simultaneously as part of a sequence of windows occupying the screen.

It is programming of the OS/2 PM that constitutes the major enhancement of Version 1.1. This programming employs techniques similar to those outlined in the Windows Software Development Kit (SDK) [26–28] for development of Windows programs.

1.3 THE OS/2 PRESENTATION MANAGER

It is worthwhile to look briefly at the Presentation Manager (PM) to get a feeling for how this type of interface is implemented. The PM runs as an executive subset under OS/2. IBM has developed the Systems Application Architecture (SAA) and the PM implements the Common Programming Interface (CPI) component of SAA, which makes portability to other SAA-supported environments (such as VM and MVS on the System/370 and Operating System/400 on the Application System/400) relatively straightforward.

Communications and network-intensive applications are not generally amenable to the SAA without additional software support. The Extended Edition Version 1.1, for example, is intended for these more uniquely hardware-specific applications. Examples include airline reservation systems, bank transaction processing, some large-scale process control applications, real-time processing, and communications front-end (physical layer) processing.

The PM interacts with the OS/2 user via a graphical user interface [30–35]. By graphical user interface we mean the screen appearance when the PM is invoked. This display is illustrated in Figure 1.7 with a typical pulldown menu. The maximize/minimize buttons can be used to reduce the contents of the client area to an icon. This icon can be restored using the mouse. The client area contains the visible portion of the display context, which presents the active window interface. It is here that the executing program displays its particular graphical context. The PM allows

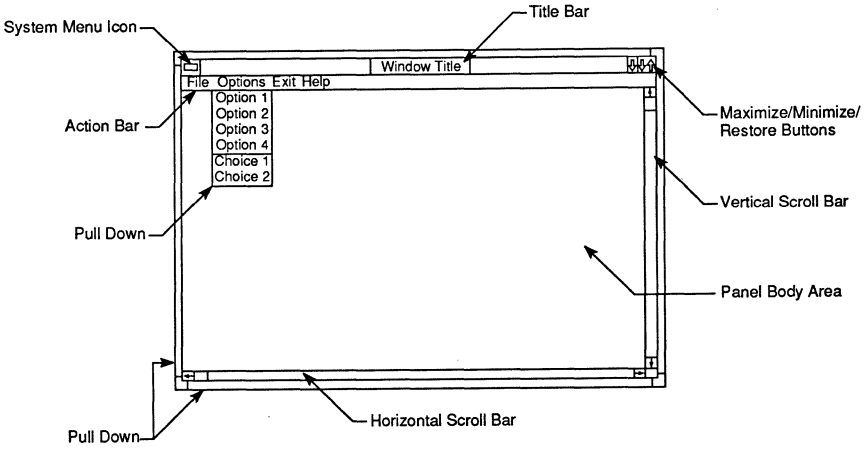


Figure 1.7 The Presentation Manager standard window.

the client area to be subdivided into tiled (windows adjacent to each other) or overlapped (windows lying on top of each other with varying offsets) windows. This facilitates a partial display of the contents of several windows simultaneously.

In addition to the features illustrated in Figure 1.7, the programmer can call up modal and modaless dialog boxes and message boxes. These can be used to achieve I/O in the PM context. A modal dialog box retains control of the execution until it is destroyed (usually by clicking the mouse over a termination panel). A modaless dialog box allows the PM to permit windows in other applications to be activated after it has been created. A message box is a predefined dialog window available to all applications for displaying text and receiving user I/O.

The PM has a strong graphics capability (as differentiated from graphical interface) with which computer-generated graphics may be displayed in the client area. This Graphics Program Interface (GPI) employs API calls beginning with Gpi. The PM also has a clipboard that can be used to hold intermediate data and resources (such as metafiles and bitmaps). A metafile defines the contents of a windowed picture so that it can be used by other applications. These metafiles are created using GPI calls and conform to the Mixed Object Document Content Architecture (MODCA) interchange standard. A bitmap, on the other hand, is a representation in memory of data displayed on an all-points-addressable basis and requires that the object in question be capable of being specified in this mode.

Finally, the programming for the interface itself employs a number of new library elements. The PM executive is a dynamic program that is constantly accessing each application context for changes and conversing with the application via a stream of messages. When an application executes various window functions, for example, the function causes specific messages to be sent to the PM executive. These are then interpreted and the executive generates a response.

The general PM program flow of activity is illustrated in Figure 1.8, where termination of the window is accomplished by the executive in response to a WM_QUIT message. This flowchart shows the setup code as distinct from the message-processing loop, as it is. C is the language of choice for programming the PM executive.

Conventional C programs have a basic template that appears as

```
Preprocessor
main()
   {
   ...
   {
function1()
      }
   ...
      }
   ...
functionN()
   {
   ...
   {
```

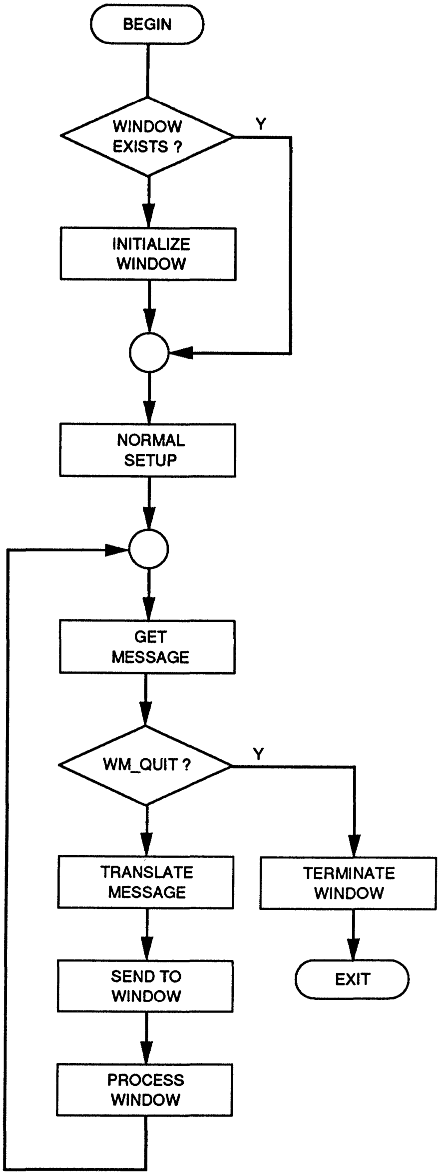


Figure 1.8 Dynamic picture of a Windows application, similar to the PM implementation.

Each function is callable internal to either main ( ) or another (group of) function(s). A simple PM program with one window might have a template of the following form:

```
Preprocessor
void cdecl main(argc,argv)
  {
  -code to initialize window
  -loop to continuously read messages sent from the executive
  }
"window function
  {
  - this function directs execution to appropriate PM func-
    tions based on "message" input from the PM executive
   }
"initialization functions"
  {
  -functions needed to initialize the first, additional, and
   every instance of a window
   }
  ...
  other needed user-defined functions
  ...
```

Figure 1.9 illustrates a Structure Chart for the upper hierarchical levels of a PM application. This chart is generic in the sense that it only indicates entities that are common to all PM programs. The reader familiar with the Microsoft Windows executive will see a close parallel between programming for this executive and programming the PM executive [29].

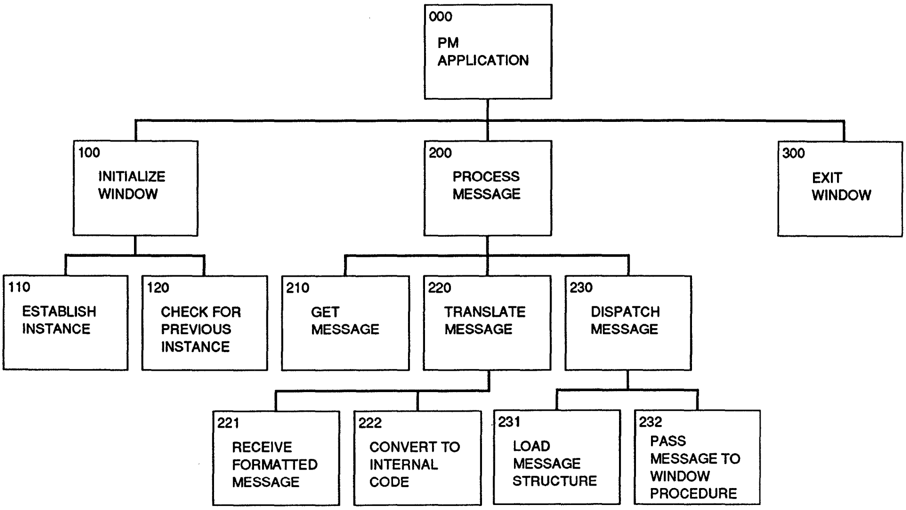


Figure 1.9 Generic Structure Chart for the upper hierarchical levels of a Presentation Manager application.

1.4 SUMMARY

In this chapter we have introduced the IBM Operating System/2 in the framework of the Intel 80286 and 80386 CPUs. Initially, the CPU registers were described and the Protected Mode address space examined. The Protected Mode provides a framework from which to perform memory management and multitasking because of the hardware interlocks into segment management. Basically, the access rights control byte in the segment address translation register word determines what data and control segments will have access to a given memory location.

This segment control, then, is the mechanism by which the hardware delimits Protected Mode access. This is applied to the software via the operating system (and the local descriptor tables (LDTs) and the global descriptor table (GDT)). OS/2 provides system services similar to the BIOS and DOS interrupt services via the family Applications Programming Interface (FAPI). The FAPI is a subset of the more complete API functions, which represent a complete set of Protected Mode services. Representative of these services are the following categories

1. Mouse (Mou)
2. Video (Vio)
3. DOS (Dos)
4. Graphical (Gpi)
5. Spool (Spl)
6. Device (Dev)
7. Keyboard (Kbd)
8. Window (Win)

The API calls, then, allow access to system hardware and file services under OS/2. OS/2 has provision to add devices to the system by creation of additional device drivers and input/output privilege level (IOPL) is assisted using the Dev and Dos services.

The Presentation Manager (PM) represents the Version 1.1 user-friendly graphical interface for OS/2. Under Version 1.1 the user also has a choice of the full-screen command prompt interface mode which is that employed by Version 1.0. The PM display is similar to that used by Microsoft Windows Version 2.0 and provides for overlapped (or tiled) window presentation of active process information in a multitasking environment. The PM executive interacts dynamically with the executing programs. Associated with the PM are a large class of functions (window functions) used to regulate the interface under program control. The messages exchanged between the PM executive and the program are continuous and dynamically varying. Hence this executive provides a time-varying interactive display that can be updated and controlled using the mouse. It is very similar to the interface provided by the Apple MacIntosh operating system.

REFERENCES

1. *Disk Operating System Version 3.30 Reference,* International Business Machines Corporation, Boca Raton, FL, 1986.
2. *iAPX 86/88, 186/188 User's Manual: Programmer's Reference,* Intel Corporation, Santa Clara, CA, 1986.
3. *iAPX 286: Programmer's Reference Manual,* Intel Corporation, Santa Clara, CA, 1985.
4. *80386 Programmer's Reference Manual,* Intel Corporation, Santa Clara, CA, 1986.
5. *IBM Operating System/2 Standard Edition: User's Reference,* International Business Machines Corporation, Boca Raton, FL, 1987.
6. *IBM Operating System/2 Technical Reference,* Vols. I and II, International Business Machines Corporation, Boca Raton, FL, 1988.
7. *IBM Operating System/2 Programmer's Toolkit,* International Business Machines Corporation, Boca Raton, FL, 1987.
8. *Microsoft Operating System/2 Software Development Kit,* Microsoft Corporation, Redmond, WA, 1987.
9. *IBM Operating System/2 Technical Reference Version 1.1: Programming Reference,* Vols. 1, 2, and 3, International Business Machines Corporation, Boca Raton, FL, 1988.
10. Petzold, C., Presentation Manager Dialog Procedures, *PC Magazine*, September 1988, p. 302.
11. Malloy, R., IBM's OS/2 Extended Edition, *Byte Magazine*, July 1988, p. 111.
12. *Microsoft C 5.1 Optimizing Compiler Reference Manual*, Microsoft Corporation, Redmond, WA, 1987.
13. *IBM Macro Assembler/2 Reference Manual,* International Business Machines Corporation, Boca Raton, FL, 1987.
14. VEDIT PLUS Reference Manual, CompuView Products, Inc., Ann Arbor, MI, 1986.
15. Petzold, C., Introducing the OS/2 Presentation Manager, *PC Magazine,* July 1988, p. 379.
16. *IBM Technical Reference Personal Computer AT,* International Business Machines Corporation, Boca Raton, FL, 1984.
17. *IBM Personal System/2 Model 50 and 60 Technical Reference,* International Business Machines Corporation, Boca Raton, FL, 1987.
18. Tanenbaum, A. S., *Operating Systems: Design and Implementation,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987, p. 36.
19. Davis, W. S., *Operating Systems: A Systematic View,* Addison-Wesley Publishing Company, Reading, MA, 1987, p. 73.
20. Beck, L. L., *System Software: An Introduction to Systems Programming,* Addison-Wesley Publishing Company, Reading, MA, 1985, p. 302.
21. Katzau, H., *Operating Systems: A Pragmatic Approach,* Van Nostrand Reinhold Company, New York, 1986, p. 161.
22. Godfrey, J. T., *IBM Microcomputer Assembly Language: Beginning to Advanced,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989, p. 423.
23. Letwin, G., *Inside OS/2,* Microsoft Corporation, Redmond, WA, 1988, p. 39.

24. Iacobucci, E., *OS/2 Programmer's Guide*, Osborne McGraw-Hill, Berkeley, CA, 1988, p. 51.
25. Krantz, J.I., Mizell, A.M., and Williams, R.L., *OS/2: Features, Functions, and Applications,* John Wiley & Sons, Inc., New York, 1988, p. 12.
26. *Microsoft Windows Software Development Kit: Quick Reference Programming Guide,* Microsoft Corporation, Redmond, WA, 1987.
27. *Microsoft Windows Software Development Kit: Update/Programmer's Utility Guide,* Microsoft Corporation, Redmond, WA, 1987.
28. *Microsoft Windows Software Development Kit: Programmer's Reference,* Microsoft Corporation, Redmond, WA, 1987.
29. Godfrey, J.T., *Applied C: The IBM Microcomputers,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990 p. 356.
30. *IBM Operating System/2 Programmer's Toolkit Version 1.1: Programming Overview,* International Business Machines Corporation, Boca Raton, FL, 1988.
31. *IBM Operating System/2 Programmer's Toolkit Version 1.1: Programming Guide,* International Business Machines Corporation, Boca Raton, FL, 1988.
32. *IBM Operating System/2 Programmer's Toolkit Version 1.1: Building Programs,* International Business Machines Corporation, Boca Raton, FL, 1988.
33. *IBM Operating System/2 Technical Reference Version 1.1: I/O Subsystems and Device Drivers*, Vols. 1 and 2, International Business Machines Corporation, Boca Raton, FL, 1988.
34. *IBM Operating System/2 Technical Reference Version 1.1: Macro Assembler/2 Bindings Reference,* International Business Machines Corporation, Boca Raton, FL, 1988.
35. *IBM Operating System/2 Technical Reference Version 1.1: C/2 Bindings Reference,* International Business Machines Corporation, Boca Raton, FL, 1988.

PROBLEMS

1.1 In Real Address Mode assume a CS register value of 07F8H and an IP register value of 274AH. What is the 20-bit physical address?

1.2 Does the fact that OS/2 is a multitasking operating system imply that it is a multiprocessor operating system, as well?

1.3 What is the largest fixed-point value that the 80386 can accommodate? Largest signed fixed-point value?

1.4 The exit processing for OS/2 is via a call to DOSEXIT rather than a RET instruction. If the requisite processing for DOSEXIT is

```
EXTRN  DosExit:FAR
...
PUSH  WORD ActionCode ;Indicates end thread or process
PUSH  WORD ResultCode ;Result Code
CALL  DosExit
```

define a macro

```
@DosExit  action,  result
```

that can be used to setup and execute the exit operation.

1.5 The video screen unlock processing for OS/2 is via a call to VIOSCRUNLOCK. Assuming that the requisite processing for this call is

```
EXTRN  VioScrUnLock:FAR
...
PUSH  WORD  VioHandle  ;Video  handle
CALL  VioScrUnLock
```

define a macro

```
@VioScrUnLock  viohdl
```

that can be used to setup and execute the unlock operation.

1.6 The video screen lock processing for OS/2 is via a call to VIOSCRLOCK. Assuming that the requisite processing for this call is

```
EXTRN  VioScrLock:FAR
...
PUSH  WORD  WaitFlag   ;Block  or  not
PUSH  BYTE  Status     ;Lock  status  returned  (address)
PUSH  WORD  VioHandle  ;Video  handle
CALL  VioScrLock
```

where PUSH@ means to push an address on the stack, define a macro

```
@VioScrLock  waitf,iostat,viohdl
```

that can be used to setup and execute the lock operation.

1.7 What are the three principal features that the OS/2 Standard Edition contributes over conventional DOS operating system characteristics?

1.8 While 80286 code (source) will run on 80386 systems, why will 80386 applications code generally not run on 80286 systems?

1.9 In the IBM PC AT, 16 levels of hardware interrupts are available to the system user. How many 8259As are required to support this number of interrupt levels?

1.10 The DOS partition in the IBM microcomputer environment supports the first 1 MB of addressable memory. Why do most early systems allow a maximum of only 640 KB of program memory access? Where does OS/2 extended memory reside?

1.11 What is the difference between physical and virtual memory, and how is virtual memory managed?

1.12 How does OS/2 differentiate system memory space from applications memory space? How much virtual memory space is accessible by applications?

1.13 Why would data communications processing not reside at level 0 to ensure that no data is lost during a communications session?

1.14 How does the 80286 CPU know that the system is to operate in Protected Mode?

1.15 When writing a device driver, mixed-language programming is probably an optimum approach. Assuming that a driver is written using a combination of C and assembly language, what parts are likely candidates for assembler code? What parts are likely candidates for C code?

1.16 Explain the major difference between a pipe and a queue.

1.17 Would you say that the API implementation represents a favorable step for assembly language programming of OS/2? For C programming of OS/2? Explain.

1.18 If two threads from the same process need to access a common data area, will they run synchronously or asynchronously? If the threads are from different processes, will they access in synchronous or asynchronous fashion?

1.19 What is the thread equivalent to DosKillProcess? How does it differ from the activity for a process?

1.20 Can we use the Gpi services with full-screen command mode to generate screen graphics under CGA, for example?

1.21 Which is preferred in a multitasking environment: modal or modaless dialog box implementation?

PART II
Programming OS/2 Using Assembler

2 Introductory OS/2 Assembler Programming

OS/2 is a unique program environment devoid of the normal interrupt calls found in conventional assembly language programs. In their place OS/2 implements Application Program Interface (API) function calls, which provide the programmer with access to system services. Specific services include the familiar DOS BIOS and INT21H function calls, an enhanced set of video display handlers, mouse services, and keyboard handlers. These are the most obvious extensions of OS/2, and they permit the user a vastly increased capability to develop multitasking modules and extend program usage beyond the normal 64K segment limit.

In this chapter we examine assembly language programming in the context of OS/2 [1,2]. The goal of the exposition is to provide the reader with examples of the usage of assembly language in the OS/2 framework. This is not a treatise on how to program assembler; rather, we hope to achieve an understanding of the OS/2 interface.

2.1 OS/2 SERVICES: ACCESSING THE API

A great deal of the new programming emphasis using OS/2 is the API services which are contained in the IBM (or Microsoft) supplied library, API.LIB. The services contained in API.LIB can be accessed through uppercase specification of the service name preceded by proper setup of parameter information appropriate to the function in question. Unlike the DOS and BIOS interrupt routines, which pass parameter information using the general-purpose registers, the OS/2 API procedures receive parameters via the stack, which must be installed by the user. This is in much the same fashion as the passing of parameters to functions or subroutines in a higher-level language (HLL).

To understand how this works consider the video API call, which returns the cursor position to two stack locations. This routine, VioGetCurPos, has the following calling sequence for the service [3]:

1. Define VioGetCurPos as EXTRN and FAR
2. PUSH a 32-bit address for

   ```
   row       (word)
   column    (word)
   ```

   on the stack, respectively
3. PUSH a device handle

   ```
   VioHandle (word)
   ```

   on the stack
4. CALL VioGetCurPos

In this example the routine VioGetCurPos is treated in mixed upper and lower case for readability. The actual OS/2 library reference is upper case:

```
VIOGETCURPOS
```

To continue to use the more readable mixed-case references, which are in the style of the IBM references, the programmer must consider what is available or can be developed to facilitate the use of these mixed-case calls. Fortunately, IBM provides several include files (with extension .inc) for use with the assembler that set up macros for using the API library. This setup includes loading the stack with the proper parameters needed by the API service routine. OS/2 has two include files, doscalls.inc and subcalls.inc, that properly develop macros to be called for API service. These two files are loaded using a third file, sysmac.inc, which simply installs doscalls and subcalls as macro libraries:

```
...
IF1
          include  sysmac.inc
ENDIF
...
```

The file doscalls.inc contains macros for calling all the Dos...calls. The file subcalls.inc contains macros for calling all kbd..., Mou..., and Vio... calls.

Returning to VioGetCurPos, consider the subcalls macro used to set up and call this service routine:

```
@VioGetCurPos    macro       row, column, handle
                 @define     VIOGETCURPOS
                 @pushs      row
                 @pushs      column
                 @pushw      handle
                 call        far ptr VIOGETCURPOS
                 endm
```

We see immediately that this macro calls three other macros: @define, @pushs, and @pushw. These macros are defined as follows:

```
@define macro   callname
        ifndef  callname
        extrn   callname:far
        endif
        endm
@pushs  macro   parm
        .errb   <parm>
        mov     ax,SEG parm
        push    ax
        lea     ax,parm
        push    ax
        endm
@pushw  macro   parm
        mov     ax,parm
        push    ax
        endm
```

Clearly, @define is used to get VIOGETCURPOS as an externally defined FAR procedure (it appears in API.LIB). The macro @pushs pushes a 32-bit address for the dummy parameter, parm, onto the stack and @pushw pushes parm itself onto the stack. The calling sequence for @VioGetCurPos sets up row and column to receive the cursor position values after the final FAR call to VIOGETCURPOS.

This is how the OS/2 API services are accessed using assembly language and the doscalls.inc, subcalls.inc, and sysmac.inc files. In this chapter we use only a small subset of the API calls. These services are indicated in Table 2.1. Generally, the focus of interest in this chapter is on the printer, keyboard interrupt, and screen buffer, as the API calls of Table 2.1 indicate.

OS/2 reserves the right to redefine memory dynamically during program execution. This is necessary to implement multitasking and memory management of huge segments (greater than 64K segments). Since OS/2 can access 16 Megabytes (MB) of actual memory because of the 24-bit physical address size, it must map the full virtual program memory into this space, or smaller, during program execution. The virtual memory access may contain up to a full gigabyte (2^{30} bytes) of individually addressable byte locations.

Clearly, physical address space is normally difficult to access and naturally remains the province of OS/2. In some cases, however, the programmer has access to this dimension. We will see a situation of actually writing to the OS/2 physical memory when the screen buffer

TABLE 2.1 API SUBSET USED IN CHAPTER 2

API function	Comment
DosOpen	Open specified device or file
DosExit	Terminates active threads and processes
DosWrite	Transfers the specified bytes from a buffer to the specified file
DosClose	Closes the specified device or file
VioScrollUp	Scrolls the screen upward
VioSetMode	Sets the graphics or alphanumeric screen mode
VioScrLock	Locks the physical display buffer context
VioGetPhysBuf	Retrieves a segment selector for the physical display buffer
VioScrUnLock	Unlocks the physical display buffer context
KbdStringIn	Loads a keyboard buffer with a character string

is accessed in a subsequent example. It is possible to gain access to the screen buffer by locking the screen context and then using a segment selector returned by OS/2 for writing directly to the physical buffer containing the screen addresses. This differentiates the IBM physical screen buffer, with its fixed physical locations in memory, from other RAM addresses, which can vary in dynamic but protected fashions under OS/2.

2.2 INTRODUCTORY ASSEMBLER PROGRAMMING

As indicated earlier, we have assumed that the reader has a background in both 80286 assembler and the C language. This book does not teach either, but we do provide a brief review of the syntax associated with the languages. In this section we examine the macro assembler that is compatible with the Protected Mode. Appendix A contains the Macro Assembler/2 instructions and pseudo-ops.

2.2.1 The IBM Macro Assembler/2

There are two reasons why programmers should be interested in assembly languages. First, assembler provides an understanding about both the underlying software architecture for a given microprocessor and the needed chip interfaces for a given microcomputer. Second, situations can arise where other languages are inadequate for achieving optimized performances. The Macro Assembler/2 has basically the same features as other Intel assemblers. The dominant active instruments in the assembler are the instructions with the form

```
[label] instruction-mneumonic   [operand(s)]   [;comment]
```

Here the brackets indicate that the quantities contained within are optional depending on instruction type. The instruction sequence

```
          ...
          mov cx,1000   ;load loop limit
          mov si,0      ;initialize index
DO11:                   ;label for loop
          mov ax,si     ;load ax with index
          sub ax,100    ;subtract 100 from index
          cmp ax,0      ;check to see if zero
          je ELSE1      ;jump if zero to ELSE1
          inc si        ;increment index
          loop DO11     ;loop back to DO11
ELSE1:
          ...
```

is an example of the use of the move(mov), subtraction(sub), jump-if-equal(je), increment(inc), compare(cmp), and loop instructions. Note that the labels DO11 and ELSE1 go with the next line of code. In this fragment the loop instruction decrements cx each time. When ax becomes zero the jump takes place to the target label ELSE1. This very brief illustration of the assembler instruction usage is intended as an example of the IBM Macro Assembler/2, MASM. For a complete discussion of the assembler instructions, consult the Language Reference Manual [4].

In addition to the instructions the assembler has a class of statements that provide information about the program environment. These statements do not result in machine code and are referred to as pseudo-ops. Typical of the pseudo-ops is the SEGMENT directive, which is used to demarcate the various segment definitions within the source code. The SEGMENT pseudo-op has the form

```
sequence SEGMENT      align-type combine-type 'class'
```

where segname is the name of the segment. Align-type indicates how the segment begins in memory [PARA: paragraph boundary [address divisible by 16]; BYTE; WORD; or PAGE: last 8 bits of address are zero], and combine-type indicates how the segment is to be linked [PUBLIC: all public segments with the same name are linked; COMMON: all segments with the same name overlap; AT(exp): segment located at nearest paragraph to "exp"; STACK: stack segment; and MEMORY: higher addresses than other segments]. The designator 'class' refers to a collection of segments with the same class name. Segments end with

```
segname ENDS
```

To define segment type the ASSUME pseudo-op is used to associate a name with a segment register:

```
ASSUME CS:segname, SS:segname[,DS:segname[,ES:segment]]
```

Here CS is required and SS is required when a stack segment is present. Both DS and ES are optional. We could continue to enumerate the Macro Assembler features; however, the best technique for elucidating the language is through illustration. In the following section we consider such an example.

2.2.2 An Example Program: Printer Control

Figure 2.1 contains an assembler program that causes the printer to print in graphics mode under OS/2. The program opens with two pseudo-ops: PAGE and TITLE. PAGE has the form

```
PAGE  operand1,operand2
```

The entry in operand1 indicates the number of horizontal lines per page in the assembler listing (here it is 55). Operand2 is the number of characters per line in the listing. The TITLE pseudo-op specifies the title on the first line of each assembler listing page. Spread throughout the program are semicolons. All text following a semicolon on the same line is treated as a comment. The pseudo-op IF1 (a conditional pseudo-op) indicates that all instructions and pseudo-ops following it and prior to the next ENDIF are to be implemented during pass 1 of the assembler. In Figure 2.1 the file sysmac.inc is to be included at this point.

Sysmac causes doscalls.inc and subcalls.inc to be included which set up macros for all API calls that appear in the subsequent code segments. The pseudo-op .sall causes macro listings to be suppressed. Next follows the GROUP pseudo-op. This pseudo-op collects the data segment under the name dgroup:

```
dgroup   GROUP  data
```

The stack segment follows. Here 256 copies of the string

```
STACK...
```

are used to form the stack segment. This should be more than adequate for the stack size required by most small programs. The pseudo-op, db, stands for define byte and the dup operator duplicates the 8-byte string within parentheses.

The data segment follows and requires some explanation in conjunction with the API calls that are in the code segment. Consider first the variables defined in this data segment that begin dev_.... There are eight of these variables and they are defined in reference to the @DosOpen API macro call. Consider the form of this call in the code segment

```
@DosOpen  dev_name,dev_hand,dev_act,dev_size,
  dev_attr,dev_flag,dev_mode,dev_rsv
```

This API call opens a file with file path name dev_name. The path is the zero-terminated string: 'LPT1',0. The file handle is returned with dev_hand. The action taken is returned in dev_act, where

```
                    PAGE 55,132
TITLE PRT2 - This is the initial printer routine (PRT2.ASM)
;
;       DESCRIPTION: This program simply prints a "74" in
;       graphics mode (320 times) for two lines which are
;       meshed together.
;
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
in_buffer       db      400 dup(0)
in_leng         dw      $ - offset in_buffer
bytesin         dw      320
bytesout        dw      0
in_buffer1      db      1BH,4BH,64D,01H ;320 columns
bytesin1        dw      4
in_buffer2      db      0DH,0AH
bytesin2        dw      2
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3
in_buffer4      db      1BH,32H
;
dev_name        db      'LPT1',0
dev_hand        dw      0
dev_act         dw      0
dev_size        dd      0
dev_attr        dw      0
dev_flag        dw      00000001b       ;Open File
dev_mode        dw      0000000011000001b ;Hdl private,deny none,w/o
dev_rsv         dd      0
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CSEG,DS:DATA,ES:DATA,SS:STACK
PRTSC1  PROC    FAR
        push ds
        pop es
                                        ;
                                        ;Open LPT1 as device
@DosOpen dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag,dev_mode,dev_rsv
        cmp ax,0
        je ELSE1
                                        ;Exit
           @DosExit 1,0
                                        ;
ELSE1:
           mov cx,320                   ;320 columns
           mov si,0                     ;initialize index
LOOP1:
           mov al,74                    ;pins 2,4,5, and 7
           mov in_buffer[si],al         ;load printer write buffer
           inc si                       ;increment buffer index
        loop LOOP1
                                        ;Set lpt1 vertical spacing
```

Figure 2.1 Assembler program prt2.asm, which prints printer graphics under OS/2 Protected Mode.

```
         @DosWrite dev_hand,in_buffer3,bytesin3,bytesout
                                         ;Activate spacing
         @DosWrite dev_hand,in_buffer4,bytesin2,bytesout
                                         ;Initialize printer graphics
         @DosWrite dev_hand,in_buffer1,bytesin1,bytesout
                                         ;Write print buffer
         @DosWrite dev_hand,in_buffer,bytesin,bytesout
                                         ;CR & LF
         @DosWrite dev_hand,in_buffer2,bytesin2,bytesout
                                         ;Reset graphics mode
         @DosWrite dev_hand,in_buffer1,bytesin1,bytesout
                                         ;Write print buffer again
         @DosWrite dev_hand,in_buffer,bytesin,bytesout
                                         ;Close device
         @DosClose dev_hand
                                         ;Exit
         @DosExit 1,0
PRTSC1   endp
CSEG     ends
         end PRTSC1
```

Figure 2.1 *(Concluded)*

```
0001H = file exists
0002H = file created
0003H = file replaced
```

Here the file's size in bytes is returned in dev_size. The file attribute bits are defined as follows:

```
0001H = read only file
0002H = hidden file
0004H = system file
0010H = subdirectory
0020H = file archive
```

with other dev_attr combinations corresponding to reserved values. Dev_flag specifies the action to be taken if the file exists, where

```
00000001B
```

indicates that the file should be opened. The dev_mode parameter has the form

```
        15                                           0
bit:    D  W  F  R  R  R  R  R  I  S  S  S  R  A  A  A
```

where

D = 0 means open in normal way

W = 0 writes may be run through the DOS buffer cache

F = 0 errors reported through system error handler

R	=	0	reserved and must = 0
I	=	1	file handle is private to the current process
SSS	=	100	deny neither Read nor Write access
AAA	=	001	Write only access

Hence

```
dev_mode dw 0000000011000001B
```

corresponds to file handle private, deny none, and write only. The parameter dev_rsv must be zero. We will return to the remaining data segment variables as the code segment API calls that use these variables are considered.

Following the termination of the data segment, DATA, the code segment is developed. This segment, CSEG, opens with an ASSUME pseudo-op that associates each segment register with an appropriate segment name. Here both DS and ES are associated with DATA. Next a FAR procedure PRTSC1 is set up. Notice that the normal DOS program segment prefix (PSP) area is not required. The return is FAR and will be accomplished using

```
@DosExit 1,0
```

which automatically returns execution to the proper OS/2 entry point at the close of PRTSC1. In this API call the first parameter is set to 1 and causes all threads in the process to end. The second parameter is the result code, and this is used by any threads requiring input from the process prior to its termination.

Upon entry to PRTSC1, DS is pushed on the stack and popped into ES. Then @DosOpen is called as discussed above. The return value from this call is in ax and, if 0, means that a normal open occurred. If ax is not zero, @DosExit is called. To understand the remaining instructions and macro calls, it is necessary to understand how the printer works in graphics mode. The @DosOpen macro opens LPT1 (the line printer) as a file. This file can be written using the @DosWrite macro. The line printer used in this example is an EPSON FX-85 [5]. The @DosWrite macro can be used to pass characters for output to the printer as well as passing control codes. We would like to use the printer in graphics mode.

The print head consists of a vertical array of eight pins. In graphics mode these pins have an associated weight as follows:

PIN	WEIGHT
o	128
•	64
o	32
o	16
•	8

```
o    4
•    2
o    1
```

Here three pins have been darkened to indicate that they are active. The total sum of the pin values for these darkened pins is 74; hence when in graphics mode, a 74 output to the printer will cause these pins to print. Similarly, 255 would cause all pins to print. Also, 128 would cause only the top pin to print.

How is the printer placed in graphics mode? Most of the printer control characters are of the form ESC... . To put the printer in single-density graphics mode the sequence

```
ESC "K" (nl) (n2)
```

must be sent. Using ESC = 1BH and "K" = 4BH, it follows that if

```
nl  = d MOD 256
n2  = INT (d/256)
```

where d = total number of columns to be printed, then

```
1BH,     4BH,    64D,    D1H
```

corresponds to setting the printer in the graphics mode with a total of 320 printer columns active, out of a possible 480 for the FX-85.

Returning to the code appearing in Figure 2.1, we see that the buffer, in_buffer(), is loaded with 320 values of 74 (the character value corresponding to the pins discussed earlier). Following the loading of this buffer the macro call

```
@DosWrite dev_hand,in_buffer3,bytesin3,bytesout
```

is made. Here

```
in_buffer3 = 1BH,41H,08H
```

where the first character is ESC. The second character sets the vertical spacing to $^{8}/_{72}$-inch line spacing:

```
ESC A (8)
```

The third parameter in all the @DosWrite calls is the buffer length, and the fourth parameter is the number of bytes written. The macro call

```
@DosWrite dev_hand,in_buffer4,bytesin2,bytesout
```

executes

```
ESC 2
```

which implements the line spacing set above. The macro call

```
@DosWrite  dev_hand,in_buffer1,bytesin1,bytesout
```

sets up the call

```
ESC  K  64  1
```

to specify 320 columns. This command must be followed by 320 characters. The command

```
@DosWrite  dev_hand,in_buffer,bytesin,bytesout
```

outputs 320 columns, corresponding to the 74 graphics combination already illustrated.

Next

```
@DosWrite  dev_hand,in_buffer2,bytesin2,bytesout
```

causes 0DH and 0AH to be output for the carriage return and line feed. This is followed by a reset of the graphics mode and a second print of the 320 values of the graphics mode 74. Figure 2.2a illustrates the output for this program. When the buffer value is changed from 74 to 255, all pins print. This case is illustrated in Figure 2.2b.

(a)

(b)

Figure 2.2 Printer output from prt2.asm (a) with fill character "74" and (b) with fill character "255".

The program appearing in Figure 2.1 illustrates the main features of how to access the API from assembler. Here the printer was accessed using API calls and placed in graphics mode as well as used to output graphics characters. In the next section we look at more complex programs that access the screen buffer. Since we have information about the screen pixels, it will be possible to develop a screen print program that uses the printer in graphics mode to print the screen.

2.3 ACCESSING THE VIDEO SERVICES

To be able to access the screen context requires a knowledge of the physical memory associated with the display. This memory has different partitioning depending on what display mode is being used. Typically, the graphics modes normally accessed by the IBM PS/2 computers (and the IBM AT) are Color Graphics Adapter (CGA) mode, which is a 320-column by 200-row pixel screen, the Enhanced Graphics Adapter (EGA) mode, which is a 640-column by 350-row pixel screen, and the Video Graphics Adapter (VGA) mode, which is a 640-column by 480-row pixel screen.

2.3.1 The Display Buffer

In this chapter we access the CGA screen context. This is mode Hex 5. The memory is allocated into two buffer regions specified as follows:

1. Even Scans (rows 0, 2, 4, ..., 198) starts at address B8000H.
2. Odd Scans (rows 1, 3, 5, ..., 199) starts at address BA000H.
3. Each raster row occupies 80 bytes, where a byte has the following form:

Pixel:	N		N+1		N+2		N+3	
	C1	C0	C1	C0	C1	C0	C1	C0

with color section determined by

C1	C0	
0	0	black
0	1	light cyan
1	0	light magenta
1	1	intensified white

4. Address B8000H contains the pixel information for the first four pixels in the upper left-hand corner.

There is a second CGA mode, which is 640 columns by 200 rows; however, we will not consider this mode. We use the terms *pel* and *pixel* interchangeably herein.

How does the actual location of a pixel attribute get set based on row and column data about the screen? To locate the correct (row, col) byte in screen buffer physical memory, it must be remembered that the even-row value starts at location

```
80 * (row/2)
```

offset from B8000H. Similarly, recognizing that integer division truncates (3/2 becomes 1, ...), the same expression serves to locate an odd-row relative to BA000H. Since there are 80 bytes for 320 columns, we need to locate

```
col/4
```

Hence the offset location of a given byte in terms of (row,col) is given by

```
80 * (row/2) + (col/4)
```

This would correspond to the code

```
...
mov  ax,row
shr  ax,1
mov  dx,0 ;  clear  upper
mul  eighty
mov  bx,col
shr  bx,1
shr  bx,1
add  ax,bx
...
```

To identify an individual pixel within a byte, we note that the least significant bit (LSB) and LSB+1 correspond to the attribute positions for the fourth pixel, (LSB+2, LSB+3) correspond to the attribute positions for the third pixel, and so on. Hence

Bit:	7	6	5	4	3	2	1	0
Pixel:	1	1	2	2	3	3	4	4

We will simply turn the pixel on using a mask:

```
MASK1 = 01H
```

This will produce a light cyan screen color. Dividing col by 4 generates a remainder (0,1,2,3), which is in reverse order to the pixel number (assuming that we start numbering the pixels within a byte 0,1,2,3). Hence

```
3-col mod 4
```

indicates the actual pixel position within the (row,col) byte. Starting with

```
0 0 0 0 0 0 0 1
```

it is clear that a shift

```
2 * (3-col mod 4)
```

will place 1 in bits 6, 4, 2, or 0 as needed to specify the pixel attribute. The following code uses the coprocessor to load xxx with this pixel value based on row, col:

```
...
fild four
fild col
fprem                      ;modulo
fistp xx
fistp dummy
mov  al,3
mov  bl,byte ptr xx
sub  al,bl
mov  ah,0
mul  two
mov  cl,al
mov  al,MASK1              ;MASK1 = 01H
shl  al,cl
mov  xxx,cl
...
```

This, then, is a prescription for using a screen direct memory access (DMA) technique to the video physical buffer, once that buffer has been accessed.

The last code necessary to complete specification of a video buffer location is to specify the precise offset location for address above. Here we assume that the even-row or odd-row location must also be taken into consideration. Consider the code

```
          ...
          mov  ax,row
          and  ax,MASK11               ;MASK11=0001H
          cmp  ax,0
          jle  ELSE1
                 mov  ax,address
                 add  ax,OFFSET1       ;OFFSET1 = 2000H
                 jmp  IF11
ELSE1:
          mov  ax,address
IF11:
          mov  bp,ax
          mov  al,xxx
          or   es:[bp],al
          ...
```

This code checks to see if the row is even or odd. If odd, an offset of 2000H is added to address. The full pixel byte offset is in address and the byte value in xxx. Assuming that the extra segment register contains the video segment selector value, then

```
ov  es:[bp],al
```

changes the bit values from 00 to 01 as needed for the pixel in question.

2.3.2 Locking the Screen Context

Figure 2.3a presents a function flowchart for a program that plots two lines across the screen. Figure 2.3b illustrates this program, which calls the video buffer and plots two parallel lines across the screen. The program also calls a routine scr_ld that loads an intermediate buffer, scr_buffer, with the screen context pixel values. This buffer is then used to output the display context to the printer. We will not focus on the routines that write the display context to the printer until Section 2.3.3. In this section we examine the video API calls.

Consider the first executable instruction in the program the call to cls to clear the screen. The procedure cls, in turn, has a single call (besides the return):

```
@VioScrollUp  tr,lc,br,rc,no_line,blank,viohdl
```

The parameters appearing in this API call are among the first nine parameters appearing in the data segment. Viohdl is a handle to the display. The parameters tr and lc are the top row and left column to be scrolled. The parameters br and vc are the bottom row and right column to be subtended for the scroll operation. A parameter no_line is the number of lines to be scrolled and blank the attribute to be used to replace each character (in this case a blank) pair. This routine effectively blanks the screen.

Next the main FAR procedure sets the screen in CGA graphics mode. To do this the video API call is made referencing the video handle and a CGA structure that contains parameter data:

```
@VioSetMode  CGAm,viohdl
```

The video CGA structure is specified in the data segment by the required parameter values between the statements

```
CGAm  label  FAR
...
vrCGA  dw  200
```

where the last value is the number of rows (the vertical resolution) on the CGA screen. Below this structure in the data segment is a second structure, STDm, which is used later with the call to return to text 80 x 25 mode. This structure spans the lines between

```
STDm  label  FAR
...
vr80  dw  400
```

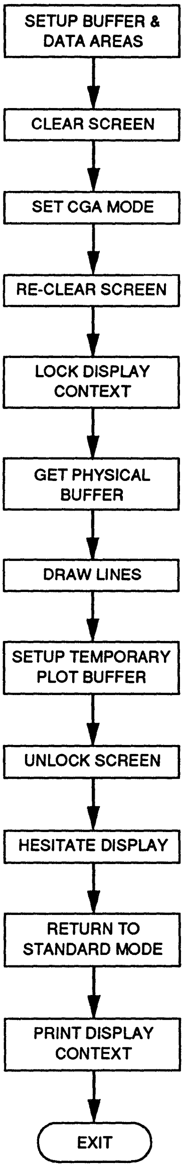


Figure 2.3a Functional flowchart for boxprtl.asm, the program that calls the video buffer and plots two lines.

```
PAGE 55,132
TITLE BOXPRT1 - This program checks print graphics(BOXPRT1.ASM)
;
;       DESCRIPTION: This program plots two lines in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the lines.
;
.8087
PUBLIC  xx,xxx                          ;CodeView symbol map
EXTRN   prtscr:FAR,scr_ld:FAR
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
PUBLIC  in_buffer,bytesin,bytesout,in_buffer1,bytesin1
PUBLIC  in_buffer2,bytesin2,in_buffer3,bytesin3,in_buffer4
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,col1,N
PUBLIC  s,eight,eighty,four,shift1,scr_buffer
PUBLIC  sixforty,N4,ddd,w,b1
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     0                       ;Terminates current thread
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
hrCGA   dw      320                     ;horizontal resolution
vrCGA   dw      200                     ;vertical resolution
;
STDm    label   FAR                     ;Video mode structure-80x25
lmode80 dw      12                      ;Structure length
type80  db      00000001B               ;Mode identifier-Mode 3+
col80   db      4                       ;Color option
txtc80  dw      80                      ;text characters/line
txtr80  dw      25                      ;text lines
hr80    dw      720                     ;horizontal resolution
vr80    dw      400                     ;vertical resolution
;
kbd_buf db      80                      ;Keyboard buffer
lkbd_buf dw     $-kbd_buf               ;Length keyboard buffer
iowait  dw      0                       ;Wait for CR
kbdhdl  equ     0                       ;Keyboard handle
;
waitf   equ     1                       ;Screen waiting status
dstat   db      ?                       ;Returned status
```

Figure 2.3b Program code for boxprtl.asm.

```
;
PVBPtr1 label   FAR                     ;Video buffer structure
bufst1  dd      0B8000H                 ;Start physical address
buflen1 dd      4000H                   ;Buffer length
physel1 dw      0                       ;OS/2 screen buffer selector
;
MASK1   db      01H                     ;PEL byte mask
MASK11  dw      0001H                   ;Odd/even row mask
OFFSET1 dw      2000H                   ;Odd row buffer offset
four    dw      4
xx      dw      ?                       ;PEL modulo parameter
dummy   dw      ?                       ;80287 dummy "pop"
two     db      2
xxx     db      ?                       ;Output value
eighty  dw      80
row     dw      ?                       ;row
col     dw      ?                       ;column
address dw      ?                       ;Address screen dot
;
x       dw      ?                       ;Box col parameter
y       dw      ?                       ;Box row parameter
xb      dw      75                      ;Start column
xe      dw      150                     ;End column
yb      dw      25                      ;Start row
ye      dw      175                     ;End row
;
eight   dw      8
;
;       ------------------------------------------------------
;       Data area below is used for screen print routine.
;
;       ------------------------------------------------------
;
;
in_buffer       db      320 dup(0)      ;print buffer
bytesin dw      320                     ;CGA line
bytesout        dw      0               ;output count
in_buffer1      db      1BH,4BH,64D,01H ;printer setup
bytesin1        dw      4               ;count bytes In_buffer1
in_buffer2      db      0DH,0AH         ;LF/CR
bytesin2        dw      2               ;in_buffer2 byte count
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3               ;in_buffer3 byte count
in_buffer4      db      1BH,32H
;
dev_name        db      'LPT1',0        ;name of printer device
dev_hand        dw      0               ;device handle
dev_act         dw      0               ;
dev_size        dd      0               ;
dev_attr        dw      0               ;
dev_flag        dw      00000001b       ;Open File
dev_mode        dw      0000000011000001b       ;hdl private,deny none,w/o
dev_rsv         dd      0               ;reserved
;
N4              dw      ?
MM              db      40H,10H,04H,01H ;pel mask
w               db      128,64,32,16,8,4,2,1    ;pin weights
col1            db      320 dup(?)      ;column index-printer
b1              db      4 dup(?)
N               dw      ?               ;printer line
shift1          db      6,4,2,0
s               db      4 dup(?)        ;dup copies pel byte
ddd             dw      ?
sixforty        dw      640
scr_buffer      db      16384 dup(0)    ;temporary buffer--screen values
;
;
```

Figure 2.3b (*Continued*)

```
;               -----------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        call cls                        ;Clear screen
        @VioSetMode CGAm,viohdl         ;Set CGA Graphics mode
        call clsCGA                     ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer selector
        push physel1                    ;Save selector
        pop es                          ;Load selector into extra segment
;
        mov ax,0
        mov y,ax
        call lineh                      ;Draw line
        mov ax,100
        mov y,ax
        call lineh                      ;draw second line
                                        ;
        call scr_ld
                                        ;
        @VioScrUnLock   viohdl          ;Unlock screen context
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
;
        @VioSetMode STDm,viohdl         ;80 x 25 alpha mode
;
        call prtscr
;
        @DosExit action,result          ;Terminate process
;
OS21    ENDP
;
cls     PROC    NEAR
;
        @VioScrollUp tr,lc,br,rc,no_line,blank,viohdl
        ret
;
cls     ENDP
;
clsCGA  PROC    NEAR
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer
        push physel1                    ;Screen selector
        pop es                          ;Load extra segment
;
        mov bp,0                        ;Start offset zero
        mov al,0                        ;Zero attribute-clear
DO1:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,1F3FH                    ;Check end 1st buffer
        jle DO1
;
        mov bp,2000H                    ;Offset 2nd buffer-odd
        mov al,0                        ;Zero attribute-clear
DO2:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,3F3FH                    ;Check end 2nd buffer
```

Figure 2.3b (*Continued*)

```
        jle DO2
;
        @VioScrUnLock viohdl            ;Unlock screen context
;
        ret
clsCGA  ENDP
;
wdot    PROC    NEAR
;
;       (col,row) = (x,y)
;
        fild four                       ;Load stack with 4
        fild col                        ;ST = col, ST(1) = 4
        fprem                           ;Modulo
        fistp xx                        ;Store remainder in xx
        fistp dummy                     ;Pop stack
        mov al,3
        mov bl,byte ptr xx
        sub al,bl                       ;(3 - col % 4)
        mov ah,0                        ;Clear upper multiplicand
        mul two
        mov cl,al                       ;Shift value for PEL
        mov al,MASK1                    ;PEL color mask
        shl al,cl                       ;Shift to correct PEL
        mov xxx,al                      ;Store buffer value
                                        ;
        mov ax,row                      ;Begin address calculation
        shr ax,1                        ;Divide row by 2
        mov dx,0                        ;Clear upper multiplicand
        mul eighty
        mov bx,col                      ;Convert column value to bytes
        shr bx,1
        shr bx,1
        add ax,bx                       ;offset in ax
        mov address,ax                  ;Save offset base
        mov ax,row                      ;Check even/odd row
        and ax,MASK11                   ;Look for bit 0 set
        cmp ax,0
        jle ELSE1
           mov ax,address
           add ax,OFFSET1               ;add odd buffer offset
           jmp IF11
ELSE1:
        mov ax,address
IF11:
        mov bp,ax                       ;screen buffer address
        mov al,xxx                      ;Attribute value for dot
;
        or es:[bp],al                   ;Write dot
;
        ret
wdot    ENDP
;
lineh   PROC    NEAR
;
;       y = row position, xb = begin, xe = end
;
        mov ax,y                        ;Establish row for wdot
        mov row,ax
                                        ;
        mov ax,0                        ;x-begin position for line
        mov xb,ax
        mov ax,319                      ;x-end position for line
        mov xe,ax
        mov ax,xb                       ;Establish start column
```

Figure 2.3b (*Continued*)

```
DO10:
        mov col,ax
        push ax                         ;Save column value
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall column
        inc ax                          ;Increment column
        cmp ax,xe                       ;Check end horizontal line
        jle DO10
                                        ;
        ret
lineh   ENDP
;
CSEG    ENDS
        END     OS21
```

Figure 2.3b *(Concluded)*

appearing in the data segment. Finally, a call to clsCGA is made, which reclears the screen in CGA mode. This call is needed because the switch to CGA mode leaves the screen in an unpredictable state. The call to clsCGA is somewhat different than the prior cls call because the screen is now in CGA mode and the screen context must be locked prior to accessing it.

In the procedure clsCGA, the first executable statement is the macro call

```
@VioScrLock waitf,dstat,viohdl
```

This call locks (or requests ownership) of the physical display buffer. The flag waitf is 0 if the screen is not available; otherwise, it is 1. The status, dstat, is 0 if the lock is successful; otherwise, it is 1, and viohdl is the video handle. Once this routine is executed the physical buffer may be accessed. This is accomplished using the statement

```
@VioGetPhysBuf  PVBPtrl,viohdl
```

Here PVBPtrl is a structure with the form (see data segment)

```
...
PVBPtrl   label  FAR
bufstl    dd     0B8000H
buflenl   dd     4000H
physell   dw     0
...
```

The first parameter in this structure, bufstl, is the start address of the physical display buffer specified as a 32-bit physical address. We see that this is merely the beginning of the CGA even-row buffer space, as described above for normal IBM memory allocation (B8000H). The second parameter, buflenl, is the length of the buffer, which is 4000H or 16384 bytes long. Finally, physell is the physical selector which is returned by the call. Upon completion of the call the physical selector value is immediately loaded in the extra segment register es. Hence, es then points to the beginning of the physical buffer. This step is very important because it confirms the translation of the segment registers and the segment arithmetic for calculating a physical or virtual address. Following the loading of the selector address, the two buffer regions are cleared: even rows (offset 0-1F3FH) and odd rows (offset 2000H-3F3FH). Then the screen context is unlocked with the call

```
@VioScrUnlock viohdl
```

The actual screen write is accomplished using two calls to the procedure lineh, one call at y value 0 and one call at y value 100 (halfway down the screen). The form of lineh use in this program merely draws a straight horizontal line from column 0 to column 319 of the screen. The actual drawing of the dot is accomplished by a procedure wdot, which implements the techniques of section 2.3.1 discussed earlier.

Following the plotting of the two horizontal lines on the display, the screen context is loaded in the buffer, scr_buffer, based on a call to scr_ld. Eventually, the screen is printed using prtscr, which employs this buffer as a template of the screen context. The screen is next unlocked and the keyboard pause or hesitation is instituted with the call

```
[@kbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl
```

Here kbd_buf is a buffer for a character string that is 80 bytes wide. The variable 1kbd_buf is the length of this buffer. A value of 0 for iowait indicates that the system should wait or hesitate if a character is not available. The parameter kbdhdl is the handle to the keyboard device context. This call, of course, pauses the action and allows the user to view the screen.

The second call to @VioSetMode returns the video context to 80 x 25 text mode. Calling prtscr prints the intermediate screen buffer on the printer as described above. Finally, @DosExit causes the program to exit back to OS/2. Figure 2.4 is the actual print of the screen output.

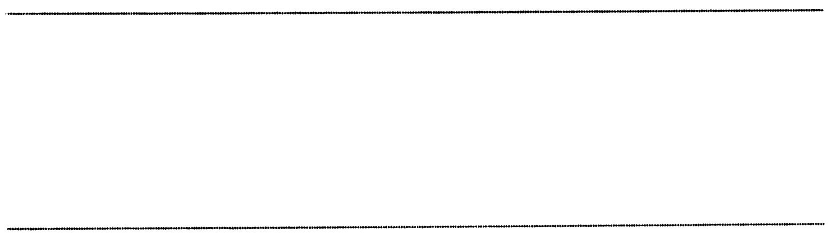


Figure 2.4 Output print screen from boxprtl.asm (Figure 2.3b).

2.3.3 Printing the Graphics Screen under OS/2

In Figure 2.3 a portion of the data segment was devoted to parameters and variables used by scr_ld and prtscr for the printer dump of the screen context. These variables appeared earlier in the program of Figure 2.1, where a simple graphics print output was generated. In this section we address the topic of how to achieve a printout of the graphics screen context. This is similar to employing GRAPHICS.COM under DOS except that our screen print program does not run in the background but is directly callable by the program executing. IBM and Microsoft did not provide the equivalent of GRAPHICS.COM with their system software during the early releases of OS/2. Hence this program is both useful for obtaining a hard copy of the graphics screen and as information for illustrating the combined techniques of display access and graphics printer output.

We have seen how to access the screen physical buffer using API calls. Also, we saw a routine, scr_ld, used ostensibly to load a buffer scr_buffer. Figure 2.5 illustrates this routine and we see it is a very simple procedure with no API calls. Only the byte array, scr_buffer, is external. The routine also interleaves the even and

```
PAGE 55,132
TITLE SCRLD -- This routine loads the screen print buffer (scrld.asm)
;
;       DESCRIPTION:  This routine accompanies prtscr to load
;       and print the screen in 320 x 200 mode.
;       The prtscr buffers are assumed loaded.  This is an OS/2
;       routine.
;
EXTRN   scr_buffer:BYTE
;
        .sall
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        PUBLIC  scr_ld
scr_ld  PROC    FAR
        ASSUME  CS:CSEG
                                        ;
        mov cx,100                      ;no. of raster pairs
        mov di,0                        ;index to screen buffer
        mov si,0                        ;index to dummy array
DO55:
        push cx
        mov cx,80                       ;raster row length
DO56:
        mov al,es:[di]                  ;load even row physical buffer
        mov ah,es:[di+2000H]            ;odd row physical buffer
        lea bx,scr_buffer[0]            ;dummy buffer
        mov ds:[bx+si],al               ;load even rows
        mov ds:[bx+si+80],ah            ;odd rows
        inc si
        inc di
        loop DO56
                                        ;
        add si,80                       ;skip to next double set
        pop cx
        loop DO55
                                        ;
        ret
scr_ld  ENDP
CSEG    ENDS
        END
```

Figure 2.5 Routine to set up temporary screen print buffer.

odd rows from the physical buffer regions into a single buffer area which represents the full screen context in contiguous fashion.

Figure 2.6a illustrates the function flowchart for the print screen routine. Figure 2.6b contains the actual print screen routine. All the printer parameters referenced in the earlier data segments appear as external variables and are defined as such at the beginning of the program. Following the usual loading of sysmac.inc, the program starts immediately with the code segment, CSEG. The routine prtscr is declared PUBLIC. In general, our approach will be to treat prtscr and scr_ld as externally callable modules whenever a printer screen dump is desired. Hence these two modules will become workhorse functions for illustrating graphics displays and the reader can expect to encounter them throughout the book. Shortly we will install them in a general-purpose library GRAPHLIB.LIB where they will be universally accessible. The only difficult part about programming in this fashion is the large data segment areas that are needed to set up the calls to these printer procedures (and the screen parameter areas).

Returning to Figure 2.6, we see immediately the usual call, @DosOpen, to open the printer device context. This was discussed in reference to Figure 2.1. Since this program returns to a calling procedure, the ret instruction is implemented rather than @DosExit. The sequence of API calls to @DosWrite is generally in agreement with the earlier programming of Figure 2.1 except that the double output is omitted. A loop is set up to increment 25 times, once for each eight-line graphics print. This yields a total of 200 rows displaced vertically. These rows correspond to the actual screen buffer rows for the raster scan. Since each row of the screen buffer consists of 80 bytes of pixel data, eight rows at a time correspond to blocks of 640 bytes of data.

The call to ldarray sets up the output for the printer eight rows at a time. Basically, a small 32-element buffer, col1[], is loaded with the four pixels' worth of data contained in each byte of the physical display buffer. This is done for the same byte from eight consecutive rows of the screen buffer. Hence ldarray sets up a group of pixel data representing a block of the screen context. To do this an array of four elements, s[0] to s[3], is loaded with a byte of the screen buffer data from scr_buffer. Each pixel is then masked off from its position in this byte, shifted, and weighted to generate the correct graphics printer character. The weights, for example, contained in the array, w[], must be specified in the calling program's reserved printer data area in the usual fashion. It is this technique that is used to load the array col1[].

Returning to prtscr itself, we see that after each eight-line block by 320 columns is loaded and in_buffer[] properly loaded the graphics print is implemented. This is in the fashion of Figure 2.1 and is followed by a carriage return and line feed. Once the complete screen dump to the printer has been accomplished, prtscr closes the printer device handle with

```
@DosClose dev_hand
```

and returns to the calling routine.

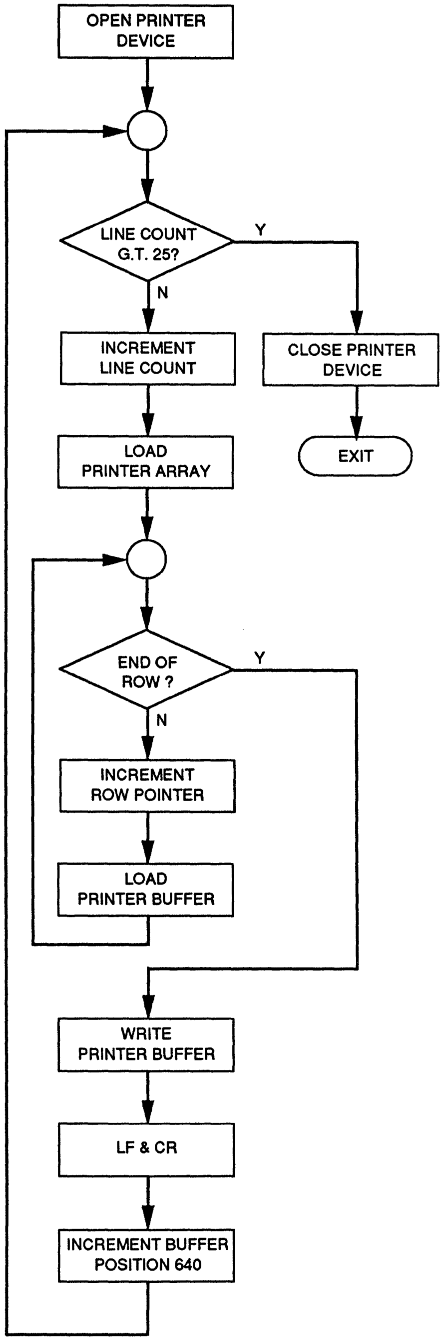


Figure 2.6a Functional flowchart for prtscr, the screen dump routine.

```
PAGE    55,132
TITLE   prtscr - print screen (prtscr.asm)
;
;       DESCRIPTION: This routine prints the screen in
;       320 x 200 CGA mode.  This routine needs the
;       following data items in the calling routine
;       data segment:
;
;               ------------------------------------------
;               in_buffer       db      320 dup(0)
;               bytesin         dw      320
;               bytesout        dw      0
;               in_buffer1      db      1BH,4BH,64D,01H
;               bytesin1        dw      4
;               in_buffer2      db      0DH,0AH
;               bytesin2        dw      2
;               in_buffer3      db      1BH,41H,08H
;               bytesin3        dw      3
;               in_buffer4      db      1BH,32H
;                                       ;
;               dev_name        db      'LPT1',0
;               dev_hand        dw      0
;               dev_act         dw      0
;               dev_size        dd      0
;               dev_attr        dw      0
;               dev_flag        dw      00000001b
;               dev_mode        dw      0000000011000001b
;               dev_rsv dd      0
;                                       ;
;               MM              db      40H,10H,04H,01H
;               w               db      128,64,32,16,8,4,2,1
;               col1            db      320 dup(?)
;               N               dw      ?
;               N4              dw      ?
;               s               db      4 dup(?)
;               shift1          db      6,4,2,0
;               eight           dw      8
;               eighty          dw      80
;               b1              db      4 dup(?)
;               four            dw      4
;               ddd             dw      ?
;               sci_buffer      db      16192 dup(0)
;               sixforty        dw      640
;               ---------------------------------
;
EXTRN   MM:BYTE,w:BYTE,col1:BYTE
EXTRN   in_buffer:BYTE,in_buffer1:BYTE,in_buffer2:BYTE
EXTRN   in_buffer3:BYTE,in_buffer4:BYTE
EXTRN   bytesin:WORD,bytesin1:WORD,bytesin2:WORD,bytesin3:WORD
EXTRN   bytesout:WORD,dev_name:BYTE,dev_hand:WORD
EXTRN   dev_act:WORD,dev_size:DWORD,dev_attr:WORD
EXTRN   dev_flag:WORD,dev_mode:WORD,dev_rsv:DWORD
EXTRN   N:WORD,N4:WORD
EXTRN   eighty:WORD,eight:WORD,four:WORD,s:BYTE,shift1:BYTE
EXTRN   scr_buffer:BYTE,ddd:WORD,b1:BYTE,sixforty:WORD
;
IF1
        include sysmac.inc
ENDIF
;
        .sall
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        PUBLIC  prtscr
```

Figure 2.6b Routine to print the screen once the physical display buffer is captured.

```
prtscr  PROC     FAR
        ASSUME   CS:CSEG
                                        ;open device
@DosOpen dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag,dev_mode,dev_rsv
        cmp ax,0
        je ELSE1
                                        ;Exit
           ret
ELSE1:
                                        ;initialize device
        @DosWrite dev_hand,in_buffer3,bytesin3,bytesout
        @DosWrite dev_hand,in_buffer4,bytesin2,bytesout
                                        ;
        mov dx,25                       ;number print lines(+1)
        mov si,0                        ;index to 8 row block
LOOP1:
        push dx                         ;preserve dx
        push si                         ;preserve block count
        mov ax,si
        mul sixforty                    ;640 block size
        mov N,ax                        ;Save in N
                                        ;
        call ldarray
                                        ;
        mov di,0                        ;initialize 320 column counter
        mov cx,80                       ;count of column bytes
LOOP2:
        mov al,col1[di]                 ;column 1 from byte
        mov in_buffer[di],al            ;load print buffer
        mov al,col1[di+1]               ;column 2 from byte
        mov in_buffer[di+1],al          ;load print buffer
        mov al,col1[di+2]               ;column 3 from byte
        mov in_buffer[di+2],al          ;load print buffer
        mov al,col1[di+3]               ;column 4 from byte
        mov in_buffer[di+3],al          ;load print buffer
        add di,four                     ;increment column index
        loop LOOP2
;
                                        ;write print row
        @DosWrite dev_hand,in_buffer1,bytesin1,bytesout
        @DosWrite dev_hand,in_buffer,bytesin,bytesout
        @DosWrite dev_hand,in_buffer2,bytesin2,bytesout
                                        ;
        pop si                          ;recall block count
        pop dx                          ;recall print line count
        dec dx                          ;decrement count
        inc si                          ;increase block count
        cmp dx,0                        ;check 25 lines printed
        jle DII1
           jmp LOOP1
DII1:
        @DosClose dev_hand              ;close print device
        ret
prtscr  endp
;
ldarray PROC     NEAR
;
;       N is the printer row # - 640 byte intervals [0,24]
;       MM[0] = 40H,...,MM[3] = 01H (pel mask)
;       w[0] = 128,w[1] = 64,...,w[7] = 1
;
        mov si,0                        ;column count initialization
        mov cx,320                      ;320 columns
DO110:
        mov al,0                        ;clear print buffer
        mov col1[si],al
        inc si                          ;increment column count
```

Figure 2.6b (*Continued*)

```
        loop DO110
                                        ;
        mov si,0                        ;index into 80 bytes/row
        mov N4,si                       ;N4 = row byte block count
        mov dx,80                       ;counter - row bytes
DO111:
                                        ;
        mov di,0                        ;raster row counter (1 of 8)
        mov ddd,di                      ;80 block counter
        mov cx,8                        ;raster row counter
DO112:
        push cx                         ;preserve row count
        mov bp,ddd                      ;bp = # 80 byte blocks
        add bp,N                        ;add printer line count
        push bx                         ;preserve bx
        lea bx,scr_buffer[0]            ;load address screen buffer
        add bp,bx                       ;add to index
        mov al,ds:[bp+si]               ;4 pel bytes
        pop bx
        mov s[0],al                     ;1st copy
        mov s[1],al                     ;2nd copy
        mov s[2],al                     ;3rd copy
        mov s[3],al                     ;4th copy
                                        ;
        and al,MM[0]                    ;1st pel mask
        mov cl,shift1[0]                ;load 1st pel shift
        shr al,cl                       ;shift right
        mov ah,0                        ;clear upper
        mul w[di]                       ;multiply by weight (row)
        mov b1[0],al                    ;save 1st printer column
                                        ;
        mov al,s[1]                     ;load 2nd pel
        and al,MM[1]                    ;mask 2nd pel
        mov cl,shift1[1]                ;load 2nd pel shift
        shr al,cl                       ;shift right
        mov ah,0                        ;clear upper
        mul w[di]                       ;multiply by weight (row)
        mov b1[1],al                    ;save 2nd printer column
                                        ;
        mov al,s[2]                     ;load 3rd pel
        and al,MM[2]                    ;mask 3rd pel
        mov cl,shift1[2]                ;load 3rd pel shift
        shr al,cl                       ;shift right
        mov ah,0                        ;clear upper
        mul w[di]                       ;multiply by weight (row)
        mov b1[2],al                    ;save 3rd printer column
                                        ;
        mov al,s[3]                     ;load 4th pel
        and al,MM[3]                    ;mask 4th pel
        mov cl,shift1[3]                ;load 4th pel shift
        shr al,cl                       ;shift right
        mov ah,0                        ;clear upper
        mul w[di]                       ;multiply by weight (row)
        mov b1[3],al                    ;save 4th printer column
                                        ;
        push bx                         ;preserve bx
        mov bx,N4                       ;counter into print buffer
        mov al,b1[0]                    ;load column N4
        add col1[bx],al
        mov al,b1[1]                    ;load column N4+1
        add col1[bx+1],al
        mov al,b1[2]                    ;load column N4+2
        add col1[bx+2],al
        mov al,b1[3]                    ;load column N4+3
        add col1[bx+3],al
        pop bx
```

Figure 2.6b *(Continued)*

```
                                              ;
        pop cx                                ;recall print block row index
        inc di                                ;increase print block row counter
        add ddd,80                            ;increase byte count
        dec cx                                ;decrease row bound
        cmp cx,0
        jle DO133
           jmp DO112
DO133:
        add N4,4                              ;add 4 columns to index
        dec dx                                ;decrement row bytes
        inc si                                ;increase screen buffer index
        cmp dx,0
        jle DII2
           jmp DO111
DII2:
        ret
ldarray ENDP
;
CSEG    ENDS
        END
```

Figure 2.6b *(Concluded)*

Figure 2.7a presents a Structure Chart for the modularized boxprtl.asm program. Figure 2.7b presents a modularized version of the earlier boxprtl.asm program. Here all the graphics and print routines have been assembled as separate modules. Only the large data segment areas are present with the small FAR procedure that actually plots and prints the two lines.

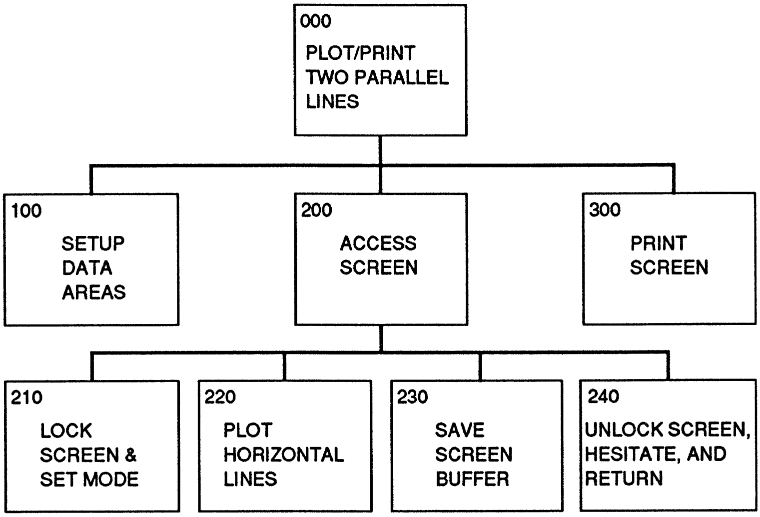


Figure 2.7a Structure Chart for modularized boxprtl.asm program.

Figure 2.8 illustrates a module that is used to build GRAPHLIB.LIB, a graphics and print library. This module contains the routines needed by twoln.asm to develop the two-line output in modular fashion. There is only one difference: In the twoln.asm program the length of the lines must be specified in the routine lineh.

This was implemented to be 320 columns with the earlier version of lineh. That version was used with twolin.asm, and we have illustrated the more general lineh in Figure 2.8 because it is characteristic of what appears in GRAPHLIB.LIB. Here, the beginning and ending column values must be specified in xb and xe, respectively.

```
        PAGE 55,132
TITLE TWOLN - This program plots/prints 2 lines (twoln.asm)
;
;       DESCRIPTION: This program plots two lines in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the lines.
;
.8087
EXTRN   prtscr:FAR,scr_ld:FAR,cls:FAR,clsCGA:FAR,lineh:FAR
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
;       ------------------------------------------------
;               Graphics (printer) variables public
;       ------------------------------------------------
;
PUBLIC  in_buffer,bytesin,bytesout,in_buffer1,bytesin1
PUBLIC  in_buffer2,bytesin2,in_buffer3,bytesin3,in_buffer4
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,col1,N
PUBLIC  s,eight,eighty,four,shift1
PUBLIC  sixforty,N4,ddd,w,b1,scr_buffer
;       ------------------------------------------------
;
;       ------------------------------------------------
;               Graphics (screen) variables public
;       ------------------------------------------------
PUBLIC  xx,xxx,tr,lc,br,rc,no_line,blank,viohdl,PVBPtr1,physel1,waitf
PUBLIC  dstat,four,two,col,dummy,MASK1,MASK11,row,eighty,address
PUBLIC  OFFSET1,y,xb,xe
;       ------------------------------------------------
;
;
;       ------------------------------------------------
;               Screen display variables
;       ------------------------------------------------
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     0                       ;Terminates current thread
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
```

Figure 2.7b Modularized program twoln.asm.

```
hrCGA   dw      320                         ;horizontal resolution
vrCGA   dw      200                         ;vertical resolution
;
STDm    label   FAR                         ;Video mode structure-80x25
lmode80 dw      12                          ;Structure length
type80  db      00000001B                   ;Mode identifier-Mode 3+
col80   db      4                           ;Color option
txtc80  dw      80                          ;text characters/line
txtr80  dw      25                          ;text lines
hr80    dw      720                         ;horizontal resolution
vr80    dw      400                         ;vertical resolution
;
kbd_buf db      80                          ;Keyboard buffer
lkbd_buf dw     $-kbd_buf                   ;Length keyboard buffer
iowait  dw      0                           ;Wait for CR
kbdhdl  equ     0                           ;Keyboard handle
;
waitf   equ     1                           ;Screen waiting status
dstat   db      ?                           ;Returned status
;
PVBPtr1 label   FAR                         ;Video buffer structure
bufst1  dd      0B8000H                     ;Start physical address
buflen1 dd      4000H                       ;Buffer length
physel1 dw      0                           ;OS/2 screen buffer selector
;
MASK1   db      01H                         ;PEL byte mask
MASK11  dw      0001H                       ;Odd/even row mask
OFFSET1 dw      2000H                       ;Odd row buffer offset
four    dw      4
xx      dw      ?                           ;PEL modulo parameter
dummy   dw      ?                           ;80287 dummy "pop"
two     db      2
xxx     db      ?                           ;Output value
eighty  dw      80
row     dw      ?                           ;row
col     dw      ?                           ;column
address dw      ?                           ;Address screen dot
;
x       dw      ?                           ;Box col parameter
y       dw      ?                           ;Box row parameter
xb      dw      75                          ;Start column
xe      dw      150                         ;End column
yb      dw      25                          ;Start row
ye      dw      175                         ;End row
;
eight   dw      8
;
;       ------------------------------------------------------
;       Data area below is used for screen print routine.
;       ------------------------------------------------------
;
;
in_buffer       db      320 dup(0)          ;print buffer
bytesin dw      320                         ;CGA line
bytesout        dw      0                   ;output count
in_buffer1      db      1BH,4BH,64D,01H     ;printer setup
bytesin1        dw      4                   ;count bytes In_buffer1
in_buffer2      db      0DH,0AH             ;LF/CR
bytesin2        dw      2                   ;in_buffer2 byte count
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3                   ;in_buffer3 byte count
in_buffer4      db      1BH,32H
;
dev_name        db      'LPT1',0            ;name of printer device
dev_hand        dw      0                   ;device handle
dev_act         dw      0                   ;
```

Figure 2.7b (*Continued*)

```
dev_size        dd      0                   ;
dev_attr        dw      0                   ;
dev_flag        dw      00000001b           ;Open File
dev_mode        dw      0000000011000001b         ;hdl private,deny none,w/o
dev_rsv         dd      0                   ;reserved
;
N4              dw      ?
MM              db      40H,10H,04H,01H ;pel mask
w               db      128,64,32,16,8,4,2,1     ;pin weights
col1            db      320 dup(?)          ;column index-printer
b1              db      4 dup(?)
N               dw      ?                   ;printer line
shiftl          db      6,4,2,0
s               db      4 dup(?)            ;dup copies pel byte
ddd             dw      ?
sixforty        dw      640
scr_buffer      db      16384 dup(0)        ;temporary buffer--screen values
;
;
;               -----------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        call cls                            ;Clear screen
;
        @VioSetMode CGAm,viohdl             ;Set CGA Graphics mode
;
        call clsCGA                         ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl      ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl       ;Get physical buffer selector
        push physel1                        ;Save selector
        pop es                              ;Load selector into extra segment
;
        mov ax,0
        mov y,ax
        call lineh                          ;Draw line
        mov ax,100
        mov y,ax
        call lineh                          ;draw second line
;
        call scr_ld                         ;loads the temporory buffer
;
        @VioScrUnLock   viohdl              ;Unlock screen context
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl      ;hesitate
;
        @VioSetMode STDm,viohdl             ;80 x 25 alpha mode
;
        call prtscr                         ;prints temporary buffer
;
        @DosExit action,result              ;Terminate process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figured 2.7b (*Concluded*)

```
PAGE 55,132
TITLE GRAPH1 - This program is part of graphlib.lib(graph1.ASM)
;
;       DESCRIPTION: cls,clsCGA,wdor,and lineh routines
;
.8087
IF1
        include sysmac.inc
ENDIF
;
EXTRN   tr:WORD,lc:WORD,br:WORD,rc:WORD,no_line:WORD,blank:WORD
EXTRN   viohdl:WORD,PVBPtr1:FAR,physel1:WORD,waitf:WORD
EXTRN   dstat:BYTE,four:WORD,col:WORD,xx:WORD,dummy:WORD
EXTRN   MASK1:BYTE,xxx:BYTE,row:WORD,eighty:WORD,address:WORD
EXTRN   MASK11:WORD,OFFSET1:WORD,y:WORD,xb:WORD,xe:WORD
EXTRN   two:WORD
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg
        PUBLIC cls,clsCGA,wdot,lineh
;
cls     PROC    FAR
;
        @VioScrollUp tr,lc,br,rc,no_line,blank,viohdl
        ret
;
cls     ENDP
;
clsCGA  PROC    FAR
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer
        push physel1                    ;Screen selector
        pop es                          ;Load extra segment
;
        mov bp,0                        ;Start offset zero
        mov al,0                        ;Zero attribute-clear
DO1:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,1F3FH                    ;Check end 1st buffer
        jle DO1
;
        mov bp,2000H                    ;Offset 2nd buffer-odd
        mov al,0                        ;Zero attribute-clear
DO2:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,3F3FH                    ;Check end 2nd buffer
        jle DO2
;
        @VioScrUnLock viohdl            ;Unlock screen context
;
        ret
clsCGA  ENDP
;
wdot    PROC    FAR
;
;       (col,row) = (x,y)
;
        fild four                       ;Load stack with 4
        fild col                        ;ST = col, ST(1) = 4
```

Figure 2.8 Listing of partial content of GRAPHLIB.LIB.

```
        fprem                           ;Modulo
        fistp xx                        ;Store remainder in xx
        fistp dummy                     ;Pop stack
        mov al,3
        mov bl,byte ptr xx
        sub al,bl                       ;(3 - col % 4)
        mov ah,0                        ;Clear upper multiplicand
        mul two
        mov cl,al                       ;Shift value for PEL
        mov al,MASK1                    ;PEL color mask
        shl al,cl                       ;Shift to correct PEL
        mov xxx,al                      ;Store buffer value
                                        ;
        mov ax,row                      ;Begin address calculation
        shr ax,1                        ;Divide row by 2
        mov dx,0                        ;Clear upper multiplicand
        mul eighty
        mov bx,col                      ;Convert column value to bytes
        shr bx,1
        shr bx,1
        add ax,bx                       ;offset in ax
        mov address,ax                  ;Save offset base
        mov ax,row                      ;Check even/odd row
        and ax,MASK11                   ;Look for bit 0 set
        cmp ax,0
        jle ELSE1
           mov ax,address
           add ax,OFFSET1               ;add odd buffer offset
           jmp IF11
ELSE1:
        mov ax,address
IF11:
        mov bp,ax                       ;screen buffer address
        mov al,xxx                      ;Attribute value for dot
;
        or es:[bp],al                   ;Write dot
;
        ret
wdot    ENDP
;
lineh   PROC    FAR
;
;       y = row position, xb = begin, xe = end
;
        mov ax,y                        ;Establish row for wdot
        mov row,ax
                                        ;
        mov ax,xb                       ;Establish start column
DO10:
        mov col,ax
        push ax                         ;Save column value
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall column
        inc ax                          ;Increment column
        cmp ax,xe                       ;Check end horizontal line
        jle DO10
                                        ;
        ret
lineh   ENDP
;
CSEG    ENDS
        END
```

Figure 2.8 (*Concluded*)

2.3.4 Connecting Line Graphics with OS/2

Consider two disjoint points on the screen at coordinates (x_0,y_0) and (x_1,y_1), respectively. (Assume that x_i corresponds to a column value (1,320) and y corresponds to a row value [1,200].) If we are plotting a dot at these points, it is desirable perhaps to link two points with a line to show connectivity. Since there may exist pixels on the screen between these two points, a program could fill in these pixels and the screen would appear to have a line connecting the two points. To do this, we use the equation for a straight line:

$$y_2 = y_0 + m(x_2 - x_0)$$

Here the slope is

$$m = \frac{y_2 - y_0}{x_2 - x_0}$$

We have used y_2 and x_2 as dummy variables to represent the intermediate points in question.

Unfortunately, the density of dots available on the IBM Color Graphics Adapter screen is at most 320 x 200 or 640 x 200. Although this seems like a lot of points, the screen is large and frequently the connecting lines appear jagged. This is because the slope is effectively quantized. To understand this, consider two points with slope 0.1 between them. Recognizing that y_2, y_0, and x_0 are all integers in Equation (2.1), it follows that

$$y_2 = y_0 + (0.1)\,(x_2 - x_0)$$

Clearly, for y_2 to increase by one pixel on the screen, $x_2 - x_0$ must change by 11 pixels in the horizontal direction. Thus the lines appear broken.

Equations (2.1) and (2.2) are the key to developing techniques for plotting connecting line graphics in the IBM microcomputer context (or any other raster scanning device, for that matter). Figure 2.9a contains the flowchart for the connecting line program. Figure 2.9b illustrates the procedure CONNL2, which plots connecting lies between the points (X0,Y0) and (X1,Y1) using as dummy variables (X2,Y2). The remaining variables (NCOUNT, SIGN, and M) are self-explanatory. The only complex feature of this routine, as it implements Equations (2.1) and (2.2), is the scaling mechanism. To prevent undue round-off the numerator of the slope is scaled up by a factor of 100. This is subsequently removed. The sign of the slope (SIGN) is calculated and used to demarcate the procedure based on positive versus negative values. The routine wdot is used to plot the connecting line.

Figure 2.10 presents a program, slopeln.asm, that plots a connecting line between the points

$$(x_0, y_0) = (25,25)$$

and

$$(x_2, y_2) = (275,175)$$

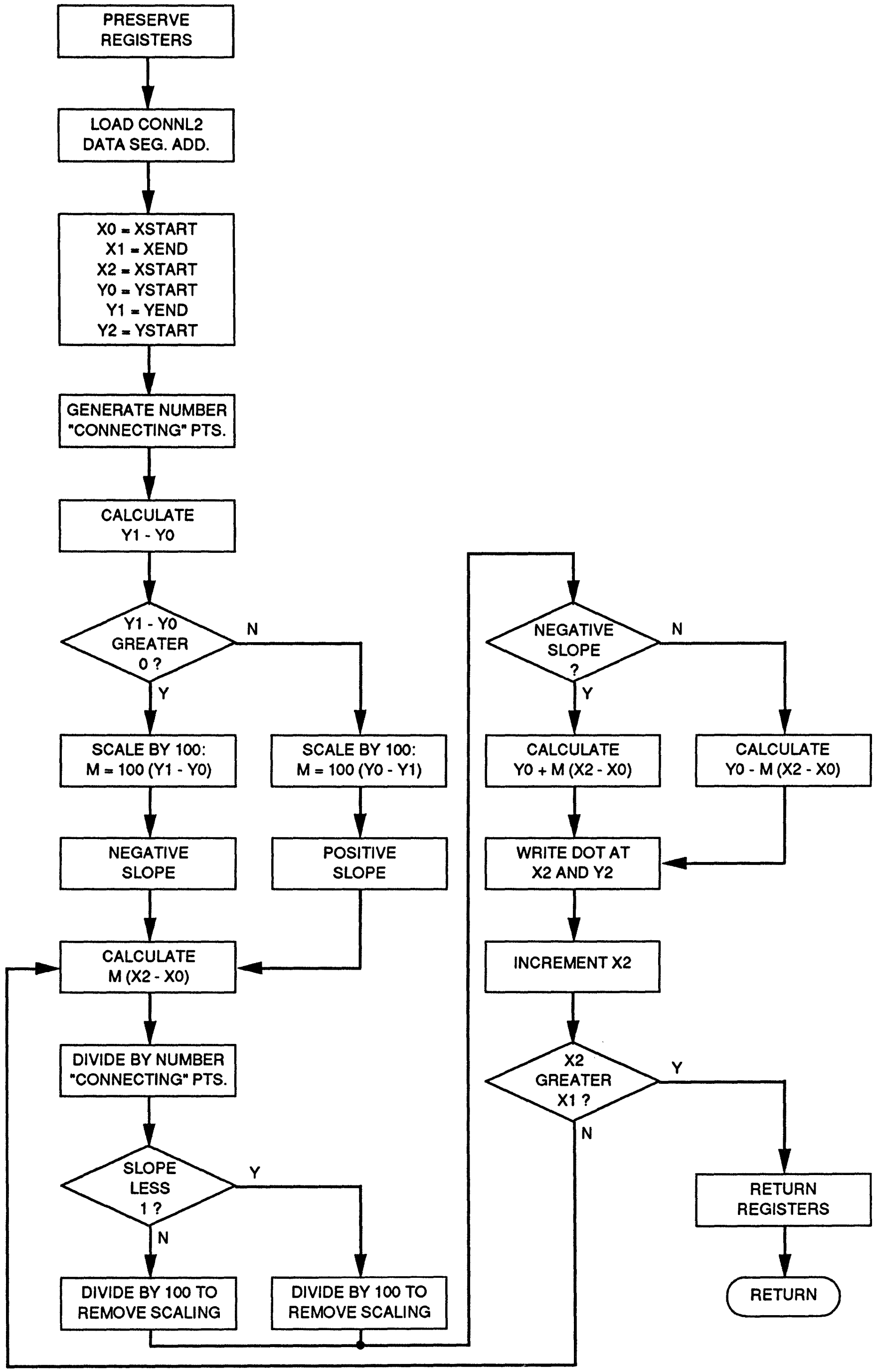


Figure 2.9a Functional flowchart for connecting line routine, conn12.

```
PAGE 40,132
TITLE CONNL2- CONNECT LINE AND PLOT (CONNL2.ASM)
;
;               DESCRIPTION:This routine reads DX =
;               (YSTART,YEND), BX = XSTART, and CX = XEND.
;               It generates a connecting line between the
;               points (XSTART,YSTART) and (XEND,YEND) and
;               plots the points.  The routine as part of the
;               S/2 graphlib.lib
;
EXTRN   Y0:WORD,Y1:WORD,Y2:WORD,X0:WORD,X1:WORD,X2:WORD
EXTRN   NCOUNT:WORD,SIGN:WORD,M:WORD,col:WORD,row:WORD
;
EXTRN   wdot:FAR
;
;       -----------------------------------------------------
;               Connl2 Variables
;       -----------------------------------------------------
;       Y0      DW      0               ;Y start
;       Y2      DW      0               ;Y-value (dynamic)
;       Y1      DW      0               ;Y end
;       X0      DW      0               ;X start
;       X2      DW      0               ;X-value (dynamic)
;       X1      DW      0               ;X end
;       NCOUNT  DW      0               ;Number points in line
;       SIGN    DW      0               ;Sign slope
;       M       DW      0               ;Scaled partial slope
;       ------------------------------------------------------
;
CLINE   SEGMENT PARA PUBLIC 'CODE'
        PUBLIC CONNL2
CONNL2  PROC    FAR
        ASSUME CS:CLINE
                                ;
        PUSH DS
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH DX
        PUSH DI
        PUSH SI
                                ;Load screen coordinates
        MOV AL,DH               ;DH contains YSTART
        MOV AH,0                ;Clear top half AX
        MOV Y0,AX               ;Start Y-point
        MOV Y2,AX               ;Also save YSTART in y
        MOV AL,DL               ;DL contains YEND
        MOV AH,0                ;Clear top half AX
        MOV Y1,AX               ;End Y-point
        MOV X0,BX               ;Start X-point
        MOV X2,BX               ;Save XSTART in x also
        MOV X1,CX               ;End X-point
                                ;
                                ;Generate count index
                                ;
        MOV AX,X1               ;Larger X-value in increment
        SUB AX,X0               ;Calculate X-increment
        MOV NCOUNT,AX           ;Number of X-points to connect
                                ;Generate slope
        MOV DX,0                ;Clear upper numerator register
        MOV AX,Y1
        SUB AX,Y0               ;Begin calculation Y1 - Y0 for slope
```

Figure 2.9b Program code for connl2, the connecting line routine.

```
        JB ELSE1
                MOV CX,100      ;Scale slope by 100
                MOV DX,0        ;Clear upper multiplicand register
                MUL CX
                MOV M,AX        ;Slope in M
                MOV AX,1        ;Sign negative for slope
                MOV SIGN,AX     ;Sign increment Y-axis points
                JMP IIF1
ELSE1:
                MOV AX,Y0       ;Positive slope
                SUB AX,Y1       ;Calculate (Y1 - Y0)
                MOV CX,100      ;Scale by 100
                MOV DX,0        ;Clear upper register
                MUL CX
                MOV M,AX        ;Slope in M
                MOV AX,0        ;Positive slope
                MOV SIGN,AX     ;Sign deecision Y-axis points
IIF1:
DO1:
                MOV AX,X2
                SUB AX,X0       ;(X - X0)
                MOV DX,0        ;Clear upper multiplicand register
                MUL M           ;Multiply by slope numerator
                DIV NCOUNT      ;Begin completion slope calculation
                MOV CX,100      ;Value corresponding to slope 1
                CMP AX,CX       ;Check for slope less 1
                JB ELSE2                ;Jump slope less/= 1
                        MOV DX,0        ;Clear upper register
                        DIV CX          ;Remove scaling
                        JMP IIF2
ELSE2:
                        MOV AX,0        ;0 slope
IIF2:
                MOV BX,SIGN
                CMP BX,1        ;Jump positive slope
                JB ELSE3
                        MOV BX,Y0       ;Load Y-start value
                        ADD AX,BX       ;Add Mx(X-X0)
                        JMP IIF3
ELSE3:
                        MOV BX,Y0       ;Positive slope
                        SUB BX,AX       ;Generate Y0 - M x (X - X0)
                        MOV AX,BX       ;Save in AX
                                ;
IIF3:
                MOV CX,X2       ;X-position
                MOV DX,AX       ;Y-position
                                ;
                MOV col,CX      ;write dot
                MOV row,DX
                CALL wdot       ;OS/2 dot routine
                                ;
                INC X2          ;Next point
                MOV BX,X2
                CMP BX,X1       ;Ck X<= X1
                JBE DO1
                                ;
        POP SI
        POP DI
        POP DX
        POP CX
        POP BX
        POP AX
        POP DS
        RET
CONNL2  ENDP
                                ;
CLINE   ENDS
        END     CONNL2
```

Figure 2.9b (*Concluded*)

```
PAGE 55,132
TITLE SLOPELN - This program plots/prints sloped line (slopeln.asm)
;
;       DESCRIPTION: This program plots a sloped line in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the lines.
;
.8087
EXTRN   prtscr:FAR,scr_ld:FAR,cls:FAR,clsCGA:FAR,lineh:FAR,connl2:FAR
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
;       --------------------------------------------------
;               Graphics (printer) variables public
;       --------------------------------------------------
;
PUBLIC  in_buffer,bytesin,bytesout,in_buffer1,bytesin1
PUBLIC  in_buffer2,bytesin2,in_buffer3,bytesin3,in_buffer4
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,col1,N
PUBLIC  s,eight,eighty,four,shift1,scr_buffer
PUBLIC  sixforty,N4,ddd,w,b1
;       --------------------------------------------------
;
;       --------------------------------------------------
;               Graphics (screen) variables public
;       --------------------------------------------------
PUBLIC  xx,xxx,tr,lc,br,rc,no_line,blank,viohdl,PVBPtr1,physel1,waitf
PUBLIC  dstat,four,two,col,dummy,MASK1,MASK11,row,eighty,address
PUBLIC  OFFSET1,y,xb,xe
;       --------------------------------------------------
;
;       --------------------------------------------------
;               Sloped line public variables
;       --------------------------------------------------
PUBLIC  X0,X1,X2,Y0,Y1,Y2,M,NCOUNT,SIGN
;       --------------------------------------------------
;
;       --------------------------------------------------
;               Sloped line declarations
;       --------------------------------------------------
X0      dw      0                       ;x start
X1      dw      0                       ;x end
X2      dw      0                       ;dummy x
Y0      dw      0                       ;y start
Y1      dw      0                       ;y end
Y2      dw      0                       ;dummy y
NCOUNT  dw      0                       ;number points in line
SIGN    dw      0                       ;sign slope
M       dw      0                       ;scaled partial slope
;       --------------------------------------------------
;
;       --------------------------------------------------
```

Figure 2.10 The program slopeln.asm, which plots on the screen a connecting line from (row,col) = (25,25) to (175,275).

```
;               Screen display variables
;       --------------------------------------------------
;
viohdl  equ     0                        ;Required video handle
result  dw      0                        ;Completion code
action  equ     0                        ;Terminates current thread
tr      dw      0                        ;Top row screen clear
lc      dw      0                        ;Left column screen clear
br      dw      23                       ;Bottom row screen clear
rc      dw      79                       ;Right column screen clear
no_line dw      25                       ;Number lines scrolled
blank   dw      0007H                    ;Blank character pair
;
CGAm    label   FAR                      ;Video mode structure-CGA
lmodeE  dw      12                       ;Structure length
typeCGA db      00000111B                ;Mode identifier
colCGA  db      2                        ;Color option-Mode 5
txtcCGA dw      40                       ;text characters/line-ignore
txtrCGA dw      25                       ;text lines-ignore
hrCGA   dw      320                      ;horizontal resolution
vrCGA   dw      200                      ;vertical resolution
;
STDm    label   FAR                      ;Video mode structure-80x25
lmode80 dw      12                       ;Structure length
type80  db      00000001B                ;Mode identifier-Mode 3+
col80   db      4                        ;Color option
txtc80  dw      80                       ;text characters/line
txtr80  dw      25                       ;text lines
hr80    dw      720                      ;horizontal resolution
vr80    dw      400                      ;vertical resolution
;
kbd_buf db      80                       ;Keyboard buffer
lkbd_buf dw     $-kbd_buf                ;Length keyboard buffer
iowait  dw      0                        ;Wait for CR
kbdhdl  equ     0                        ;Keyboard handle
;
waitf   equ     1                        ;Screen waiting status
dstat   db      ?                        ;Returned status
;
PVBPtr1 label   FAR                      ;Video buffer structure
bufst1  dd      0B8000H                  ;Start physical address
buflen1 dd      4000H                    ;Buffer length
physel1 dw      0                        ;OS/2 screen buffer selector
;
MASK1   db      01H                      ;PEL byte mask
MASK11  dw      0001H                    ;Odd/even row mask
OFFSET1 dw      2000H                    ;Odd row buffer offset
four    dw      4
xx      dw      ?                        ;PEL modulo parameter
dummy   dw      ?                        ;80287 dummy "pop"
two     db      2
xxx     db      ?                        ;Output value
eighty  dw      80
row     dw      ?                        ;row
col     dw      ?                        ;column
address dw      ?                        ;Address screen dot
;
x       dw      ?                        ;Box col parameter
y       dw      ?                        ;Box row parameter
xb      dw      75                       ;Start column
xe      dw      150                      ;End column
yb      dw      25                       ;Start row
ye      dw      175                      ;End row
;
eight   dw      8
;
```

Figure 2.10 (*Continued*)

```
;       ----------------------------------------------------
;       Data area below is used for screen print routine.
;       ----------------------------------------------------
;
;
in_buffer       db      320 dup(0)      ;print buffer
bytesin dw      320                     ;CGA line
bytesout        dw      0               ;output count
in_buffer1      db      1BH,4BH,64D,01H ;printer setup
bytesin1        dw      4               ;count bytes In_buffer1
in_buffer2      db      0DH,0AH         ;LF/CR
bytesin2        dw      2               ;in_buffer2 byte count
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3               ;in_buffer3 byte count
in_buffer4      db      1BH,32H
;
dev_name        db      'LPT1',0        ;name of printer device
dev_hand        dw      0               ;device handle
dev_act         dw      0               ;
dev_size        dd      0               ;
dev_attr        dw      0               ;
dev_flag        dw      00000001b       ;Open File
dev_mode        dw      0000000011000001b       ;hdl private,deny none,w/o
dev_rsv         dd      0               ;reserved
;
N4              dw      ?
MM              db      40H,10H,04H,01H ;pel mask
w               db      128,64,32,16,8,4,2,1    ;pin weights
col1            db      320 dup(?)      ;column index-printer
b1              db      4 dup(?)
N               dw      ?               ;printer line
shift1          db      6,4,2,0
s               db      4 dup(?)        ;dup copies pel byte
ddd             dw      ?
sixforty        dw      640
scr_buffer      db      16384 dup(0)    ;temporary buffer--screen values
;
;
;               ----------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        call cls                        ;Clear screen
;
        @VioSetMode CGAm,viohdl         ;Set CGA Graphics mode
;
        call clsCGA                     ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer selector
        push physel1                    ;Save selector
        pop es                          ;Load selector into extra segment
;
        mov dh,25                       ;y-begin
        mov dl,175                      ;y-end
        mov bx,25                       ;x-begin
        mov cx,275                      ;x-end
        call connl2                     ;plot sloped line
;
        call scr_ld                     ;loads the temporory buffer
```

Figure 2.10 (*Continued*)

```
;
        @VioScrUnLock   viohdl          ;Unlock screen context
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
;
        @VioSetMode STDm,viohdl         ;80 x 25 alpha mode
;
        call prtscr                     ;prints temporary buffer
;
        @DosExit action,result          ;Terminate process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 2.10 (*Concluded*)

This program calls conn12 to plot the line on the screen in the usual fashion. Note that the assembler is case insensitive. Next the print of the graphics screen is accomplished. The nine variables needed by the connecting line module, conn12, are declared public and specified in the data segment of slopeln.asm. Figure 2.11 illustrates the sloped-line output.

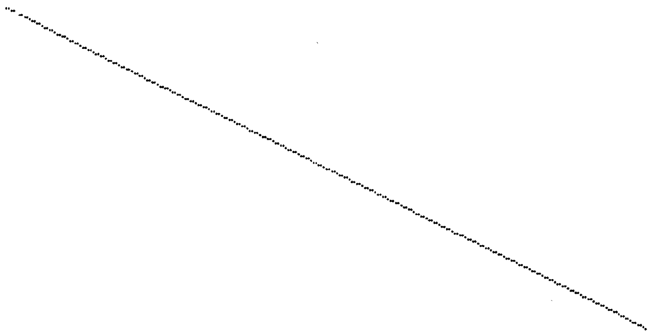


Figure 2.11 Screen dump of sloped connecting line from slopeln.asm.

Figure 2.12 illustrates a procedure bbox1.asm that plots a box based on general parameter inputs: (xb,yb) and (xe,ye) being the opposite corners of the box, with the first the upper left side and the second the lower right side. Figure 2.13 is the procedure linev, which is the corollary to lineh, and this procedure plots a vertical line. Figure 2.14 contains the contents of the library module GRAPHLIB.LIB. This module has all the graphics primitives and graphics printer screen dump modules.

```
PAGE 55,132
TITLE BBOX1 - This is the boxx module (bbox1.asm)
;
;       DESCRIPTION: This module generates a box in protected
;       mode.  It contains a procedures: boxx
;
.8087
;
EXTRN   y:WORD,yb:WORD,ye:WORD,x:WORD,xb:WORD,xe:WORD,lineh:FAR
EXTRN   row:WORD,col:WORD,wdot:FAR,linev:FAR
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg
        PUBLIC boxx
;
boxx    PROC    FAR
;
;       xb = x-begin,xe = x-end,yb = y-begin,ye = y-end
;
        mov ax,yb                       ;Top box line
        mov y,ax
        call lineh                      ;Draw top horizontal line
        mov ax,ye                       ;Bottom box line
        mov y,ax
        call lineh                      ;Draw bottom horizontal line
        mov ax,xb                       ;Left box line
        mov x,ax
        call linev                      ;Draw left vertical line
        mov ax,xe                       ;Right box line
        mov x,ax
        call linev                      ;Draw right vertical line
;
        ret
boxx    ENDP
;
;
CSEG    ENDS
        END     boxx
```

Figure 2.12 Routine bboxl.asm, which plots box and contains procedure boxx.

Finally, Figure 2.15 illustrates a program that exercises boxx, the procedure for plotting a box based on the module bbox1.asm. This program is called bbox.asm and links as follows:

```
[c:\)] link bbox

IBM Linker/2     Version 1.00
Copyright(c)     IBM Corporation 1987
Copyright(c)     Microsoft Corp 1983-1987. All rights reserved.

Run File [BBOX.EXE]:
List Files (NUL.MAP):
Libraries [.LIB]: doscalls + graphlib
Definitions File [NUL.DEF]:
```

```
PAGE 55,132
TITLE LLINEV - This is the linev module (llinev.asm)
;
;       DESCRIPTION: This module generates a vertical line in protected
;       mode.  It contains a procedure: linev
.8087
;
EXTRN   y:WORD,yb:WORD,ye:WORD,x:WORD,xb:WORD,xe:WORD
EXTRN   row:WORD,col:WORD,wdot:FAR
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg
        PUBLIC linev
;
;
linev   PROC    FAR
;
;       x = col position, yb = begin, ye = end
;
        mov ax,x                        ;Establish column for wdot
        mov col,ax
                                        ;
        mov ax,yb                       ;Establish start row
DO20:
        mov row,ax
        push ax                         ;Save row value
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall row
        inc ax                          ;Increment row
        cmp ax,ye                       ;Check end vertical line
        jle DO20
                                        ;
        ret
linev   ENDP
;
CSEG    ENDS
        END     linev
```

Figure 2.13 Vertical line procedure, linev.

```
BOXX.............bbox1              CLS..............graph1
CLSCGA...........graph1             LINEH............graph1
LINEV............llinev             PRTSCR...........prtscr
SCR_LD...........scrld              WDOT.............graph1


scrld           Offset: 00000010H  Code and data size: 29H
  SCR_LD

prtscr          Offset: 000000a0H  Code and data size: 233H
  PRTSCR

bbox1           Offset: 000006b0H  Code and data size: 2dH
  BOXX

llinev          Offset: 000007b0H  Code and data size: 1bH
  LINEV

graph1          Offset: 00000870H  Code and data size: 104H
  CLS             CLSCGA             LINEH              WDOT
```

Figure 2.14 Listing of GRAPHLIB.LIB, illustrating the screen graphics and screen print routines.

```
PAGE 55,132
TITLE BBOX - This program plots/prints a box using modules (bbox.asm)
;
;       DESCRIPTION: This program plots a box in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the lines.
;
.8087
EXTRN   prtscr:FAR,scr_ld:FAR,cls:FAR,clsCGA:FAR,boxx:FAR
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
;       -------------------------------------------------
;               Graphics (printer) variables public
;       -------------------------------------------------
;
PUBLIC  in_buffer,bytesin,bytesout,in_buffer1,bytesin1
PUBLIC  in_buffer2,bytesin2,in_buffer3,bytesin3,in_buffer4
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,col1,N
PUBLIC  s,eight,eighty,four,shift1,scr_buffer
PUBLIC  sixforty,N4,ddd,w,b1
;       -------------------------------------------------
;
;       -------------------------------------------------
;               Graphics (screen) variables public
;       -------------------------------------------------
PUBLIC  xx,xxx,tr,lc,br,rc,no_line,blank,viohdl,PVBPtr1,physel1,waitf
PUBLIC  dstat,four,two,col,dummy,MASK1,MASK11,row,eighty,address
PUBLIC  OFFSET1,y,xb,xe,x,yb,ye
;       -------------------------------------------------
;
;       -------------------------------------------------
;               Screen display variables
;       -------------------------------------------------
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     0                       ;Terminates current thread
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
hrCGA   dw      320                     ;horizontal resolution
vrCGA   dw      200                     ;vertical resolution
```

Figure 2.15 Program that graphs box on screen and then dumps output to printer.

```
;
STDm    label   FAR                     ;Video mode structure-80x25
lmode80 dw      12                      ;Structure length
type80  db      00000001B               ;Mode identifier-Mode 3+
col80   db      4                       ;Color option
txtc80  dw      80                      ;text characters/line
txtr80  dw      25                      ;text lines
hr80    dw      720                     ;horizontal resolution
vr80    dw      400                     ;vertical resolution
;
kbd_buf db      80                      ;Keyboard buffer
lkbd_buf dw     $-kbd_buf               ;Length keyboard buffer
iowait  dw      0                       ;Wait for CR
kbdhdl  equ     0                       ;Keyboard handle
;
waitf   equ     1                       ;Screen waiting status
dstat   db      ?                       ;Returned status
;
PVBPtr1 label   FAR                     ;Video buffer structure
bufstl  dd      0B8000H                 ;Start physical address
buflenl dd      4000H                   ;Buffer length
physell dw      0                       ;OS/2 screen buffer selector
;
MASK1   db      01H                     ;PEL byte mask
MASK11  dw      0001H                   ;Odd/even row mask
OFFSET1 dw      2000H                   ;Odd row buffer offset
four    dw      4
xx      dw      ?                       ;PEL modulo parameter
dummy   dw      ?                       ;80287 dummy "pop"
two     db      2
xxx     db      ?                       ;Output value
eighty  dw      80
row     dw      ?                       ;row
col     dw      ?                       ;column
address dw      ?                       ;Address screen dot
;
x       dw      ?                       ;Box col parameter
y       dw      ?                       ;Box row parameter
xb      dw      75                      ;Start column
xe      dw      150                     ;End column
yb      dw      25                      ;Start row
ye      dw      175                     ;End row
;
eight   dw      8
;
;       -----------------------------------------------------
;       Data area below is used for screen print routine.
;       -----------------------------------------------------
;
;
in_buffer       db      320 dup(0)      ;print buffer
bytesin dw      320                     ;CGA line
bytesout        dw      0               ;output count
in_buffer1      db      1BH,4BH,64D,01H ;printer setup
bytesin1        dw      4               ;count bytes In_buffer1
in_buffer2      db      0DH,0AH         ;LF/CR
bytesin2        dw      2               ;in_buffer2 byte count
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3               ;in_buffer3 byte count
in_buffer4      db      1BH,32H
;
dev_name        db      'LPT1',0        ;name of printer device
dev_hand        dw      0               ;device handle
dev_act         dw      0               ;
dev_size        dd      0               ;
dev_attr        dw      0               ;
```

Figure 2.15 *(Continued)*

```
dev_flag        dw      00000001b         ;Open File
dev_mode        dw      0000000011000001b       ;hdl private,deny none,w/o
dev_rsv         dd      0                 ;reserved
;
N4              dw      ?
MM              db      40H,10H,04H,01H ;pel mask
w               db      128,64,32,16,8,4,2,1    ;pin weights
coll            db      320 dup(?)        ;column index-printer
b1              db      4 dup(?)
N               dw      ?                 ;printer line
shiftl          db      6,4,2,0
s               db      4 dup(?)          ;dup copies pel byte
ddd             dw      ?
sixforty        dw      640
scr_buffer      db      16384 dup(0)      ;temporary buffer--screen values
;
;
;               ----------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        call cls                        ;Clear screen
;
        @VioSetMode CGAm,viohdl         ;Set CGA Graphics mode
;
        call clsCGA                     ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer selector
        push physel1                    ;Save selector
        pop es                          ;Load selector into extra segment
;
        call boxx                       ;generate box
;
        call scr_ld                     ;loads the temporory buffer
;
        @VioScrUnLock   viohdl          ;Unlock screen context
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
;
        @VioSetMode STDm,viohdl         ;80 x 25 alpha mode
;
        call prtscr                     ;prints temporary buffer
;
        @DosExit action,result          ;Terminate process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 2.15 *(Concluded)*

Figure 2.16 presents the output for the box. It assumes the following corner values based on the data segment specification:

$$(xb, yb) = (75,25)$$

and

$$(xe, ye) = (150,175)$$

It is important to recognize that the method for accessing the OS/2 physical display buffer is initially cumbersome since the display context must first be locked. Good programming practice, however, would implement this step only once during execution of a given process or thread. Then full-screen DMA is permitted. Although the advantage of using multitasked display contexts under OS/2, for example, is lost, the programmer still has access to the very large address space provided by OS/2.

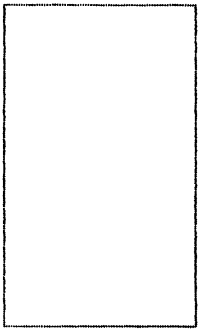


Figure 2.16 Box screen print from bbox.asm (Figure 2.15).

It is the benefits of multitasking and memory access that highlight OS/2's attributes. If animation or rapid update of the screen context is needed, the application must be structured to maximize its screen access while continuing to take advantage of other task-sharing arrangements and memory requirements in a multitasking environment.

2.4 SOFTWARE DESIGN

No introductory programming discussion would be complete without mentioning software design techniques. Three principals come to mind:

1. Modular code
2. Top-down design
3. Structured programming

These form the foundation for developing optimized computer programs [6,7].

We have seen examples of the use of modular code in the examples presented so far. Basically, programs have been developed on existing independent smaller modules (such as prtscr, bbox, lineh, linev, . . .).These modules may be linked as separately assembled (in this case) routines or appear in libraries. The entire notion of the API embodies a modular approach to handling system services.

Top-down design is somewhat straightforward and starts with a high-level statement of the problem to be solved. From this approach a flowchart can be developed or, alternatively, a written language rendition of what the program will accomplish. These two approaches lead naturally into program development. It is not the intention of this book to teach design fundamentals; hence we will usually confine ourselves to the actual code implemented in each instance. It is assumed that the reader can generate design artifacts in either flowchart format or pseudo-code. General guidelines tend to suggest flowcharts when program dynamics are particularly important. As size becomes a factor, the interfaces tend to dominate programming considerations and pseudo-code becomes desirable.

Finally, we mentioned structured programming. Assembler is intrinsically unstructured because of the conditional and unconditional jump instructions. Guidelines exist, however, that permit structured techniques with assembly language and very understandable code results [8,9]. As Martin points out (reference 6), an excellent starting point for top-down design is the development of a structure chart that links each module in static fashion. Structured code accomplishes similar design tasks by forcing the designer to limit access to each module. Parameter passing becomes very orderly and the variables used within a module are maintained locally except for those passed externally. The flow of execution is orderly and downward avoiding unrestrained jumps within the code. In BASIC and C, for example, the goto instruction is avoided. Entry points are accessed in traceable fashion. We will discuss C programming in a subsequent chapter, and the syntax of the language naturally lends itself to structured code. In assembly language the use of syntax of the form

```
   ...
   cmp  ax,parm
   jle  ELSE1
        ...
        jmp  IIF1
ELSE1:
   ...
IIF1:
   ...
```

implements the familiar IF. . .THEN. . .ELSE. . . structure. Note the use of unconditional jumps to do this. The reader must, of course, recognize that any decision logic in a higher-level language results in conditional or unconditional jumps at the assembler or machine code level. The fact that we can structure such an assembler, however, yields significant improvement in clarity and understanding.

2.5 SUMMARY

The goal of this chapter was to introduce assembler programming for OS/2 with a particular emphasis on the techniques needed to access the Application Program Interface. It was the use API services, as demonstrated through specific examples, that we intended to emphasize. We developed examples of printer and display access within the API context. A set of program modules was written that allow the user to obtain a screen dump for the CGA graphics screen. This is similar to the familiar DOS GRAPHICS.COM routine except that they are not terminate and stay resident (TSR) programs but must be called as active program modules. Similarly, they must be linked to the user program either as stand-alone modules or from the library GRAPHLIB.LIB established in the chapter.

It is clear from the chapter that the API must be accessed in more structured fashion than the normal interrupt call, where free access to the general-purpose registers is permitted. The advantages of accessing memory above 1 MB are provided by OS/2, and the API architecture is a consequence of this process. Also, the features of multitasking require a Protected Mode structure. Hence these two aspects of programming with Intel hardware such as the 80286 cause programming structures such as the API to become essential. This, then, is the justification for learning the API and using OS/2. Eventually, when the programming applications reach a size where only OS/2 can offer the memory allocation or severe multitasking constraints are placed on the user, DOS will inevitably be replaced by OS/2. Thus the programming techniques of this chapter are the beginning approaches to learning how to manage IBM's next-generation microcomputer operating system.

REFERENCES

1. Iacobucci, E., *OS/2 Programmer's Guide,* Osborne McGraw-Hill, Berkeley, CA, 1988.
2. Letwin, G., *Inside OS/2,* Microsoft Corporation, Redmond, WA, 1988.
3. *Operating System/2 Programmer's Toolkit,* Programmer's Guide, International Business Machines Corporation, Boca Raton, FL, 1987.
4. IBM Macro Assembler/2, Language Reference and Fundamentals, International Business Machines Corporation, Boca Raton, FL, 1987.
5. *EPSON FX-85 and FX-185 Printers User's Manual,* Seiko Epson Corporation, Nagano, Japan, 1985.
6. Martin, J., and McClure, C., *Structured Techniques for Computing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.
7. Alagic', S., and Arbib, M. A., *The Design of Well-Structured and Correct Programs,* Springer-Verlag, New York, 1978.
8. Scanlon, L., *IBM PC and XT Assembly Language,* Brady Communications Company, Inc., New York, Chap. 11, 1985.
9. Godfrey, J. T., *IBM Microcomputer Assembly Language: Beginning to Advanced,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

PROBLEMS

2.1 Output of a character in the printer graphics mode causes the pins in the print head to fire, depending on the value of the character (0-255). Which pins fire for the Epson

FX-85 for the following values: (a) 174; (b) 203; (c) 85? Assume that the top pin is number 8.

2.2 The macro call

```
@VioGetPhysBuf    PVBPrtl,rsv
```

is an example of the OS/2 access to the physical buffer selector (contained in the structure PVBPrtl). If the calling sequence for VIOGETPHYSBUF, the API routine, is given by

```
EXTRN    VioGetPhysBuf:FAR

PUSH@    Label Data structure (PVBPrtl)
PUSH     word reserved
CALL     VIOGETPHYSBUF
```

write a macro to achieve the call indicated above. Use the macros defined in the chapter: @define, @pushw, and @pushs.

2.3 As in Problem 2.2, if the macro @VioScrLock has the calling sequence

```
EXTRN    VioScrLock:FAR

PUSH     word   waitflag
PUSH@    BYTE   status
PUSH     word   handle
CALL     VIOSCRLOCK
```

define a macro to achieve the call using the macros defined in the chapter: @define, @pushw, and @pushs.

2.4 The EGA graphics on the IBM Monochrome Display is a 640 x 350 graphics mode (ModeHexF). The first eight pixels on the screen are defined by the contents of memory in location A0000H. The screen buffer is mapped into two planes (Map0 and Map2). Assume that only Map0 is specified (a default value of white on the normal background) and this subtends the first 28,000 bytes at A0000H. If each bit corresponds to 0 (black) or 1(white), what changes would be needed to the programs in this chapter to allow the code to properly access the EGA screen buffer?

2.5 Write a code fragment that generates the note C (5000 Hz) for 1 second.

2.6 Write a code fragment that plots a straight line from (row, col) = (25,75) to (165,235). Use conn12.

2.7 Write a code fragment that beeps the speaker for 5 seconds (1000 Hz) following the striking of a key on the keyboard.

2.8 Is it possible to program OS/2 using simply the OS/2 supplied by IBM? Explain.

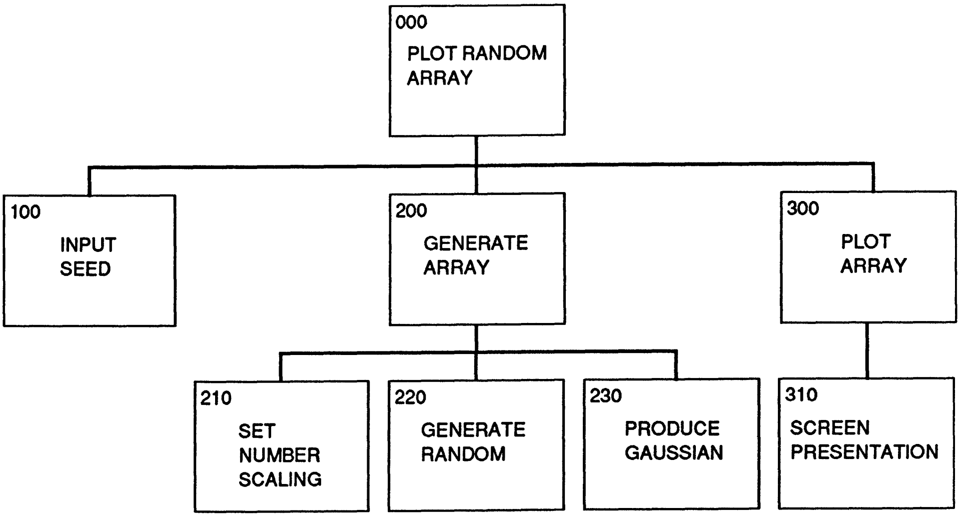
2.9 When accessing the physical screen buffer, a segment selector was obtained using VioGetPhysBuf. This selector was pushed and popped into es, the extra segment register. Screen buffer address locations were then accessed using, for example, a segment override:

```
es: [bp]
```

where bp was the offset in the screen buffer. What key assumption permits this addressing scheme, and how is it implemented?

2.10 Structure labels used by the API Toolkit services represent FAR or NEAR locations? Why?

2.11 A structure chart lays out the hierarchy of control for a program. Consider a structure chart that calculates a Gaussian random array and plots it on the screen. What deficiency exists in this presentation assuming that the functional characteristics of each block are accurate and complete?



2.12 In ordinary printing the line spacing is 1/6 inch. The command

```
ESC A (n)
```

produces a line spacing of n/72 of an inch. What command is needed in prtscr to return the printer to normal spacing?

2.13 A key aspect of OS/2 programming is that the device driver Interrupt Service Routine (ISR) is the only type of program authorized to receive hardware interrupts. Each device driver can have an ISR to process the device interrupts. When the interrupt routine is given control, it has access to the GDT but not any specific LDT. The routines must rely on the DevHlp services to access application buffers. These routines run at the highest level of the kernel mode. Are they preemptable by OS/2 task switches? Explain.

2.14 Explain why scr_ld must be called with the screen locked and why prtscr should not be called with the screen locked.

2.15 When drawing a line segment from

$$(\text{row,column}) = (10,10)$$

to

$$(\text{row,column}) = (20,110)$$

how many small straight-line raster segments will be visible in the actual plotted line?

2.16 DosExitCritSec executes after DosEnterCritSec and reenables thread switching for the current process. A count of the number of outstanding DosEnterCritSec requests is maintained. When are these functions likely to be used?

2.17 Observe that DosExit is used only at the conclusion of the main calling module. What are the criteria for this exit, and how should other modules be terminated?

3 Memory Management and Multitasking with Assembler

OS/2 is first and foremost a multitasking and memory management operating system. It was developed around the Intel 80286 Protected Mode hardware implementation and employs the features of this hardware, such as segment selector protection mechanisms, to achieve multitasking operation. The goal of this chapter is to acquaint the reader with these features of OS/2 in an assembler programming environment.

3.1 MEMORY MANAGEMENT AND MULTITASKING

Memory management and multitasking represent static and dynamic aspects of programming and are intimately tied to the protection mechanisms of the 80286 for the OS/2 environment. The 80286 has three basic protection aspects:

1. Isolation of system software from end-user applications
2. Isolation of users from each other
3. Data-type checking

In terms of a ringed picture, the 80286 provides a four-level increasingly privileged protection mechanism that isolates applications from the various layers of system software. To understand protection on the 80286, we must begin with its basic parts: segments and tasks. It is interesting that this division leads naturally into memory management and multitasking. Both are interrelated as pointed out above and yield a full Protected Mode picture.

The following illustrates a complete 80286, descriptor cache register (as we have seen in Chapter 1):

Selectors

15 0	47 40	39 16	15 0
Segment Register	Access Rights	Segment Base Address	Segment Size
Program Visible (loaded by program)		Program Invisible (loaded by CPU)	

The important features of this register that should be recognized are that in the invisible portion of the register both the access rights byte (with segment type and privilege level) and size are used to determine protection priority and the segment base address confirms that the selected segment is valid. Hence a hardware implementation is used to restrict segment access.

By way of review, dynamically each task consists of up to four active segments (with segment registers CS, SS, DS, and ES), as we have seen. The methodology above is used to control segment protection, and since the hardware can dynamically check protection data, this extends into the task realm. The protection data are used at two different times dynamically: upon loading a segment register and upon each reference to the selected segment. Each task can address up to a gigabyte (2^{14}–2 segments of up to 65,536 bytes) of virtual memory defined by the task's LDT and the system GDT. The task's private address space (LDT) can occupy up to one-half this memory. The rest is defined by the GDT. The CPU has a set of base and limit registers that point to the GDT and the LDT of the currently running task. These registers exist for CS, SS, DS, and ES as defined for the task. An active task can only load selectors that reference segments defined by its LDT descriptors or the GDT. Since a task cannot reference descriptors in other LDTs, protection violations cannot occur.

All descriptor tables have a limit used by the protection hardware to ensure that correct address space size allocation occurs. This ensures that separation of tasks takes place. The third ingredient in this protection mechanism is the implementation of privilege-level checks based on the access rights byte assigned privilege level. The 80286 privilege-levels are:

1. Level 0: The kernel (most trusted) includes memory management, task isolation, multitasking, intertask communication, and I/O resource control.
2. Level 1: System Services (next most trusted level) provides high-level func-

tions such as file access scheduling, character I/O, data communications, and resource allocation.

3. Level 2: Custom Extensions (third most trusted level) allows standard sys tem software to be customized: database managers, logical file access services, and so on.
4. Level 3: Applications (least trusted) include normal programming environment.

It is important to recognize the protection intrinsic to this hierarchy. Functions catastrophic to system failure are more tightly protected. This ensures that tasking can continue to execute in the event of single-task failures. Descriptor privilege, including code segment privilege, is assigned when the descriptor is created. The system designer assigns privilege directly when the system is constructed or indirectly using a loader. This is how OS/2 functions. OS/2 was designed with appropriate access byte values associated with the kernel and each system service segment. The linker (and loader) assign privilege at a higher level (less trusted) to all applications software subsequently developed. The programmer does not normally have access to this privilege specification, and we will not tamper with this mechanism. These are, however, the techniques used by OS/2 to provide memory management and multitasking protection (segment base checking, limit checks, and access rights byte validation).

Task privilege is dynamic and can change only when control transfers from one code segment to another. Descriptor privilege, including code segment privilege, is assigned when the descriptor is created. Clearly, as a task executes, the privilege level of the task must correspond to that of the code segment currently executing. Hence such privilege is dynamic. Several general rules apply:

1. Data access is restricted to those segments whose privilege level is the same or less privileged (numerically greater) than the current privilege level (CPL).
2. Direct code access is restricted to code segments of equal privilege.
3. A gate is required for access to code at more privileged levels.

A gate is a control descriptor consisting of four words. It is used to redirect execution to a different segment at a more privileged level. These call gate descriptors are used by control instructions (call and jump instructions) in much the same fashion as segment descriptors are used during normal transfer of code segments at the same level.

Above, we have briefly examined the background for memory management and multitasking. Here the emphasis has been on hardware features and privileged access (and summarizes the discussion of protection in Chapter 1). It is important to recognize that these are hardware features in the Intel chips. Multitasking and dynamic memory allocation are particularly well suited to the Intel segmented memory [1]. CPUs with less protection, such as simple system and user privilege, are less ideally suited for protection during multitasking [2].

3.2 MEMORY MANAGEMENT ACTIVITIES

This section is intended to illustrate simple programming concepts related to memory management. Under OS/2 the management of memory is accomplished dynamically using the API calls. These calls all execute primarily at level 0 and permit memory management of higher-level code, such as applications developed by the user.

3.2.1 Creating and Accessing Memory Segments

The simplest activity associated with memory management is the creation, access, and destruction of a memory segment. The code associated with accomplishing this action is represented as follows:

```
...
msize dw (size in bytes of segment)
msel1 dw ?
mflag dw 0000000000000111B ;sharable, discardable
...
@DosAllocSeg msize, msel1, mflag
...
@DosFreeSeg msel1
...
```

Figure 3.1 illustrates the program twolnm.asm, a modified version of the module twoln.asm. This program creates a temporary buffer in place of scr_buffer (the buffer used earlier) and uses the representative code shown above to accomplish this action. The parameter msize has been specified to be 16,384 bytes, the buffer size for the screen buffer. A selector, msel1, is obtained from the call to @DosAllocSeg, and the specified flag indicates that the selector's segment is sharable among code segments other than the one creating the segment, and may be discarded. Finally, once the print screen function is performed, the segment is discarded using the @DosFreeSeg call.

Figure 3.2 contains the code for scr_ldm, a modified version of scr_ld. Since the physical screen buffer segment must be accessed using ES (based on selector physel1) and the newly created temporary buffer (using the selector msel1), the extra segment must share its usage between these two selected segments. To accomplish this, es is pushed after each reference to the physical screen buffer, loaded with msel1, and then used to access the temporary buffer. Following this action, the physical screen buffer is popped back into es and another access of this buffer accomplished. Intermediate buffer values are passed using AX. (*Note*: We use upper and lowercase references to the 80286 registers in interchangeable fashion.)

Figure 3.3 provides the program for the module prtscrm, a modified version of prtscr. The only change in this routine is the earlier access to scr_buffer:

```
PAGE 55,132
TITLE TWOLNM - This program plots/prints 2 lines (twolnm.asm)
;
;       DESCRIPTION: This program plots two lines in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the lines.
;       The routine uses DYNAMIC MEMORY ALLOCATION.
;
.8087
EXTRN   prtscrm:FAR,scr_ldm:FAR,cls:FAR,clsCGA:FAR,lineh:FAR
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
;       -------------------------------------------------
;               Graphics (printer) variables public
;       -------------------------------------------------
;
PUBLIC  in_buffer,bytesin,bytesout,in_buffer1,bytesin1
PUBLIC  in_buffer2,bytesin2,in_buffer3,bytesin3,in_buffer4
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,col1,N
PUBLIC  s,eight,eighty,four,shift1
PUBLIC  sixforty,N4,ddd,w,b1
;       -------------------------------------------------
;
;       -------------------------------------------------
;               Graphics (screen) variables public
;       -------------------------------------------------
PUBLIC  xx,xxx,tr,lc,br,rc,no_line,blank,viohdl,PVBPtr1,physel1,waitf
PUBLIC  dstat,four,two,col,dummy,MASK1,MASK11,row,eighty,address
PUBLIC  OFFSET1,y,xb,xe
;       -------------------------------------------------
;               Dynamic Memory Allocation
;       -------------------------------------------------
PUBLIC  msel1
;       -------------------------------------------------
;
;       -------------------------------------------------
;               Screen display variables
;       -------------------------------------------------
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     0                       ;Terminates current thread
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
```

Figure 3.1 The program twolnm.asm (a modified version of twoln.asm), which creates a buffer for the screen memory.

```
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
hrCGA   dw      320                     ;horizontal resolution
vrCGA   dw      200                     ;vertical resolution
;
STDm    label   FAR                     ;Video mode structure-80x25
lmode80 dw      12                      ;Structure length
type80  db      00000001B               ;Mode identifier-Mode 3+
col80   db      4                       ;Color option
txtc80  dw      80                      ;text characters/line
txtr80  dw      25                      ;text lines
hr80    dw      720                     ;horizontal resolution
vr80    dw      400                     ;vertical resolution
;
kbd_buf db      80                      ;Keyboard buffer
lkbd_buf dw     $-kbd_buf               ;Length keyboard buffer
iowait  dw      0                       ;Wait for CR
kbdhdl  equ     0                       ;Keyboard handle
;
waitf   equ     1                       ;Screen waiting status
dstat   db      ?                       ;Returned status
;
PVBPtr1 label   FAR                     ;Video buffer structure
bufst1  dd      0B8000H                 ;Start physical address
buflen1 dd      4000H                   ;Buffer length
physel1 dw      0                       ;OS/2 screen buffer selector
;
MASK1   db      01H                     ;PEL byte mask
MASK11  dw      0001H                   ;Odd/even row mask
OFFSET1 dw      2000H                   ;Odd row buffer offset
four    dw      4
xx      dw      ?                       ;PEL modulo parameter
dummy   dw      ?                       ;80287 dummy "pop"
two     db      2
xxx     db      ?                       ;Output value
eighty  dw      80
row     dw      ?                       ;row
col     dw      ?                       ;column
address dw      ?                       ;Address screen dot
;
x       dw      ?                       ;Box col parameter
y       dw      ?                       ;Box row parameter
xb      dw      0                       ;Start column
xe      dw      319                     ;End column
yb      dw      25                      ;Start row
ye      dw      175                     ;End row
;
eight   dw      8
;
;       ------------------------------------------------------
;       Data area below is used for screen print routine.
;       ------------------------------------------------------
;
;
in_buffer       db      320 dup(0)      ;print buffer
bytesin dw      320                     ;CGA line
bytesout        dw      0               ;output count
in_buffer1      db      1BH,4BH,64D,01H ;printer setup
bytesin1        dw      4               ;count bytes In_buffer1
in_buffer2      db      0DH,0AH         ;LF/CR
bytesin2        dw      2               ;in_buffer2 byte count
in_buffer3      db      1BH,41H,08H
bytesin3        dw      3               ;in_buffer3 byte count
in_buffer4      db      1BH,32H
```

Figure 3.1 (*Continued*)

```
;
dev_name        db      'LPT1',0        ;name of printer device
dev_hand        dw      0               ;device handle
dev_act         dw      0               ;
dev_size        dd      0               ;
dev_attr        dw      0               ;
dev_flag        dw      00000001b       ;Open File
dev_mode        dw      0000000011000001b       ;hdl private,deny none,w/o
dev_rsv         dd      0               ;reserved
;
N4              dw      ?
MM              db      40H,10H,04H,01H ;pel mask
w               db      128,64,32,16,8,4,2,1    ;pin weights
col1            db      320 dup(?)      ;column index-printer
b1              db      4 dup(?)
N               dw      ?               ;printer line
shift1          db      6,4,2,0
s               db      4 dup(?)        ;dup copies pel byte
ddd             dw      ?
sixforty        dw      640
;
;               ------------------------------------------------
;               Data area below used for dynamic memory allocation
;               ------------------------------------------------
msize           dw      16384           ;temporary buffer-screen values
msel1           dw      ?               ;allocated selector
mflag           dw      0000000000000111B       ;segment sharable, discardable
;               ------------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        call cls                        ;Clear screen
;
        @VioSetMode CGAm,viohdl         ;Set CGA Graphics mode
;
        call clsCGA                     ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer selector
        push physel1                    ;Save selector
        pop es                          ;Load selector into extra segment
;
        mov ax,0
        mov y,ax
        call lineh                      ;Draw line
        mov ax,100
        mov y,ax
        call lineh                      ;draw second line
;
        @DosAllocSeg msize,msel1,mflag  ;allocate temporary buffer
;
        call scr_ldm                    ;loads the temporory buffer
;
        @VioScrUnLock   viohdl          ;Unlock screen context
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
;
        @VioSetMode STDm,viohdl         ;80 x 25 alpha mode
;
        call prtscrm                    ;prints temporary buffer
```

Figure 3.1 (*Continued*)

```
;
        @DosFreeSeg msell               ;free allocated space
;
        @DosExit action,result          ;Terminate process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.1 (*Concluded*)

```
PAGE 55,132
TITLE SCRLDM -- This routine loads the screen print buffer (scrldm.asm)
;
;       DESCRIPTION:  This routine accompanies prtscr to load
;       and print the screen in 320 x 200 mode.
;       The prtscr buffers are assumed loaded.  This is an OS/2
;       routine.  This routine uses DYNAMIC MEMORY ALLOCATION through msell.
;
EXTRN   msell:WORD
;
        .sall
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        PUBLIC  scr_ldm
scr_ldm PROC    FAR
        ASSUME  CS:CSEG
                                        ;
        mov cx,100                      ;no. of raster pairs
        mov di,0                        ;index to screen buffer
        mov si,0                        ;index to dummy array
DO55:
        push cx
        mov cx,80                       ;raster row length
DO56:
        mov al,es:[di]                  ;load even row physical buffer
        mov ah,es:[di+2000H]            ;odd row physical buffer
;
        push es
        mov es,msell
        mov es:[si],al                  ;load even rows
        mov es:[si+80],ah               ;odd rows
        pop es
;
        inc si
        inc di
        loop DO56
                                        ;
        add si,80                       ;skip to next double set
        pop cx
        loop DO55
                                        ;
        ret
scr_ldm ENDP
CSEG    ENDS
        END
```

Figure 3.2 The modified scrld.asm (scrldm.asm) used with the program wolnm.asm.

```
PAGE    55,132
TITLE   prtscrm - print screen (prtscrm.asm)
;
;       DESCRIPTION: This routine prints the screen in
;       320 x 200 CGA mode.  This routine uses DYNAMIC
;       MEMORY ALLOCATION.  This routine needs the
;       following data items in the calling routine
;       data segment:
;
;               ------------------------------------------
;               in_buffer       db      320 dup(0)
;               bytesin         dw      320
;               bytesout        dw      0
;               in_buffer1      db      1BH,4BH,64D,01H
;               bytesin1        dw      4
;               in_buffer2      db      0DH,0AH
;               bytesin2        dw      2
;               in_buffer3      db      1BH,41H,08H
;               bytesin3        dw      3
;               in_buffer4      db      1BH,32H
;                                       ;
;               dev_name        db      'LPT1',0
;               dev_hand        dw      0
;               dev_act         dw      0
;               dev_size        dd      0
;               dev_attr        dw      0
;               dev_flag        dw      00000001b
;               dev_mode        dw      0000000011000001b
;               dev_rsv dd      0
;                                       ;
;               MM              db      40H,10H,04H,01H
;               w               db      128,64,32,16,8,4,2,1
;               col1            db      320 dup(?)
;               N               dw      ?
;               N4              dw      ?
;               s               db      4 dup(?)
;               shift1          db      6,4,2,0
;               eight           dw      8
;               eighty          dw      80
;               b1              db      4 dup(?)
;               four            dw      4
;               ddd             dw      ?
;               sixforty        dw      640
;               ----------------------------------
;
EXTRN   MM:BYTE,w:BYTE,col1:BYTE
EXTRN   in_buffer:BYTE,in_buffer1:BYTE,in_buffer2:BYTE
EXTRN   in_buffer3:BYTE,in_buffer4:BYTE
EXTRN   bytesin:WORD,bytesin1:WORD,bytesin2:WORD,bytesin3:WORD
EXTRN   bytesout:WORD,dev_name:BYTE,dev_hand:WORD
EXTRN   dev_act:WORD,dev_size:DWORD,dev_attr:WORD
EXTRN   dev_flag:WORD,dev_mode:WORD,dev_rsv:DWORD
EXTRN   N:WORD,N4:WORD
EXTRN   eighty:WORD,eight:WORD,four:WORD,s:BYTE,shift1:BYTE
EXTRN   ddd:WORD,b1:BYTE,sixforty:WORD,msel1:WORD
;
IF1
        include sysmac.inc
ENDIF
;
        .sall
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        PUBLIC  prtscrm
```

Figure 3.3 The modified prtscr.asm (prtscm.asm) used with the program twolnm.asm.

```
prtscrm PROC    FAR
        ASSUME  CS:CSEG
                                        ;open device
@DosOpen dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag,dev_mode,dev_rsv
        cmp ax,0
        je ELSE1
                                        ;Exit
           ret
ELSE1:
                                        ;initialize device
        @DosWrite dev_hand,in_buffer3,bytesin3,bytesout
        @DosWrite dev_hand,in_buffer4,bytesin2,bytesout
                                        ;
        mov dx,25                       ;number print lines(+1)
        mov si,0                        ;index to 8 row block
LOOP1:
        push dx                         ;preserve dx
        push si                         ;preserve block count
        mov ax,si
        mul sixforty                    ;640 block size
        mov N,ax                        ;Save in N
                                        ;
        call ldarray
                                        ;
        mov di,0                        ;initialize 320 column counter
        mov cx,80                       ;count of column bytes
LOOP2:
        mov al,col1[di]                 ;column 1 from byte
        mov in_buffer[di],al            ;load print buffer
        mov al,col1[di+1]               ;column 2 from byte
        mov in_buffer[di+1],al          ;load print buffer
        mov al,col1[di+2]               ;column 3 from byte
        mov in_buffer[di+2],al          ;load print buffer
        mov al,col1[di+3]               ;column 4 from byte
        mov in_buffer[di+3],al          ;load print buffer
        add di,four                     ;increment column index
        loop LOOP2
;
                                        ;write print row
        @DosWrite dev_hand,in_buffer1,bytesin1,bytesout
        @DosWrite dev_hand,in_buffer,bytesin,bytesout
        @DosWrite dev_hand,in_buffer2,bytesin2,bytesout
                                        ;
        pop si                          ;recall block count
        pop dx                          ;recall print line count
        dec dx                          ;decrement count
        inc si                          ;increase block count
        cmp dx,0                        ;check 25 lines printed
        jle DII1
           jmp LOOP1
DII1:
        @DosClose dev_hand              ;close print device
        ret
prtscrm endp
;
ldarray PROC    NEAR
;
;       N is the printer row # - 640 byte intervals [0,24]
;       MM[0] = 40H,...,MM[3] = 01H (pel mask)
;       w[0] = 128,w[1] = 64,...,w[7] = 1
;
        mov si,0                        ;column count initialization
        mov cx,320                      ;320 columns
DO110:
        mov al,0                        ;clear print buffer
        mov col1[si],al
        inc si                          ;increment column count
        loop DO110
```

Figure 3.3 *(Continued)*

```
                                            ;
        mov si,0                            ;index into 80 bytes/row
        mov N4,si                           ;N4 = row byte block count
        mov dx,80                           ;counter - row bytes
DO111:
                                            ;
        mov di,0                            ;raster row counter (1 of 8)
        mov ddd,di                          ;80 block counter
        mov cx,8                            ;raster row counter
DO112:
        push cx                             ;preserve row count
        mov bp,ddd                          ;bp = # 80 byte blocks
        add bp,N                            ;add printer line count
;
        push es
        push msel1
        pop es                              ;load address screen buffer
        mov al,es:[bp+si]                   ;4 pel bytes
        pop es
;
        mov s[0],al                         ;1st copy
        mov s[1],al                         ;2nd copy
        mov s[2],al                         ;3rd copy
        mov s[3],al                         ;4th copy
                                            ;
        and al,MM[0]                        ;1st pel mask
        mov cl,shift1[0]                    ;load 1st pel shift
        shr al,cl                           ;shift right
        mov ah,0                            ;clear upper
        mul w[di]                           ;multiply by weight (row)
        mov b1[0],al                        ;save 1st printer column
                                            ;
        mov al,s[1]                         ;load 2nd pel
        and al,MM[1]                        ;mask 2nd pel
        mov cl,shift1[1]                    ;load 2nd pel shift
        shr al,cl                           ;shift right
        mov ah,0                            ;clear upper
        mul w[di]                           ;multiply by weight (row)
        mov b1[1],al                        ;save 2nd printer column
                                            ;
        mov al,s[2]                         ;load 3rd pel
        and al,MM[2]                        ;mask 3rd pel
        mov cl,shift1[2]                    ;load 3rd pel shift
        shr al,cl                           ;shift right
        mov ah,0                            ;clear upper
        mul w[di]                           ;multiply by weight (row)
        mov b1[2],al                        ;save 3rd printer column
                                            ;
        mov al,s[3]                         ;load 4th pel
        and al,MM[3]                        ;mask 4th pel
        mov cl,shift1[3]                    ;load 4th pel shift
        shr al,cl                           ;shift right
        mov ah,0                            ;clear upper
        mul w[di]                           ;multiply by weight (row)
        mov b1[3],al                        ;save 4th printer column
                                            ;
        push bx                             ;preserve bx
        mov bx,N4                           ;counter into print buffer
        mov al,b1[0]                        ;load column N4
        add col1[bx],al
        mov al,b1[1]                        ;load column N4+1
        add col1[bx+1],al
        mov al,b1[2]                        ;load column N4+2
        add col1[bx+2],al
        mov al,b1[3]                        ;load column N4+3
        add col1[bx+3],al
```

Figure 3.3 (*Continued*)

```
        pop bx
                                        ;
        pop cx                          ;recall print block row index
        inc di                          ;increase print block row counter
        add ddd,80                      ;increase byte count
        dec cx                          ;decrease row bound
        cmp cx,0
        jle DO133
           jmp DO112
DO133:
        add N4,4                        ;add 4 columns to index
        dec dx                          ;decrement row bytes
        inc si                          ;increase screen buffer index
        cmp dx,0
        jle DII2
           jmp DO111
DII2:
        ret
ldarray ENDP
;
CSEG    ENDS
        END
```

Figure 3.3 (*Concluded*)

```
...
push  bx
lea  bx,scr_buffer[0]
add  bp,bx
mov  al,ds:[bp+si]
pop  bx
...
```

This now reads

```
...
push  es
push  msel1
pop  es
mov  al,es:[bp+si]
pop  es
...
```

where the same buffer now appears starting at es:0000.

Figures 3.1 through 3.3 represent simple examples of how to create and use a memory segment under OS/2. In this case the temporary screen buffer is created, accessed, and destroyed. Clearly, this technique can be used to manage memory within a given task. In the next section we consider the creation, access, and destruction of a segment shared between two tasks.

3.2.2 Creating and Accessing a Shared Segment

The program discussed in this section involves both a memory management function and multitasking. Although we do not formally address multitasking until Section 3.3, it is useful to include it in this discussion because at least two processes are needed to demonstrate segment sharing.

The program illustrated in Figure 3.4 opens with the following API call (which creates the shared segment):

```
@DosAllocShrSeg  shared_length,shrname,shrsel
```

Here shared_length has been set at 404 bytes. These will correspond to 400 bytes (4 extra bytes used to contain parameter data) that contain the corner values for boxes (xb, xe, yb, and ye). Further, these corner values will consist of numbers between 0 and 200; hence the full height of the display screen will be used but only the first 200 columns.

The parameter shrname is a symbolic name to be associated with the shared memory segment to be allocated. The name string must include the prefix

```
\SHAREMEM\...
```

We choose

```
'\SHAREMEM\SDAT.DAT',o
```

which is a zero-terminated string. Finally, shrsel is the shared segment selector. The name string for the shared segment must be common to all modules that use this segment and hence provides the link for ensuring that the same protected segment is accessed.

In Figure 3.4, once the shared segment is allocated, the buffer contained in this segment is set to zero. Next a child process is executed using the statement

```
@DosExecPgm obj_name_buf, lobj_name_buf, async, argptr, envptr,
pid, prgm_nm
```

where obj_name_buf is a buffer containing error pointers, lobj_name_buf is the length of this buffer, async = 1 indicates the two processes execute asynchronously, argptr = 0, envptr = 0, pid contains return codes, and prgm_num is the name of the file to be executed (in this case NOS261.EXE, which is zero terminated). This causes the process in Figure 3.5 to execute, NOS261.ASM.

The process NOS261.ASM gets the shared segment using

```
@DosGetShrSeg  shrname,  shrsel
```

where the shared name, shrname, is the same but a new selector is assigned. Next, a sequence of random numbers are loaded into the shared segment locations based on the following formula [3]:

$$x_{n+1} = (2053\, x_n + 13{,}849) \bmod 2^{16} \tag{3.1}$$

```
        PAGE 55,132
TITLE OS2512 - This is the calling OS/2 program (NOS2512.ASM)
;
;       DESCRIPTION: This program plots boxes in protected
;       mode and hesitates using a keyboard delay. Graphics
;       mode 05H is used to display the boxes. It is the same
;       as OS24 except it uses external modules.  This routine
;       employs multitasking to access the input box parameters,
;       which are generated randomly (100 boxes in square 200 x 200).
;       The program prints graphics under program control.
;
.8087
;
EXTRN   boxx:FAR,cls:FAR,clsCGA:FAR,scr_ld:FAR,prtscr:FAR
;
PUBLIC  viohdl,tr,lc,br,rc,no_line,blank,CGAm,lmodeE,typeCGA,colCGA
PUBLIC  txtcCGA,txtrCGA,hrCGA,vrCGA,STDm,lmode80,type80,col80
PUBLIC  txtc80,txtr80,hr80,vr80,waitf,dstat,PVBPtr1,bufstl,buflen1,physel1
PUBLIC  MASK1,MASK11,OFFSET1,four,xx,dummy,two,xxx,eighty,row,col
PUBLIC  address,x,y,xb,xe,ye,yb
;
;       ------------------------------------------------------------
;                       Printscreen variables
;       ------------------------------------------------------------
;
PUBLIC  in_buffer,in_buffer1,in_buffer2,in_buffer3,in_buffer4
PUBLIC  bytesin,bytesin1,bytesin2,bytesin3,bytesout
PUBLIC  dev_name,dev_hand,dev_act,dev_size,dev_attr,dev_flag
PUBLIC  dev_mode,dev_rsv,MM,coll,N
PUBLIC  s,eight,eighty,four,shift1,scr_buffer
PUBLIC  sixforty,N4,ddd,w,b1
;
;       -------------------------------------------------------------
;
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     0                       ;Terminates current thread
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
```

Figure 3.4 The program nos2512.asm, which creates a shared segment, creates a child process, and prints the screen.

```
hrCGA   dw      320                         ;horizontal resolution
vrCGA   dw      200                         ;vertical resolution
;
STDm    label   FAR                         ;Video mode structure-80x25
lmode80 dw      12                          ;Structure length
type80  db      00000001B                   ;Mode identifier-Mode 3+
col80   db      4                           ;Color option
txtc80  dw      80                          ;text characters/line
txtr80  dw      25                          ;text lines
hr80    dw      720                         ;horizontal resolution
vr80    dw      400                         ;vertical resolution
;
kbd_buf db      80                          ;Keyboard buffer
lkbd_buf dw     $-kbd_buf                   ;Length keyboard buffer
iowait  dw      0                           ;Wait for CR
kbdhdl  equ     0                           ;Keyboard handle
;
waitf   equ     1                           ;Screen waiting status
dstat   db      ?                           ;Returned status
;
PVBPtr1 label   FAR                         ;Video buffer structure
bufst1  dd      0B8000H                     ;Start physical address
buflen1 dd      8000H                       ;Buffer length
physel1 dw      0                           ;OS/2 screen buffer selector
;
MASK1   db      01H                         ;PEL byte mask
MASK11  dw      0001H                       ;Odd/even row mask
OFFSET1 dw      2000H                       ;Odd row buffer offset
four    dw      4
xx      dw      ?                           ;PEL modulo parameter
dummy   dw      ?                           ;80287 dummy "pop"
two     db      2
xxx     db      ?                           ;Output value
eighty  dw      80
zero    dw      0
one     dw      1
row     dw      ?                           ;row
col     dw      ?                           ;column
address dw      ?                           ;Address screen dot
;
x       dw      ?                           ;Box col parameter
y       dw      ?                           ;Box row parameter
xb      dw      ?                           ;Start column
xe      dw      ?                           ;End column
yb      dw      ?                           ;Start row
ye      dw      ?                           ;End row
;
obj_name_buf    dd      10 dup(0)           ;object name buffer
lobj_name_buf   dw      $-obj_name_buf      ;buffer length
async   dw      1                           ;Flag indicates async
argptr  dw      0                           ;0 for argument ptr
envptr  dw      0                           ;0 for environment ptr
pid     dw      ?                           ;Process ID result code
        dw      ?
prgm_nm db      'NOS261.EXE',0              ;program name & parameter
;
shared_length   dw      404                 ;Length shared buffer
shrname db      '\SHAREMEM\SDAT.DAT',0
shrsel  dw      ?                           ;selector
;
ESDI    db      400 dup(?)                  ;Buffer for shared data
count   dw      ?                           ;Buffer size in bytes
;
;       ------------------------------------------------------------
;                       Printscreen Variables
;       ------------------------------------------------------------
```

Figure 3.4 *(Continued)*

```
eight   dw      8
in_buffer       db      320 dup(0)      ;print buffer
in_buffer1      db      1Bh,4BH,64D,01H ;printer setup
in_buffer2      db      0DH,0AH         ;LF/CR
in_buffer3      db      1BH,41H,08H     ;
in_buffer4      db      1BH,32H
bytesin         dw      320             ;print buffer count
bytesin1        dw      4               ;count bytes in_buffer1
bytesin2        dw      2               ;count bytes in_buffer2
bytesin3        dw      3               ;count bytes in_buffer3
bytesout        dw      0
;
dev_name        db      'LPT1',0        ;name of printer device
dev_hand        dw      0               ;device handle
dev_act         dw      0                       ;
dev_size        dd      0               ;
dev_attr        dw      0
dev_flag        dw      00000001b       ;open file
dev_mode        dw      0000000011000001b       ;hdl private,deny none,w/o
dev_rsv dd      0
;
N4      dw      ?
MM      db      40H,10H,04H,01H         ;pel mask
w       db      128,64,32,16,8,4,2,1    ;pin weights
col1    db      320 dup(?)              ;columns
b1      db      4 dup(?)
N       dw      ?
shift1  db      6,4,2,0
s       db      4 dup(?)
ddd     dw      ?
sixforty        dw      640
scr_buffer      db      16384 dup(0)    ;temporary buffer
;
;       ----------------------------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        @DosAllocShrSeg shared_length,shrname,shrsel
        cmp ax,0                        ;Check on successful creation
        jz NO_ERROR1                    ;Successful
        jmp ERROR1                      ;Error
                                        ;
NO_ERROR1:
;
        push shrsel                     ;Save selector
        pop es                          ;Selector in extra segment
                                        ;
        mov ax,one
        mov es:[2],ax                   ;Flag indicating creation
                                        ;
        mov ax,shared_length            ;Length shared buffer
        mov es:[0],ax                   ;Length parameter passed-multitask
                                        ;
        mov di,four                     ;Data record offset in buffer
        mov cx,shared_length            ;Data buffer length + 4
        sub cx,four                     ;Data buffer length
        mov ax,zero                     ;Clear character
lloop:
        mov es:[di],al                  ;Clear buffer
        inc di                          ;Next buffer point
        loop lloop
;
```

Figure 3.4 *(Continued)*

```
        @DosExecPgm obj_name_buf,lobj_name_buf,async,argptr,envptr,pid,prgm_nm
        cmp ax,0                        ;Check error condition
        jz NO_ERROR2                    ;Jump no error
        jmp ERROR2                      ;Jump error
;
NO_ERROR2:
;
        mov ax,zero                     ;Indicates buffer write complete
;
NO_ERROR22:
;
        cmp es:[2],ax                   ;Check buffer write
        jz MEM_CL                       ;Jump if buffer write complete
        jmp NO_ERROR22                  ;Otherwise wait
;
MEM_CL:
;
        mov si,zero                     ;Offset in intermediate buffer
        mov di,four                     ;Offset in shared buffer
        mov cx,shared_length            ;Length data buffer + 4
        sub cx,four                     ;Length data buffer
        mov count,cx                    ;Data buffer size in bytes
loop22:
        mov al,es:[di]                  ;Obtain shared buffer value
        mov ESDI[si],al                 ;Load shared memory buffer
        inc di                          ;Increment shared buffer ptr
        inc si                          ;Increment intermediate buffer ptr
        loop loop22
;
        call cls                        ;Clear screen
        @VioSetMode CGAm,viohdl         ;Set CGA graphics mode
        call clsCGA                     ;Clear CGA screen
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer selector
        push physel1                    ;Save selector
        pop es                          ;Load selector into extra segment
;
        mov di,0                        ;Intermediate buffer offset
        mov dx,0                        ;Clear upper dividand
        mov ax,count                    ;Data buffer byte count
        div four                        ;Reduce to sets of four
        mov cx,ax                       ;Loop count
loop2:
        push cx                         ;Save loop count
        mov al,ESDI[di]                 ;Obtain 1st buffer value-set
        mov ah,ESDI[di+1]               ;Obtain 2nd buffer value-set
        cmp ah,al                       ;Check values equal
        jne EELSE1
           mov al,170                   ;Arbitrarily set 1st equal value
           mov ah,180                   ;Arbitrarily set 2nd equal value
           call xload                   ;Load xb and xe
           jmp IIF1
EELSE1:
        cmp ah,al                       ;Check ah g.t. al
        jle ELSE1
           call xload                   ;Load xb and xe
           jmp IIF1
ELSE1:
           mov bl,al                    ;Swap ah and al
           mov al,ah
           mov ah,bl
           call xload                   ;Load xb and xe
IIF1:
        mov al,ESDI[di+2]               ;Obtain 3rd buffer value-set
        mov ah,ESDI[di+3]               ;Obtain 4th buffer value-set
        cmp ah,al                       ;Check values equal
        jne EELSE2
```

Figure 3.4 *(Continued)*

```
           mov al,170                    ;Arbitrarily set 1st equal value
           mov ah,180                    ;Arbitrarily set 2nd equal value
           call yload                    ;Load yb and ye
           jmp IIF2
EELSE2:
        cmp ah,al                        ;Check ah g.t. al
        jle ELSE2
           call yload                    ;Load yb and ye
           jmp IIF2
ELSE2:
           mov bl,al                     ;Swap ah and al
           mov al,ah
           mov ah,bl
           call yload                    ;Load yb and ye
IIF2:
        push di                          ;Save buffer offset
        call boxx                        ;Draw box
        pop di                           ;Recall buffer offset
        add di,four                      ;Increment data ptr 4 bytes
;
        pop cx                           ;Recall loop count
        loop loop2
;
        call scr_ld                      ;load screen print buffers
;
        @VioScrUnLock   viohdl           ;Unlock screen context
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
        @VioSetMode STDm,viohdl          ;80 x 25 alpha mode
;
        @DosKillProcess 1,pid            ;Terminate child process
ERROR2:
        @DosFreeSeg shrsel               ;Free shared memory
ERROR1:
        call prtscr
        @DosClose dev_hand
        @DosExit action,result           ;Terminate process
;
OS21    ENDP
;
xload   PROC    NEAR
        mov bh,0                         ;Clear upper register half
        mov bl,al                        ;al = start
        mov xb,bx                        ;Load xb less than 199
        mov bh,0                         ;Clear upper register half
        mov bl,ah                        ;ah = end
        mov xe,bx                        ;Load xe less than 199
        ret
xload   ENDP
;
yload   PROC    NEAR
        mov bh,0                         ;Clear upper register half
        mov bl,al                        ;al = start
        mov yb,bx                        ;Load yb less than 199
        mov bh,0                         ;Clear upper register half
        mov bl,ah                        ;ah = end
        mov ye,bx                        ;Load ye less than 199
        ret
yload   ENDP
;
CSEG    ENDS
        END     OS21
```

Figure 3.4 *(Concluded)*

```
PAGE 55,132
TITLE OS261 - Generates multitask r.n. (OS261.ASM)
;
;       DESCRIPTION: This process generates the multitasked
;       random numbers.  It is called by the plot process.
;
.8087
;
IF1
        include sysmac.inc
ENDIF
;
        .sall                               ;Suppresses macro lists
dgroup  GROUP   data1
;
STACK1  SEGMENT PARA STACK 'STACK'
        db      256 dup('STACK1  ')
STACK1  ENDS
;
DATA1   SEGMENT PARA PUBLIC 'DATA'
;
rnd1    dw      ?                           ;seed value
;
one     dw      1
action  equ     0
result  dw      0
ssize   dw      ?                           ;Buffer size + 4
shrsel  dw      ?                           ;Selector
shrname db      '\SHAREMEM\SDAT.DAT',0      ;Shared memory name
zero    dw      0
;
DATA1   ENDS
;
CSEG1   SEGMENT PARA PUBLIC 'CODE'
        assume  cs:cseg1,ds:dgroup
OS261   PROC    FAR
;
        mov ax,one                          ;Load initial seed value
        mov rnd1,ax
        @DosGetShrSeg shrname,shrsel        ;Get shared segment
        push shrsel                         ;Save selector
        pop es                              ;Selector to extra segment
;
        mov ax,es:[0]                       ;Establish shared buffer size
        mov ssize,ax                        ;Define buffer size + 4
;
        mov di,4                            ;Pointer to data buffer
        mov cx,ssize                        ;Loop byte count + 4
        sub cx,4                            ;Loop byte count
loop1:
        mov al,0                            ;Clear buffer
        mov es:[di],al                      ;Buffer write
        inc di                              ;Increment offset
        loop loop1
                                            ;
        mov di,4                            ;Pointer to data buffer
        mov cx,ssize                        ;Loop byte count + 4
        sub cx,4                            ;Loop byte count
loop2:
        call ldmem                          ;Generate random value
        mov es:[di],al                      ;Load shared buffer (byte)
        inc di                              ;Increment byte offset
        loop loop2
```

Figure 3.5 The child process nos261.asm, used to generate random numbers in a Protected Mode multitasked environment.

```
                                      ;
        mov ax,zero                   ;Flag indicating write complete
        mov es:[2],ax                 ;Flag loaded
                                      ;
;
        @DosFreeSeg shrsel
        @DosExit action,result
                                      ;
OS261   ENDP
;
ldmem   PROC    NEAR
;                                     ;Generate r.n.
        mov dx,0                      ;Load upper multiplicand zero
        mov ax,rndl                   ;Load previous r.n.
        mov bx,2053                   ;Multiplier
        mul bx
        mov bx,13849                  ;Load additative constant
        clc
        add ax,bx                     ;Add low order result
        adc dx,0                      ;Add carry if needed
        mov bx,0FFFFH                 ;Load 2(16) - 1
        div bx                        ;Calculate modulo
        mov ax,dx                     ;Move remainder into ax
        mov rndl,ax                   ;Save r.n.
        mov bx,350                    ;Scale r.n. to less than 200
        mov dx,0                      ;Clear upper dividand
        div bx                        ;Scale
        mov ah,0                      ;Save al
        ret
ldmem   ENDP
;
CSEG1   ENDS
        END     OS261
```

Figure 3.5 (*Concluded*)

Here x_{n+1} is the (n + 1)th number and x_n the *n*th number(s). The procedure ldmem contains the code that calculates this random sequence. Once the sequence is calculated, the size of the buffer is checked. This size was loaded as a word at es:[0] in the creating routine. In NOS261.ASM the value is used to set the number of random values to be calculated (here this is 400).

Next, NOS261.ASM frees the shared segment using

```
@DosFreeSeg shrsel
```

and exits back to OS/2. Prior to freeing the segment, however, NOS261.ASM loads a zero at es:[2] to indicate that the buffer write is complete. (This value was previously set to 1.) The main calling program, NOS2512.ASM, sits in a loop checking es:[2] for a value of zero. (This is a somewhat wasteful operation and could be used asynchronously to accomplish other tasks if needed.) Once the random values have been completely loaded and NOS2512.ASM becomes aware of this, it terminates the loop and reloads these shared values into a buffer, ESDI. The processing then continues in the usual fashion to clear the screen, set the CGA mode, and capture the physical screen buffer. Using two routines, xload and yload, the box corners are loaded, ensuring that (xb, yb) are always less than (xe, ye), respectively. The routine boxx is called and the random boxes generated on the display. Figure 3.6 illustrates the variant of boxx used for this call. Note that it includes a check on the corners to ensure that xb < xe and yb < ye.

```
                PAGE 55,132
TITLE OS252 - Supplemental routines for box plotting (OS252.ASM)
;
;       DESCRIPTION: These routines set up box plots in CGA
;       mode and hesitate using a keyboard delay. Graphics
;       mode 05H is used to display the box. This set of routines
;       is called by box plotting main routine.
;
.8087
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;Suppresses macro lists
;
EXTRN   viohdl:WORD,tr:WORD,lc:WORD,br:WORD,rc:WORD
EXTRN   no_line:WORD,blank:WORD,CGAm:FAR,lmodeE:WORD,typeCGA:BYTE
EXTRN   colCGA:BYTE,txtcCGA:WORD,txtrCGA:WORD,hrCGA:WORD,vrCGA:WORD
EXTRN   STDm:FAR,lmode80:WORD,type80:BYTE,col80:BYTE,txtc80:WORD,txtr80:WORD
EXTRN   hr80:WORD,vr80:WORD
EXTRN   waitf:WORD,dstat:BYTE,PVBPtr1:FAR,bufst1:DWORD
EXTRN   buflen1:DWORD,physel1:WORD,MASK1:BYTE,MASK11:WORD,OFFSET1:WORD
EXTRN   four:WORD,xx:WORD,dummy:WORD,two:BYTE,xxx:BYTE,eighty:WORD
EXTRN   row:WORD,col:WORD,address:WORD,x:WORD,y:WORD,xb:WORD,xe:WORD
EXTRN   yb:WORD,ye:WORD
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
PUBLIC  cls,boxx,clsCGA
        assume cs:cseg
boxx    PROC    FAR
;
;       xb = x-begin,xe = x-end,yb = y-begin,ye = y-end
;
        mov ax,xb                       ;Check xb l.t. xe
        cmp ax,xe
        jl ELSE10
           xchg ax,xe                   ;Swap xb and xe
           mov xb,ax
ELSE10:
        mov ax,yb                       ;Check yb l.t. ye
        cmp ax,ye
        jl ELSE11
           xchg ax,ye                   ;Swap yb and ye
           mov yb,ax
ELSE11:
        mov ax,yb                       ;Top box line
        mov y,ax
        call lineh                      ;Draw top horizontal line
        mov ax,ye                       ;Bottom box line
        mov y,ax
        call lineh                      ;Draw bottom horizontal line
        mov ax,xb                       ;Left box line
        mov x,ax
        call linev                      ;Draw left vertical line
        mov ax,xe                       ;Right box line
        mov x,ax
        call linev                      ;Draw right vertical line
;
        ret
boxx    ENDP
;
cls     PROC    FAR
;
        @VioScrollUp tr,lc,br,rc,no_line,blank,viohdl
```

Figure 3.6 Supplemental routines needed by the random box generating routine, nos2512.asm.

```
        ret
;
cls     ENDP
;
clsCGA  PROC    FAR
;
        @VioScrLock waitf,dstat,viohdl   ;Lock screen context
        @VioGetPhysBuf PVBPtr1,viohdl    ;Get physical buffer
        push physel1                     ;Screen selector
        pop es                           ;Load extra segment
;
        mov bp,0                         ;Start offset zero
        mov al,0                         ;Zero attribute-clear
DO1:
        mov es:[bp],al                   ;Clear byte
        inc bp
        cmp bp,1F3FH                     ;Check end 1st buffer
        jle DO1
;
        mov bp,2000H                     ;Offset 2nd buffer-odd
        mov al,0                         ;Zero attribute-clear
DO2:
        mov es:[bp],al                   ;Clear byte
        inc bp
        cmp bp,3F3FH                     ;Check end 2nd buffer
        jle DO2
;
        @VioScrUnLock viohdl             ;Unlock screen context
;
        ret
clsCGA  ENDP
;
wdot    PROC    NEAR
;
;       (col,row) = (x,y)
;
        fild four                        ;Load stack with 4
        fild col                         ;ST = col, ST(1) = 4
        fprem                            ;Modulo
        fistp xx                         ;Store remainder in xx
        fistp dummy                      ;Pop stack
        mov al,3
        mov bl,byte ptr xx
        sub al,bl                        ;(3 - col % 4)
        mov ah,0                         ;Clear upper multiplicand
        mul two
        mov cl,al                        ;Shift value for PEL
        mov al,MASK1                     ;PEL color mask
        shl al,cl                        ;Shift to correct PEL
        mov xxx,al                       ;Store buffer value
                                         ;
        mov ax,row                       ;Begin address calculation
        shr ax,1                         ;Divide row by 2
        mov dx,0                         ;Clear upper multiplicand
        mul eighty
        mov bx,col                       ;Convert column value to bytes
        shr bx,1
        shr bx,1
        add ax,bx                        ;offset in ax
        mov address,ax                   ;Save offset base
        mov ax,row                       ;Check even/odd row
        and ax,MASK11                    ;Look for bit 0 set
        cmp ax,0
        jle ELSE1
           mov ax,address
           add ax,OFFSET1                ;add odd buffer offset
           jmp IF11
```

Figure 3.6 (*Continued*)

```
ELSE1:
        mov ax,address
IF11:
        mov bp,ax                       ;screen buffer address
        mov al,xxx                      ;Attribute value for dot
;
        or es:[bp],al                   ;Write dot
;
        ret
wdot    ENDP
;
lineh   PROC    NEAR
;
;       y = row position, xb = begin, xe = end
;
        mov ax,y                        ;Establish row for wdot
        mov row,ax
                                        ;
        mov ax,xb                       ;Establish start column
DO10:
        mov col,ax
        push ax                         ;Save column value
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall column
        inc ax                          ;Increment column
        cmp ax,xe                       ;Check end horizontal line
        jle DO10
                                        ;
        ret
lineh   ENDP
;
linev   PROC    NEAR
;
;       x = col position, yb = begin, ye = end
;
        mov ax,x                        ;Establish column for wdot
        mov col,ax
                                        ;
        mov ax,yb                       ;Establish start row
DO20:
        mov row,ax
        push ax                         ;Save row value
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall row
        inc ax                          ;Increment row
        cmp ax,ye                       ;Check end vertical line
        jle DO20
                                        ;
        ret
linev   ENDP
;
CSEG    ENDS
        END
```

Figure 3.6 *(Concluded)*

Once the boxes are plotted the keyboard hesitate takes place, followed by a call to prtscr that prints the screen content. A similar version of this program appears in reference 4, without the screen print logic. Figure 3.7 illustrates the plotted boxes.

3.2.3 Changing Segment Size

Among the API memory management services are functions for changing the size of an allocated segment. It is a prerequisite, however, that the segment be allocated during the existing session. Figure 3.8 presents a program that allocates and then

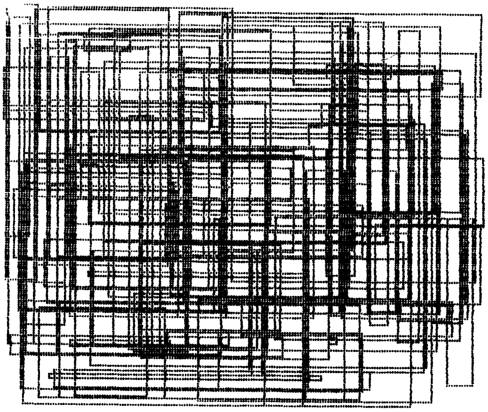


Figure 3.7 Screen print of the 100 random boxes output by nos2512.asm.

modifies the size of a segment. This program is somewhat artificial in that it serves no useful purpose other than to demonstrate this memory management technique.

The program opens with the call

```
@DosAllocSeg msize, msell, mflag
```

Here msize is the desired size of the segment (in this case 4000H), in msell the returned selector, and mflag determines the type of segment access. Specifically, bits 0 and 1, when set, permit sharing of the segment using DosGiveSeg and DosGetSeg, respectively. Bit 3 allows the segment to be discarded in low-memory situations. If the segments are shared, they can only be increased in size. We have set all three of these bits to 0.

The next block of code pushes the allocated segment selector on the stack and pops it into es. Hence, es now points to the created segment. This segment is loaded with 2048 copies of the string "MEMORY ", where two blank spaces have been added to the end of the string. Once completed, the call

```
@DosReallocSeg msize1, msell
```

is made and the segment size reduced to one byte beyond a paragraph boundary. At this point some mechanism must exist to check the segment definition. Subsequent code writes a 1 into the first location of the segment, followed by a write to the eighteenth position. The latter position should yield a protection violation.

Figure 3.9 illustrates CodeView results for the program following creation of the memory segment. Note that beginning at address es:0x0000 (here 0x0000 specifies a hexadecimal address, 0000, in C notation), the string "MEMORY " is loaded. A check at 0x4000 shows that the preceding 4000H locations are filled with this string also (Figure 3.9b).

```
                PAGE 55,132
TITLE MEMSEG -- Reallocate memory segment (memseg.asm)
;
;       DESCRIPTION: This simple routine creates and reallocates
;       a memory segment.  The final memory instruction is
;       designed to create a protection violation.  The program
;       should be run with CodeView.
;
IF1
        include sysmac.inc
ENDIF
;
        .sall
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup ('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
msize   dw      16385                   ;buffer size
msel1   dw      ?                       ;selector
mflag   dw      0000000000000000B       ;not sharable
blk_ct  dw      16384                   ;block count
mem_wd  db      'M','E','M','O','R','Y',' ',' ' ;string
msize1  dw      17                      ;new buffer size
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        @DosAllocSeg    msize,msel1,mflag       ;allocate segment
;
        push msel1
        pop es
;
        mov dx,blk_ct                   ;block counter limit
        mov di,0                        ;buffer block count
        lea bp,mem_wd                   ;string address
LOOP1:
        mov cx,8                        ;count limit for string
        mov si,0                        ;index for string/buffer
LOOP2:
        mov al,ds:[bp+si]               ;load from string
        mov es:[di],al                  ;load buffer
        inc si                          ;increment string
        inc di                          ;increment block byte
        loop LOOP2
;
        cmp di,dx                       ;check block limit
        jl LOOP1
;
        @DosReallocSeg  msize1,msel1    ;reallocate segment
;
        push msel1                      ;preserve selector
        pop es                          ;create extra segment
        mov bp,0                        ;segment index
        mov al,1                        ;load dummy value
        mov es:[bp],al                  ;single load in buffer
        mov es:[bp+17],al               ;PROTECTION VIOLATION
;
```

Figure 3.8 Simple routine for creating and reallocating memory.

```
@DosExit 1,0
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.8 (*Concluded*)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!        MEMSEG.EXE

003F:002F 268805          MOV       Byte Ptr ES:[DI],AL
003F:0032 46              INC       SI
003F:0033 47              INC       DI
003F:0034 E2F6            LOOP      002C
003F:0036 3BFA            CMP       DI,DX
003F:0038 7CEC            JL        0026
003F:003A A11000          MOV       AX,Word Ptr [0010]

IBM CodeView (R) Version 1.00
Copyright (C) IBM Corporation 1987
Copyright (C) Microsoft (R) Corporation 1986, 1987
>d es:0x0000
006F:0000  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0010  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0020  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0030  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0040  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0050  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>
```

(a)

Figure 3.9 Facsimile CodeView output, illustrating (a) low end of initialized memory segment and (b) high end of initialized memory segment. (Courtesy of the Microsoft Corporation.)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!          MEMSEG.EXE

003F:002C 3E8A02          MOV       AL,Byte Ptr DS:[BP+SI]
003F:002F 268805          MOV       Byte Ptr ES:[DI],AL
003F:0032 46              INC       SI
003F:0033 47              INC       DI
003F:0034 E2F6            LOOP      002C
003F:0036 3BFA            CMP       DI,DX
003F:0038 7CEC            JL        0026
003F:003A A11000          MOV       AX,Word Ptr [0010]

006F:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>d es:0x3ff0
006F:3FF0  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:4000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4010  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4020  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4030  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4040  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4050  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4060  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
>
```

(b)

Figure 3.9 (*Concluded*)

Figure 3.10a corresponds to execution up to 003F:0055, in which the resized segment now has its first location loaded with 1. Figure 3.10b corresponds to execution of 003F:0055 and the subsequent violation is noted.

Figure 3.11 illustrates CodeView output for the case when a segment violation might occur except that the segment is defined as sharable with bits 0 and 1 of mflag set. In this case the attempt to reallocate the size of the segment downward to 17 bytes fails and the initial segment size remains implemented. The instruction 003F:0055 executes as the segment data indicate.

3.2.4 Creating and Accessing Huge Segments

The OS/2 kernel (level 0) allocates and maintains segment descriptors. This provides a mapping of the virtual address space onto the physical memory space. OS/2 does, in fact, allow the user the capability to request and use more memory than exists in his or her system. This is accomplished using extended file management techniques. A very key boundary under DOS and within the confines of the Intel 8086 family of architectures is the 64K boundary or segment size. OS/2 has API services that allow the user to extend a huge memory block because the underlying segments are still in place. The system simply provides additional selectors, as needed, to access the subsequent high-memory spaces.

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!        MEMSEG.EXE

003F:0042 9A0000B345     CALL     45B3:0000
003F:0047 FF360200       PUSH     Word Ptr [0002]
003F:004B 07             POP      ES
003F:004C BD0000         MOV      BP,0000
003F:004F B001           MOV      AL,01
003F:0051 26884600       MOV      Byte Ptr ES:[BP+00],AL
003F:0055 26884611       MOV      Byte Ptr ES:[BP+11],AL
003F:0059 B80100         MOV      AX,0001

Copyright (C) IBM Corporation 1987
Copyright (C) Microsoft (R) Corporation 1986, 1987
>d es:0x0000
006F:0000  01 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  .EMORY  MEMORY
006F:0010  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0020  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0030  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0040  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0050  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0060  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0070  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
>
```

(a)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!        MEMSEG.EXE

003F:0042 9A0000B345     CALL     45B3:0000
003F:0047 FF360200       PUSH     Word Ptr [0002]
003F:004B 07             POP      ES
003F:004C BD0000         MOV      BP,0000
003F:004F B001           MOV      AL,01
003F:0051 26884600       MOV      Byte Ptr ES:[BP+00],AL
003F:0055 26884611       MOV      Byte Ptr ES:[BP+11],AL
003F:0059 B80100         MOV      AX,0001

Copyright (C) Microsoft (R) Corporation 1986, 1987
>d es:0x0000
006F:0000  01 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  .EMORY  MEMORY
006F:0010  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0020  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0030  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0040  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0050  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0060  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:0070  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
>Segmentation violation
```

(b)

Figure 3.10 Facilimile CodeView output, illustrating (a) reallocated memory and (b) reallocated memory with protection violation. (Courtesy of the Microsoft Corporation.)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!          MEMSEG.EXE

003F:0042 9A0000B345      CALL      45B3:0000
003F:0047 FF360200        PUSH      Word Ptr [0002]
003F:004B 07              POP       ES
003F:004C BD0000          MOV       BP,0000
003F:004F B001            MOV       AL,01
003F:0051 26884600        MOV       Byte Ptr ES:[BP+00],AL
003F:0055 26884611        MOV       Byte Ptr ES:[BP+11],AL
003F:0059 B80100          MOV       AX,0001

07E7:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>d es:0x0000
07E7:0000  01 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  .EMORY  MEMORY
07E7:0010  4D 01 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  M.MORY  MEMORY
07E7:0020  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0030  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0040  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0050  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07E7:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>
```

Figure 3.11 Facsimile CodeView output, illustrating reallocated memory for the conditions of Figure 3.10b with sharable memory. *Note:* There is no protection violation. (Courtesy of the Microsoft Corporation.)

Figure 3.12 illustrates a program hugeseg.asm, which is used for allocating such a huge segment. In this case 81,920 bytes are allocated. The initial call is

```
@DosAllocHuge mnumseg, msize, msell, msegmax, mflag
```

where

`mnumseg`	number of 64K whole blocks in segment
`msize`	number of bytes in last non-whole block
`msell`	selector
`msegmax`	maximum number of 64K whole segments occupied (set = 0 means the segment can only be decreased)
`mflag`	bit 0=1 (shareable through DosGiveSeg) bit 1=1 (shareable through DosGetSeg) bit 2=1 (discardable in low-memory cases)

In the program we use one 64K whole segment and a partial memory block of 16,384 bytes. The initial selector value returned by @DosAllocHuge points to the first block in the huge segment. Subsequent blocks must be accessed using

```
        PAGE 55,132
TITLE HUGESEG -- Allocate a huge segment (hugeseg.asm)
;
;       DESCRIPTION: This program allocates a huge segment:
;       2 (65536) and 1 (16384) byte 64k blocks.  It is checked
;       using CodeView.
;
IF1
        include sysmac.inc
ENDIF
;
        .sall                           ;suppresses listing
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup ('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
mnumseg dw      1                       ;number 64k whole blocks
msize   dw      16384                   ;bytes in last block (partial)
msel1   dw      ?                       ;selector
msegmax dw      0                       ;maximum realloc 64k blocks
mflag   dw      0000000000000111B       ;segment characteristics
blk_ct  dw      8192,2048               ;bytes in each block
shift_ct        dw      ?               ;shift count
mem_wd  db      'M','E','M','O','R','Y',' ',' '
mseg_ct dw      1                       ;block counter (0,1)
two     dw      2
;
DATA ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup
OS21    PROC    FAR
;
        @DosAllocHuge   mnumseg,msize,msel1,msegmax,mflag
;
        mov si,0                        ;block index
        push si                         ;preserve index
LOOP3:
        pop si                          ;recall block index
        mov ax,msel1                    ;load selector
        cmp si,1                        ;check if 1st block
        jl ELSE1                        ;jump if 1st block
                                        ;
           @DosGetHugeShift  shift_ct   ;get shift count
           mov bx,1                     ;bx to be shifted
           mov cl,byte ptr shift_ct     ;load shift as byte
           shl bx,cl                    ;amount shifted
           mov ax,msel1                 ;reload selector
           add ax,bx                    ;create new selector
ELSE1:
        mov msel1,ax                    ;reload selector
;
        push si
        mov ax,si
        mul two
        mov si,ax
        mov dx,blk_ct[si]               ;block byte sount
        pop si
;
```

Figure 3.12 The program hugeseg.asm, used for allocating a huge segment (81,920 bytes).

```
        mov di,0                        ;block internal index
        lea bp,mem_wd                   ;address "MEMORY  "
        inc si                          ;increment block count
        push si                         ;preserve block count
        push msell                      ;selector
        pop es                          ;selector in es
        mov cx,dx                       ;load block string count
LOOP1:
        mov si,0                        ;string index
LOOP2:
        mov al,ds:[bp+si]               ;load string member
        mov es:[di],al                  ;insert in huge segment
        inc si                          ;increase string index
        inc di                          ;increase huge segment index
        cmp si,7                        ;check string count
        jle LOOP2
;
        loop LOOP1
;
        pop si                          ;recall block count
        cmp si,mseg_ct                  ;last block?
        push si                         ;preserve block count
        jle LOOP3
;
        @DosFreeSeg msell
;
        @DosExit        1,0
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.12 (*Concluded*)

@DosGetHugeShift. This provides a shift count that can be used to calculate an offset. Note that the call

```
@DosGetHugeShift  shift_ct
```

returns a shift count in shift_ct. The selector offset increment is obtained by shifting the value 1 to the left by the amount specified as the shift count, shift_ct. This is then added to the selector value to get the new selector. For example, suppose that the selector is 6F7H. If the shift count returned is 4, an increment of 16 must be added to 6F7H to get the new selector: 707H. If several blocks have been allocated, the selector for each must be obtained by adding the increment to each successive selector to obtain the following value.

In Figure 3.12 a check is made on whether the first block is being processed (si less than 1) and the shift count processing implemented as needed. The word "MEMORY " is then written into the memory block. Finally, the block is released using @DosFreeSeg. Figure 3.13 illustrates the CodeView memory dump starting at 07F7:FFF0, the end of the segment. Since the listing wraps around at 07F7:FFFF, it is clear that the 64K block is filled with "MEMORY ". Figure 3.14a illustrates the beginning of the last partial segment and Figure 3.14b the end of this partial segment (16,384 bytes long). The partial segment is, of course, also loaded with "MEMORY ", indicating that the allocation and use of the huge segment (81,920 bytes) was successful.

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!        HUGESEG.EXE

003F:0068 BE0000          MOV       SI,0000
003F:006B 3E8A02          MOV       AL,Byte Ptr DS:[BP+SI]
003F:006E 268805          MOV       Byte Ptr ES:[DI],AL
003F:0071 46              INC       SI
003F:0072 47              INC       DI
003F:0073 83FE07          CMP       SI,+07
003F:0076 7EF3            JLE       006B
003F:0078 E2EE            LOOP      0068
003F:007A 5E              POP       SI

07F7:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>d es:0xfff0
07F7:FFF0  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0000  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0010  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0020  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0030  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0040  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0050  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
07F7:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>
```

Figure 3.13 Facsimile CodeView output, illustrating loading through first 64K block limits (07F7:0000 through 07F7:FFFF) of huge segment. (Courtesy of the Microsoft Corporation.)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!        HUGESEG.EXE

003F:006E 268805          MOV       Byte Ptr ES:[DI],AL
003F:0071 46              INC       SI
003F:0072 47              INC       DI
003F:0073 83FE07          CMP       SI,+07
003F:0076 7EF3            JLE       006B
003F:0078 E2EE            LOOP      0068
003F:007A 5E              POP       SI
003F:007B 3B361800        CMP       SI,Word Ptr [0018]

Copyright (C) IBM Corporation 1987
Copyright (C) Microsoft (R) Corporation 1986, 1987
>d es:0x0000
0807:0000  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0010  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0020  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0030  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0040  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0050  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>
```

Figure 3.14a Facsimile CodeView output, illustrating (a) loading of second block start (0807:0000) and (b) loading through second block end (0807:3FFF) of huge segment. (Courtesy of the Microsoft Corporation.)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!      HUGESEG.EXE

003F:007B 3B361800       CMP      SI,Word Ptr [0018]
003F:007F 56             PUSH     SI
003F:0080 7EA0           JLE      0022
003F:0082 A10400         MOV      AX,Word Ptr [0004]
003F:0085 50             PUSH     AX
003F:0086 9A0000BB45     CALL     45BB:0000
003F:008B B80100         MOV      AX,0001
003F:008E 50             PUSH     AX
003F:008F B80000         MOV      AX,0000

0807:0070  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>d es:0x3ff0
0807:3FF0  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
0807:4000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4010  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4020  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4030  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4040  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4050  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0807:4060  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
>
```

Figure 3.14b *(Concluded)*

OS/2 could have been structured to provide automatic memory management features, but this would have removed some of the flexibility of the operating system. The ability to clean up memory and segregate usage expands the programmer's access to more difficult problem-solving techniques. This is somewhat philosophical and the actual implementation of memory allocation is left up to the individual user. It is, of course, essential for programs that push the limits of the physical system resources.

3.2.5 Suballocating Memory

The final memory management activity considered in this section is suballocation. This is the blocking of memory within an allocated segment and is best used if an application requests and frees small portions of memory at a frequent rate. It has the advantage that an allocation at the physical level is not needed. When a normal allocation occurs an LDT entry must be defined, a descriptor defined, physical memory located, and then the reverse when memory is released. The memory suballocation package (MSP) contains the calls

```
@DosSubAlloc
@DosSubSet
@DosSubFree
```

which allow the allocation and freeing of portions of a segment without incurring the system overhead. The services in the MSP simply keep track of which portions of the memory segment are in use.

Figure 3.15 presents a program that implements a memory suballocation operation. Basically, a segment with 16,385 bytes is allocated using @DosAllocSeg.

```
PAGE 55,132
TITLE SUBALLO -- Reallocate memory segment (suballo.asm)
;
;       DESCRIPTION: This simple routine creates and suballocates
;       a memory segment.  The program should be run with CodeView.
;
IF1
        include sysmac.inc
ENDIF
;
        .sall
dgroup  GROUP   data
;
STACK   SEGMENT PARA STACK 'STACK'
        db      256 dup ('STACK   ')
STACK   ENDS
;
DATA    SEGMENT PARA PUBLIC 'DATA'
;
msize   dw      16385                   ;buffer size
msell   dw      ?                       ;selector
mflag   dw      0000000000000000B       ;not sharable
blk_ct  dw      16384,8192              ;block count
mem_wd  db      'M','E','M','O','R','Y',' ',' ','S','U','B','A','L','L','O','C'
msize1  dw      8192                    ;suballocated size
moffset dw      0                       ;offset to suballocated block
two     dw      2
;
DATA    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        assume cs:cseg,ds:dgroup
OS21    PROC    FAR
;
        @DosAllocSeg    msize,msell,mflag       ;allocate segment
;
        push msell                      ;load allocated selector
        pop es                          ;pop to es register
;
        mov di,0                        ;initialize string offset
        mov si,0                        ;initialize block count variable
        push si                         ;preserve block count
        push di                         ;preserve string offset
LOOP4:
        pop di                          ;recall string offset
        pop si                          ;recall block count
;
        mov dx,blk_ct[si]               ;block counter limit
        lea bp,mem_wd[di]               ;string address
        mov di,moffset                  ;block offset in segment
        push si                         ;preserve block count
        push di                         ;preserve block offset
LOOP1:
        mov cx,8                        ;count limit for string
        mov si,0                        ;index for string/buffer
LOOP2:
        mov al,ds:[bp+si]               ;load from string
        mov es:[di],al                  ;load buffer
        inc si                          ;increment string
        inc di                          ;increment block byte
        loop LOOP2
;
        push di                         ;block offset + block count
        sub di,moffset                  ;block count
```

Figure 3.15 The program suballo.asm, which suballocates a 16,384-byte segment into an 8192-byte block.

```
        cmp di,dx                       ;check block limit
        pop di                          ;block offset + block count
        jl LOOP1
;
        mov ax,1                        ;set suballocation flag
        mov mflag,ax                    ;load
;
        @DosSubSet  msel1,mflag,msize
;
        @DosSubAlloc msel1,moffset,msize1
;
        pop di                          ;recall string offset
        pop si                          ;recall block count
        add di,8                        ;go to "SUBALLOC"
        add si,2                        ;increment word index
        cmp si,two                      ;compare second loop
        push si                         ;preserve block count
        push di                         ;preserve string offset
        jle LOOP4
;
        @DosFreeSeg msel1
;
        @DosExit 1,0
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.15 (*Concluded*)

Then 16,384 bytes are written in blocks of 8 bytes with "MEMORY ". The service call

```
@DosSubSet msel1, mflag, msize
```

initializes the segment for suballocation. Here msel1 is the allocated segment selector; mflag is set to 1, indicating that a segment is being initialized; and msize is the original segment size.

The call

```
@DosSubAlloc msel1, moffset, msize1
```

returns an offset in the segment pointing to the start of the suballocated block whose size is msize1 (in this case 8192 bytes). The parameter msel1 is, of course, the segment selector.

Figure 3.16 illustrates the operation of this program based on CodeView output. In Figure 3.16a the initial load of the segment 006F:0000 to 006F:3FFF is indicated. Here the end of the segment is demonstrated to contain "MEMORY ". Next the suballocation is performed and in Figure 3.16b this is illustrated with "SUBALLOC" loaded up to address 000F:2007. Note that there is a slight offset within the segment for the start of the suballocated block. This offset is 8 bytes and results in an overall shift by this number of bytes from the start of the segment.

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!      SUBALLO.EXE

003F:0053 A10200        MOV      AX,Word Ptr [0002]
003F:0056 50            PUSH     AX
003F:0057 A10400        MOV      AX,Word Ptr [0004]
003F:005A 50            PUSH     AX
003F:005B A10000        MOV      AX,Word Ptr [0000]
003F:005E 50            PUSH     AX
003F:005F 9A0000A700    CALL     00A7:0000
003F:0064 A10200        MOV      AX,Word Ptr [0002]
003F:0067 50            PUSH     AX
003F:0068 B82F00        MOV      AX,002F

>d es:0x3ff0
006F:3FF0  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:4000  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4010  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4020  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4030  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4040  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4050  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
006F:4060  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
>
```

(a)

```
≡ File  Search  View  Run  Watch  Options  Calls  Trace!  Go!      SUBALLO.EXE

003F:0050 A30400        MOV      Word Ptr [0004],AX
003F:0053 A10200        MOV      AX,Word Ptr [0002]
003F:0056 50            PUSH     AX
003F:0057 A10400        MOV      AX,Word Ptr [0004]
003F:005A 50            PUSH     AX
003F:005B A10000        MOV      AX,Word Ptr [0000]
003F:005E 50            PUSH     AX
003F:005F 9A0000A700    CALL     00A7:0000
003F:0064 A10200        MOV      AX,Word Ptr [0002]

006F:0070  53 55 42 41 4C 4C 4F 43-53 55 42 41 4C 4C 4F 43  SUBALLOCSUBALLOC
>d es:0x1ff0
006F:1FF0  53 55 42 41 4C 4C 4F 43-53 55 42 41 4C 4C 4F 43  SUBALLOCSUBALLOC
006F:2000  53 55 42 41 4C 4C 4F 43-00 00 FC 1F 52 59 20 20  SUBALLOC....RY
006F:2010  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:2020  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:2030  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:2040  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:2050  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
006F:2060  4D 45 4D 4F 52 59 20 20-4D 45 4D 4F 52 59 20 20  MEMORY  MEMORY
>
```

(b)

Figure 3.16 Facsimile CodeView output, illustrating (a) end of memory block for original allocation and (b) end of suballocated block with offset of 8 bytes. Cursor located in hidden portion of screen. (Courtesy of the Microsoft Corporation.)

3.3 MULTITASKING

A major OS/2 enhancement (over DOS) is the ability to execute multiple tasks and segregate each task's parameter space so that no mixing occurs. The OS/2 implementation relies heavily on 80286 (and 80386) Protected Mode hardware features. Two threads, which exist as single entities with shared system resources, exist as stand-alone modules with their own system resources and can execute as separate tasks in Protected Mode. In this section we examine briefly the creation of threads and processes.

3.3.1 Semaphores

Before beginning our examination of task generation, however, it is necessary to consider synchronization. Assume, for example, that a given task depends on the outcome of a second task at some point in the first task's execution. Clearly, when the first task is started it must be synchronized with the second task to ensure that the proper data become available when needed. If no requirement for synchronization exists, the two tasks can execute independently and are said to be asynchronous with respect to each other.

A very important mechanism for achieving synchronization is the semaphore: RAM semaphores and system semaphores are considered in this book. A typical prescription for creating and accessing a RAM semaphore within a process (two threads) is as follows:

Thread 1

```
...
@DosSemSet  sem_handle
...
call  to  2nd  thread
...
@DosSemWait  sem_handle,-1
...
```

Thread 2

```
...
activity  to  be  synchronized
...
@DosSemClear  sem_handle
```

Here the semaphore is set and the second thread called. Meanwhile the first thread waits for the semaphore to clear. When the second thread clears the semaphore, the first thread resumes execution. Only a handle, sem_handle, is used to pass information about the semaphore. This can be passed to a second independently compiled (or assembled) process via a shared memory area; however, in the illustration above it has been assumed that both threads are common to the same process and sem_handle appears in the process data area (as a double word).

System semaphores are used commonly between diverse processes and have the following general form:

Data area 1

```
...
no_excl     dw   1                   ;no exclusive
aseml       db   '\SEM\SDAT.DAT,0    ;semaphore name
sem_hdl1    dd   0                   ;handle
no_to       dd   -1                  ;no time out
...
```

Process 1

```
...
@DosCreateSem       no_excl, sem_hdl1, aseml
@DosSemSet          sem_hdl1
...
call to execute 2nd process
...
@DosSemWait         sem_hdl1,no_to
...
```

and

Data area 2

```
...
aseml      db   '\SEM\SDAT.DAT',0  ;semaphore common name
sem_hdl1   dd   0                  ;handle
no_to      dd   -1                 ;no time out
```

Process 2

```
...
@DosOpenSem    sem_hdl1,aseml
...
activity to be synchronized
...
@DosSemClear   sem_hdl1
...
```

We see that the contrast between the two types is that system semaphores require a name and hence can be accessed from disjoint segments. RAM semaphores simply require a common handle. Fast-safe RAM semaphores are used by dynalink libraries.

3.3.2 Creating a Thread

Figure 3.17a is the flowchart for a program that generates two threads using RAM semaphores for synchronization. Figure 3.17b shows the actual code used in this process. The first thread clears the screen, writes message msg_p0 to the display,

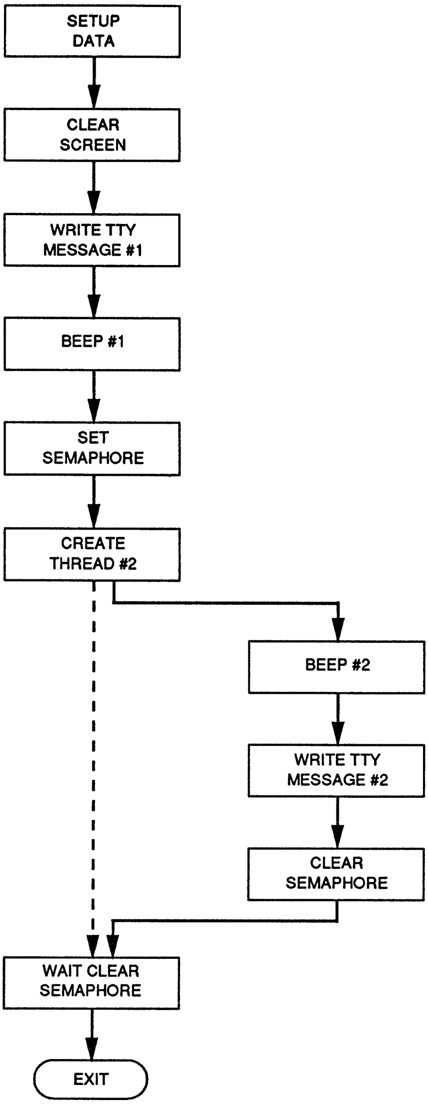


Figure 3.17a Flowchart for a program that generates two threads using RAM semaphores for synchronization.

```
        PAGE 55,132
TITLE CKTH1 -- Check thread generation (ckth1.asm)
;
;       DESCRIPTION: This routine verifies that a thread is
;       generated.
;
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
;
error1  macro
        local ERROR12
        or ax,ax
        jz ERROR12
           jmp ERROR11
ERROR12:
        endm
;
dgroup  GROUP   data
;
STACK1  SEGMENT WORD    STACK    'STACK1'        ;Stack for thread1
        dw      1024 dup(?)
stk1end equ     $
STACK1  ENDS
;
STACK   SEGMENT WORD    STACK    'STACK'         ;Stack for main program
        dw      1024 dup(?)
STACK   ENDS
;
DATA    SEGMENT WORD PUBLIC 'DATA'
;
result  dw      0                                ;Exit code from main
action  equ     1                                ;Action code from main
;
msg_p0  db      'This is the main OS/2 thread'
        db      0DH                              ;Carriage return
        db      0AH                              ;Line feed
lmsg_p0 equ     $-msg_p0                         ;Length message zero
;
msg_p1  db      'This is a separate OS/2 thread'
        db      0DH                              ;Carriage return
        db      0AH                              ;Line feed
lmsg_p1 equ     $-msg_p1                         ;Length message one
;
msg_p2  db      'An error occurred on thread open'
        db      0DH                              ;Carriage return
        db      0AH                              ;Line feed
lmsg_p2 equ     $-msg_p2                         ;Length message two
;
viohdl  equ     0                                ;Video handle
;
freq            dw      5000                     ;5000 Hz
duration        dw      500                      ;500 msec
;
;               ----------------------------------------
;                       Thread 1 parameters
;               ----------------------------------------
;
prgmadd         dd      thread1                  ;Address thread1
```

Figure 3.17b Program illustrating two threads that use RAM semaphores for synchronization. The speaker is beeped and a message written.

```
stk_adr1        dd      stk1end                 ;End STACK1
threadID        dw      0                       ;thread1 I.D.
thd1_exit_code  dw      0                       ;Thread 1 exit code
;
thd_sem1        dd      0                       ;Semaphore thread1
sem_hdl1        dd      thd_sem1                ;Address thd_sem1
no_to           dd      -1                      ;No time out
;
tr              dw      0                       ;Top row screen
lc              dw      0                       ;Left corner
br              dw      23                      ;Bottom row
rc              dw      79                      ;Right corner
no_line         dw      25                      ;Number blanked lines
blank           dw      0007H                   ;Blank attribute
;
;               -----------------------------------------
;
DATA    ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK
OS21    PROC    FAR
;
        call    cls
;
        @VioWrtTTY      msg_p0,lmsg_p0,viohdl   ;Write message one
        error1
;
        @DosBeep        freq,duration           ;Beep speaker
        error1

;
        @DosSemSet      sem_hdl1                ;Set RAM semaphore
        error1
;
        @DosCreateThread        prgmadd,threadID,stk_adr1
        error1
;
        jmp CONT
ERROR11:
        @VioWrtTTY      msg_p2,lmsg_p2,viohdl   ;Write error message
        jmp ENDD
CONT:
;
        @DosSemWait     sem_hdl1,no_to          ;Wait for semaphore clear
;
ENDD:
        @DosExit action,result                  ;Exit
;
OS21    ENDP
;
thread1 PROC    FAR
;
        @DosBeep        freq,duration           ;Beep speaker
;
        @VioWrtTTY      msg_p1,lmsg_p1,viohdl   ;Write message two
;
        @DosSemClear    sem_hdl1                ;Clear semaphore
;
        @DosExit action,thd1_exit_code          ;Exit thread1
;
thread1 ENDP
;
cls     PROC    NEAR
;
        @VioScrollUp    tr,lc,br,rc,no_line,blank,viohdl
```

Figure 3.17b (*Continued*)

```
;
            ret
cls         ENDP
;
CSEG        ENDS
            END       OS21
```

Figure 3.17b *(Concluded)*

beeps the speaker, sets a semaphore, turns on thread1, waits for the semaphore to clear, and exits the process. The second thread, thread1, beeps the speaker, writes msg_p1 to the display, clears the semaphore, and exits thread1. Synchronization is needed because both threads access the display and collisions will result if they run asynchronously.

Figure 3.18a is the flowchart for a program that generates random boxes to the screen, one at a time. The program creates a box of random size (again, in our constraint of 200 x 200 pixels for CGA mode), erases the box, and continues (creating and erasing boxes). The box creation occurs as a separate thread running asynchronously from the main thread. The main thread, once having turned on the box generator thread, simply waits for a keyboard input to terminate the process. Both threads run as part of the same process.

The program code is presented in Figure 3.18b, where the main thread clears the screen, sets CGA mode and clears the screen again, locks the display and gets a selector to the physical screen buffer, beeps the speakers, turns on thread1, and waits for a keyboard input. Following a keyboard input, the screen is unlocked, standard mode resumed (80 x 25), the screen cleared again, and the process exited.

Meanwhile the second thread, once started, first beeps the speaker and then enters an infinite loop. Within this loop a set of random box corners are generated and the box drawn on the display, as indicated above, with a call to boxx. The pixel (pel) attribute is set to unity for this call. Next, the box is erased by repeating the call with the pixel attribute set to zero.

In general, the instruction

```
@DosExit  action,  result
```

will stop both threads from executing when called from the parent. This happens only in response to the keyboard input, which is sensed using @KbdStringIn.

Figure 3.19 contains the support routines used by the box generating program: boxx, clsCGA, wdot, lineh, and linev. Note that some of the routines are slightly different from their counterparts given in GRAPHLIB.LIB (boxx is FAR, for example). Also, note that the second thread procedure is of distance attribute FAR even though it is defined within the same segment as the main thread. This allows the second thread to pass a full 32-bit address for its entry point. Multiple threads within the same process should be used when the task in question is reasonably simple and can be modularized within the same segment.

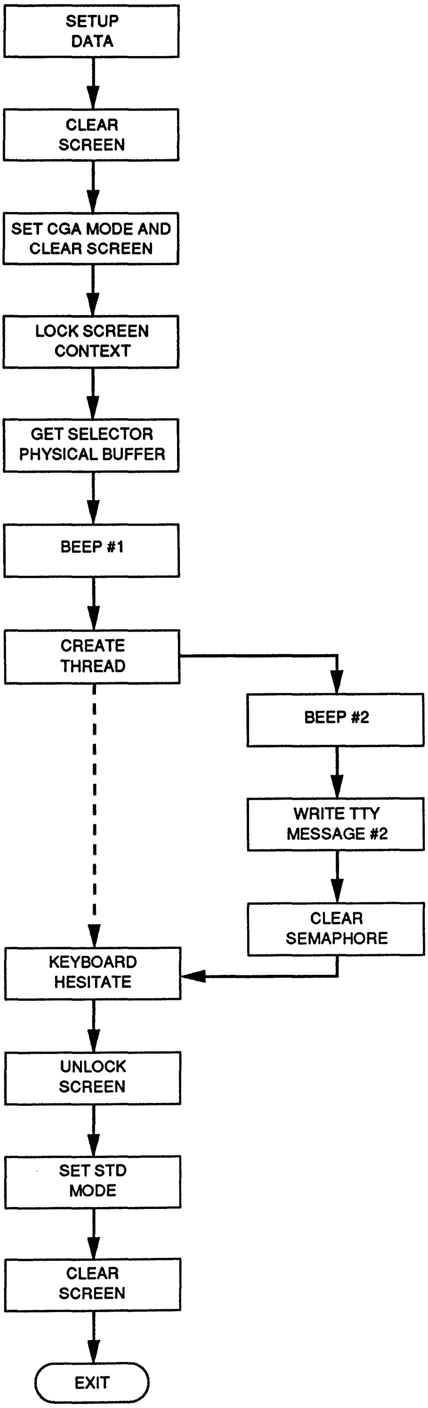


Figure 3.18a Flowchart for a program that generates single random boxes using multiple threads.

```
PAGE 55,132
TITLE UNOS251 - This is the calling OS/2 program (UNOS251.ASM)
;
;       DESCRIPTION: This program "single" random plots boxes in protected
;       mode. Graphics mode 05H is used to display the boxes. This routine
;       employs multithreading to generate the boxs which are
;       generated randomly (100 boxes in square 200 x 200).
;
.8087
;
EXTRN   boxx:FAR,clsCGA:FAR
;
PUBLIC  viohdl,CGAm,lmodeE,typeCGA,colCGA
PUBLIC  txtcCGA,txtrCGA,hrCGA,vrCGA,STDm,lmode80,type80,col80
PUBLIC  txtc80,txtr80,hr80,vr80,waitf,dstat,PVBPtr1,bufst1,buflen1,physel1
PUBLIC  MASK1,MASK11,OFFSET1,four,xx,dummy,two,xxx,eighty,row,col
PUBLIC  address1,x,y,xb,xe,ye,yb,xxxx
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
;
dgroup  GROUP   data
;
STACK1  SEGMENT WORD STACK 'STACK1'
        dw      1024 dup(?)
stk1end equ     $
STACK1  ENDS
;
STACK   SEGMENT WORD STACK 'STACK'      ;Stack for thread
        dw      1024 dup(?)
STACK   ENDS
;
DATA    SEGMENT WORD PUBLIC 'DATA'
;
viohdl  equ     0                       ;Required video handle
result  dw      0                       ;Completion code
action  equ     1                       ;Terminates current thread
action1 equ     0                       ;Thread termination action
tr      dw      0                       ;Top row screen clear
lc      dw      0                       ;Left column screen clear
br      dw      23                      ;Bottom row screen clear
rc      dw      79                      ;Right column screen clear
no_line dw      25                      ;Number lines scrolled
blank   dw      0007H                   ;Blank character pair
;
CGAm    label   FAR                     ;Video mode structure-CGA
lmodeE  dw      12                      ;Structure length
typeCGA db      00000111B               ;Mode identifier
colCGA  db      2                       ;Color option-Mode 5
txtcCGA dw      40                      ;text characters/line-ignore
txtrCGA dw      25                      ;text lines-ignore
hrCGA   dw      320                     ;horizontal resolution
vrCGA   dw      200                     ;vertical resolution
;
STDm    label   FAR                     ;Video mode structure-80x25
lmode80 dw      12                      ;Structure length
type80  db      00000001B               ;Mode identifier-Mode 3+
col80   db      4                       ;Color option
txtc80  dw      80                      ;text characters/line
txtr80  dw      25                      ;text lines
```

Figure 3.18b Main program for the “single” random box routine.

```
hr80    dw      720                         ;horizontal resolution
vr80    dw      400                         ;vertical resolution
;
kbd_buf db      80                          ;Keyboard buffer
lkbd_buf dw     $-kbd_buf                   ;Length keyboard buffer
iowait  dw      0                           ;Wait for CR
kbdhdl  equ     0                           ;Keyboard handle
;
waitf   equ     1                           ;Screen waiting status
dstat   db      ?                           ;Returned status
;
PVBPtr1 label   FAR                         ;Video buffer structure
bufst1  dd      0B8000H                     ;Start physical address
buflen1 dd      4000H                       ;Buffer length
physel1 dw      0                           ;OS/2 screen buffer selector
;
MASK1   db      ?                           ;PEL byte mask
MASK2   db      01H                         ;PEL byte mask--do
MASK22  db      00H                         ;PEL byte mask--undo
MASK11  dw      0001H                       ;Odd/even row mask
OFFSET1 dw      2000H                       ;Odd row buffer offset
four    dw      4
xx      dw      ?                           ;PEL attribute parameter
dummy   dw      ?                           ;80287 dummy "pop"
two     db      2
xxx     db      ?                           ;Output value
eighty  dw      80
zero    dw      0
one     dw      1
row     dw      ?                           ;row
col     dw      ?                           ;column
address1        dw      ?                   ;Address screen dot
rndret  db      ?                           ;random no. returned
;
x       dw      ?                           ;Box col parameter
y       dw      ?                           ;Box row parameter
xb      dw      ?                           ;Start column
xe      dw      ?                           ;End column
yb      dw      ?                           ;Start row
ye      dw      ?                           ;End row
;
;       -------------------------------------------------------------
;                       Second Thread Variables
;       -------------------------------------------------------------
rnd1            dw      1                   ;random seed
prgmadd         dd      thread1             ;address thread
stk_adr1        dd      stk1end             ;end of thread stack
threadID        dw      0                   ;thread ID
xxxx            db      ?,?,?,?             ;box corner buffer
thd1_exit_code  dw      0                   ;thread1 exit code
;
thd_sem1        dd      0                   ;Semaphore thread1
sem_hdl1        dd      thd_sem1            ;Address thd_sem1
no_to           dd      -1                  ;No time out
freq            dw      5000                ;frequency beep in Hz
duration        dw      500                 ;duration beep in millisec
;
;       ----------------------------------------------------------------
;
;
DATA    ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume cs:CSEG,ds:dgroup,ss:STACK
OS21    PROC    FAR
;
        call cls                                        ;Clear screen
```

Figure 3.18b (*Continued*)

```
;
        @VioSetMode CGAm,viohdl                         ;Set CGA graphics mode
;
        call clsCGA                                     ;Clear CGA screen
;
        @VioScrLock waitf,dstat,viohdl                  ;Lock screen context
;
        @VioGetPhysBuf PVBPtr1,viohdl                   ;Get physical buff sel
        push physel1                                    ;Save selector
        pop es                                          ;Load selector into es
;
        @DosBeep        freq,duration                   ;Beep speaker
;
        @DosCreateThread        prgmadd,threadID,stk_adr1
;
        @KbdStringIn kbd_buf,lkbd_buf,iowait,kbdhdl     ;hesitate
;
        @VioScrUnlock viohdl                            ;Unlock screen
;
        @VioSetMode STDm,viohdl                         ;80 x 25 alpha mode
;
        call cls
;
        @DosExit action,result                          ;Terminate process
;
OS21    ENDP
;
cls     PROC    NEAR
;
        @VioScrollUp tr,lc,br,rc,no_line,blank,viohdl   ;STD screen clear
;
        ret
cls     ENDP
;
xload   PROC    NEAR
        mov bh,0                        ;Clear upper register half
        mov bl,al                       ;al = start
        mov xb,bx                       ;Load xb less than 199
        mov bh,0                        ;Clear upper register half
        mov bl,ah                       ;ah = end
        mov xe,bx                       ;Load xe less than 199
        ret
xload   ENDP
;
yload   PROC    NEAR
        mov bh,0                        ;Clear upper register half
        mov bl,al                       ;al = start
        mov yb,bx                       ;Load yb less than 199
        mov bh,0                        ;Clear upper register half
        mov bl,ah                       ;ah = end
        mov ye,bx                       ;Load ye less than 199
        ret
yload   ENDP
;
thread1 PROC    FAR
;
        @DosBeep        freq,duration   ;Beep speaker
;
        mov ax,one                      ;random seed
        mov rnd1,ax                     ;load r.n. parameter
;
        lea bp,xxxx                     ;4 byte buffer
;
LOOP0:                                  ;unterminated loop
        mov cx,4                        ;box corner count
        mov di,0                        ;r.n. memory index
LOOP00:
```

Figure 3.18b (*Continued*)

```
        call ldmem                          ;load random memory values
        mov al,rndret                       ;move r.n. into register
        mov ds:[bp+di],al                   ;save r.n. in memory
        inc di                              ;increment r.n. memory index
        loop LOOP00
;
        mov al,ds:[bp]                      ;1st r.n. value
        mov ah,ds:[bp+1]                    ;2nd r.n. value
        cmp ah,al                           ;check 2nd different than 1st
        jne EELSE1                          ;jump if not equal
           mov al,170                       ;move in arbitrary value l.t. 200
           mov ah,180                       ;move in different value
           call xload                       ;load xe and xb
           jmp IIF1
EELSE1:
        cmp ah,al                           ;check 2nd less than 1st
        jle ELSE1                           ;jump if less or equal
           call xload                       ;if g.t. calculate xb and xe
           jmp IIF1
ELSE1:
           mov bl,al                        ;2nd g.t. 1st -- swap
           mov al,ah                        ;swap
           mov ah,bl                        ;reload
           call xload                       ;calculate xe and xb
IIF1:
        mov al,ds:[bp+2]                    ;3rd r.n. value
        mov ah,ds:[bp+3]                    ;4th r.n. value
        cmp ah,al                           ;check 3rd different than 4th
        jne EELSE2                          ;jump if not equal
           mov al,170                       ;move in arbitrary value l.t. 200
           mov ah,180                       ;move in different value
           call yload                       ;load ye and yb
           jmp IIF2
EELSE2:
        cmp ah,al                           ;check 4th less than 3rd
        jle ELSE2                           ;jump if less or equal
           call yload                       ;if g.t. calculate yb and ye
           jmp IIF2
ELSE2:
        mov bl,al                           ;3rd g.t. 4th -- swap
        mov al,ah                           ;swap
        mov ah,bl                           ;reload
        call yload                          ;calculate yb and ye
IIF2:
        push xb                             ;preserve box parameters
        push xe
        push yb
        push ye
        mov al,MASK2                        ;PEL value set
        mov MASK1,al                        ;load dummy
;
        call boxx                           ;write box
;
        pop ye                              ;recall box parameters
        pop yb
        pop xe
        pop xb
        mov al,MASK22                       ;PEL value black
        mov MASK1,al                        ;load dummy
;
        call boxx                           ;undo box
;
        jmp LOOP0                           ;jump unterminated loop
;
        @DosExit        action1,thd1_exit_code
;
```

Figure 3.18b *(Continued)*

```
thread1 ENDP
;
ldmem   PROC    NEAR
;
        push ax
        push bx
        push dx
;
        mov dx,0                    ;load upper multiplicand
        mov ax,rnd1                 ;load previous r.n.
        mov bx,2053                 ;multiplier
        mul bx
        mov bx,13849                ;load additative constant
        clc
        add ax,bx                   ;add lower order result
        adc dx,0                    ;add carry if needed
        mov bx,0FFFFH               ;load 2(16)-1
        div bx                      ;calculate modulo
        mov ax,dx                   ;mov remainder into ax
        mov rnd1,ax                 ;save r.n.
        mov bx,350                  ;scale r.n. less than 200
        mov dx,0                    ;clear upper dividend
        div bx
        mov ah,0                    ;save al
;
        mov rndret,al               ;returned value byte
;
        pop dx
        pop bx
        pop ax
        ret
ldmem   ENDP
;
CSEG    ENDS
        END     OS21
```

Figure 3.18b *(Concluded)*

3.3.3 Creating Another Process

When multiple processes are to be synchronized the system semaphores are appropriate. System semaphores provide a common link between the two processes through the semaphore name. If RAM semaphores are used, the semaphore handle must be shared with a common data element. Figure 3.20a illustrates a program that generates two processes using system semaphores for synchronization. The semaphore name must be zero terminated and preceded by

```
'\SEM\...'
```

In Figure 3.20b we illustrate this naming with a semaphore called

```
'\SEM\SDAT.DAT', 0
```

and given the variable name aseml.

The second process called by the program in Figure 3.20b is OS2P2.EXE, as

```
PAGE 55,132
TITLE NNOS252 - Supplemental routines for box plotting (NNOS252.ASM)
;
;       DESCRIPTION: These routines set up box plots in CGA mode.  Graphics
;       mode 05H is used to display the box. This set of routines
;       is called by box plotting main routine.  The boxes are created or
;       destroyed depending on MASK1 value.
;
.8087
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
;
EXTRN   viohdl:WORD
EXTRN   CGAm:FAR,lmodeE:WORD,typeCGA:BYTE
EXTRN   colCGA:BYTE,txtcCGA:WORD,txtrCGA:WORD,hrCGA:WORD,vrCGA:WORD
EXTRN   STDm:FAR,lmode80:WORD,type80:BYTE,col80:BYTE,txtc80:WORD,txtr80:WORD
EXTRN   hr80:WORD,vr80:WORD
EXTRN   waitf:WORD,dstat:BYTE,PVBPtr1:FAR,bufst1:DWORD
EXTRN   buflen1:DWORD,physel1:WORD,MASK1:BYTE,MASK11:WORD,OFFSET1:WORD
EXTRN   four:WORD,xx:WORD,dummy:WORD,two:BYTE,xxx:BYTE,eighty:WORD
EXTRN   row:WORD,col:WORD,address1:WORD,x:WORD,y:WORD,xb:WORD,xe:WORD
EXTRN   yb:WORD,ye:WORD
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
PUBLIC  boxx,clsCGA
        assume cs:CSEG
boxx    PROC    FAR
;
;       xb = x-begin,xe = x-end,yb = y-begin,ye = y-end
;
        push ax
        push bx
        push cx
        push dx
;
        mov ax,xb                       ;Check xb l.t. xe
        cmp ax,xe
        jl ELSE10
           xchg ax,xe                   ;Swap xb and xe
           mov xb,ax
ELSE10:
        mov ax,yb                       ;Check yb l.t. ye
        cmp ax,ye
        jl ELSE11
           xchg ax,ye                   ;Swap yb and ye
           mov yb,ax
ELSE11:
        mov ax,yb                       ;Top box line
        mov y,ax
        call lineh                      ;Draw top horizontal line
        mov ax,ye                       ;Bottom box line
        mov y,ax
        call lineh                      ;Draw bottom horizontal line
        mov ax,xb                       ;Left box line
        mov x,ax
        call linev                      ;Draw left vertical line
        mov ax,xe                       ;Right box line
        mov x,ax
        call linev                      ;Draw right vertical line
```

Figure 3.19 Associated support routines for the "single" random box program.

```
;
        pop dx
        pop cx
        pop bx
        pop ax
;
        ret
boxx    ENDP
;
clsCGA  PROC    FAR
;
        @VioScrLock waitf,dstat,viohdl  ;Lock screen context
        @VioGetPhysBuf PVBPtr1,viohdl   ;Get physical buffer
        push physel1                    ;Screen selector
        pop es                          ;Load extra segment
;
        mov bp,0                        ;Start offset zero
        mov al,0                        ;Zero attribute-clear
DO1:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,1F3FH                    ;Check end 1st buffer
        jle DO1
;
        mov bp,2000H                    ;Offset 2nd buffer-odd
        mov al,0                        ;Zero attribute-clear
DO2:
        mov es:[bp],al                  ;Clear byte
        inc bp
        cmp bp,3F3FH                    ;Check end 2nd buffer
        jle DO2
;
        @VioScrUnLock viohdl            ;Unlock screen context
;
        ret
clsCGA  ENDP
;
wdot    PROC    NEAR
;
;       (col,row) = (x,y)
;
        push ax
        push bx
        push cx
        push dx
        push bp
;
        fild four                       ;Load stack with 4
        fild col                        ;ST = col, ST(1) = 4
        fprem                           ;Modulo
        fistp xx                        ;Store remainder in xx
        fistp dummy                     ;Pop stack
        mov al,3
        mov bl,byte ptr xx
        sub al,bl                       ;(3 - col % 4)
        mov ah,0                        ;Clear upper multiplicand
        mul two
        mov cl,al                       ;Shift value for PEL
        mov al,MASK1                    ;PEL color mask
        shl al,cl                       ;Shift to correct PEL
        mov xxx,al                      ;Store buffer value
                                        ;
        mov ax,row                      ;Begin address calculation
        shr ax,1                        ;Divide row by 2
        mov dx,0                        ;Clear upper multiplicand
        mul eighty
```

Figure 3.19 (*Continued*)

```
        mov bx,col                      ;Convert column value to bytes
        shr bx,1
        shr bx,1
        add ax,bx                       ;offset in ax
        mov address1,ax                 ;Save offset base
        mov ax,row                      ;Check even/odd row
        and ax,MASK11                   ;Look for bit 0 set
        cmp ax,0
        jle ELSE1
           mov ax,address1
           add ax,OFFSET1               ;add odd buffer offset
           jmp IF11
ELSE1:
        mov ax,address1
IF11:
        mov bp,ax                       ;screen buffer address
        mov al,xxx                      ;Attribute value for dot
;
        cmp al,0                        ;check PEL black (0)
        je CCC
           or es:[bp],al                ;Write dot
           jmp DDD
CCC:
        mov es:[bp],al                  ;Clear PEL
;
DDD:
        pop bp
        pop dx
        pop cx
        pop bx
        pop ax
;
        ret
wdot    ENDP
;
lineh   PROC    NEAR
;
;       y = row position, xb = begin, xe = end
;
        push ax
        push bx
        push cx
        push dx
;
        mov ax,y                        ;Establish row for wdot
        cmp ax,199                      ;check row l.t. 199
        jg LINE1                        ;jump if greater
           mov row,ax                   ;load "row"
           jmp LINE2
LINE1:
        mov ax,180                      ;load arbitrary value l.t. 199
        mov row,ax                      ;load "row"
LINE2:                          ;
        mov ax,xb                       ;Establish start column
DO10:
        mov col,ax
        push ax                         ;Save column value
        cmp ax,319                      ;check col less than 319
        jle LINE3                       ;jump if l.t.e. 319
           mov ax,319                   ;if greater load arbitrary value
           mov col,ax                   ;load "col"
LINE3:
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall column
        inc ax                          ;Increment column
        cmp ax,xe                       ;Check end horizontal line
        jle DO10
                                        ;
```

Figure 3.19 (*Continued*)

```
        pop dx
        pop cx
        pop bx
        pop ax
;
        ret
lineh   ENDP
;
linev   PROC    NEAR
;
;       x = col position, yb = begin, ye = end
;
        push ax
        push bx
        push cx
        push dx
;
        mov ax,x                        ;Establish column for wdot
        cmp ax,319                      ;check col l.t. 319
        jg LLINE1                       ;jump if greater
           mov col,ax                   ;load "col"
           jmp LLINE2
LLINE1:
        mov ax,319                      ;greater therefore arbitrary value
        mov col,ax                      ;load "col"
                                        ;
LLINE2:
        mov ax,yb                       ;Establish start row
DO20:
        mov row,ax
        push ax                         ;Save row value
        cmp ax,199                      ;check row value g.t. 199
        jle LLINE3                      ;jump if less
           mov ax,199                   ;greater therefore arbitrary value
           mov row,ax                   ;load "row"
LLINE3:
        call wdot                       ;Write dot (col,row)
        pop ax                          ;Recall row
        inc ax                          ;Increment row
        cmp ax,ye                       ;Check end vertical line
        jle DO20
                                        ;
        pop dx
        pop cx
        pop bx
        pop ax
;
        ret
linev   ENDP
;
CSEG    ENDS
        END
```

Figure 3.19 *(Concluded)*

specified under prgrm_nm in the parameter list for @DosExecPgm. The process indicated in Figure 3.20b clears the screen, writes msg_p0 to the display, creates a system semaphore with handle sem_hdl1 and name SDAT.DAT, beeps the speaker, sets the semaphore, and turns on the second process. Following this, the process waits for the system semaphore to clear and then terminates both the second proc-

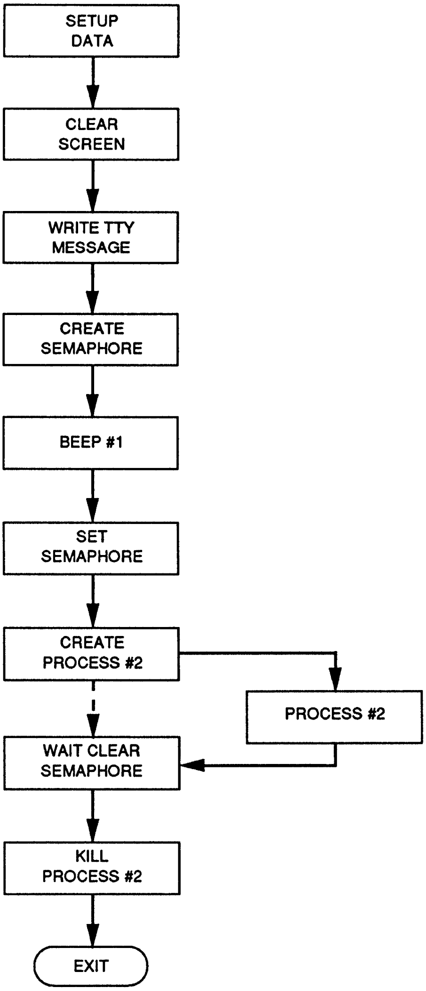


Figure 3.20a Flowchart for program that generates two processes using system semaphores.

ess and itself. Note that this main process accesses the screen at several points (cls and @VioWrtTTY).

Figure 3.21a illustrates the flowchart for the child process, OS2P2.ASM, turned on by the program in Figure 3.20b. The supporting code is shown in Figure 3.21b. The common semaphore name, '\SEM\SDAT.DAT', 0, is again defined by a variable asem1 (not related by symbol to the asem1 appearing in

```
                PAGE 55,132
TITLE CKPR1 -- Check thread generation (ckpr1.asm)
;
;       DESCRIPTION: This routine verifies that a process is
;       generated.
;
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
error1  macro
        local ERROR12
        or ax,ax
        jz ERROR12
           jmp ERROR11
ERROR12:
        endm
;
dgroup  GROUP   data
;
STACK   SEGMENT WORD    STACK   'STACK'         ;Stack for main program
        dw      1024 dup(?)
STACK   ENDS
;
DATA    SEGMENT WORD PUBLIC 'DATA'
;
result  dw      0                               ;Exit code from main
action  equ     1                               ;Action code from main
;
msg_p0  db      'This is the main OS/2 thread'
        db      0DH                             ;Carriage return
        db      0AH                             ;Line feed
lmsg_p0 equ     $-msg_p0                        ;Length message zero
;
viohdl  equ     0                               ;Video handle
;
freq            dw      4000                    ;4000 Hz
duration        dw      500                     ;500 msec
;
;               ----------------------------------------
;                       Semaphore 1 parameters
;               ----------------------------------------
;
no_excl         dw      1                       ;no exclusive
aseml           db      '\SEM\SDAT.DAT',0       ;Name system semaphore
sem_hdl1        dd      0                       ;Address thd_sem1
no_to           dd      -1                      ;No time out
;               ----------------------------------------
;
tr              dw      0                       ;Top row screen
lc              dw      0                       ;Left corner
br              dw      23                      ;Bottom row
rc              dw      79                      ;Right corner
no_line         dw      25                      ;Number blanked lines
blank           dw      0007H                   ;Blank attribute
;
;
;               ----------------------------------------
;                       Process Created Parameters
;               ----------------------------------------
;
```

Figure 3.20b Program that generates two processes using system semaphores.

```
obj_name_buf    dd      10 dup(?)                ;Process name buffer
lobj_name_buf   dw      $-obj_name_buf           ;length buffer
async           dw      1                        ;asynchronous operation
argptr          dw      0                        ;pointer arguments
envptr          dw      0                        ;environment pointer
pid             dw      ?                        ;process ID
                dw      ?
prgm_nm         db      'OS2P2.EXE',0            ;process name
;
;               -----------------------------------------
DATA    ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK
OS21    PROC    FAR
;
        call    cls                              ;Clear screen
;
        @VioWrtTTY      msg_p0,lmsg_p0,viohdl    ;Write message
        error1
;
        @DosCreateSem   no_excl,sem_hdl1,asem1   ;Create system semaphore
        error1
;
        @DosBeep        freq,duration            ;Beep speaker
;
        @DosSemSet      sem_hdl1                 ;Set semaphore
        error1
;
                                                 ;Create child process
        @DosExecPgm obj_name_buf,lobj_name_buf,async,argptr,envptr,pid,prgm_nm
        error1
;
        @DosSemWait     sem_hdl1,no_to           ;Wait for semaphore clear
;
        @DosKillProcess 1,pid                    ;Terminate child process
;
ERROR11:
        @DosExit action,result                   ;Exit
;
OS21    ENDP
;
cls     PROC    NEAR
;
                                                 ;Clear screen
        @VioScrollUp    tr,lc,br,rc,no_line,blank,viohdl
;
        ret
cls     ENDP
;
CSEG    ENDS
        END     OS21
```

Figure 3.20b (*Concluded*)

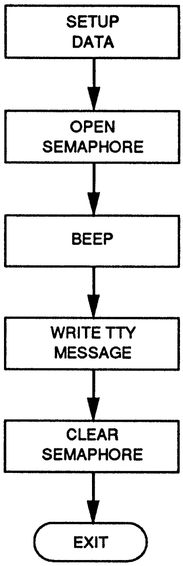


Figure 3.21a Flowchart for a child process, illustrating synchronization using system semaphores.

the parent). This child process opens the semaphore, beeps the speaker, writes a message msg_p1 to the display, and clears the semaphore. The process then terminates itself. Synchronization is needed because both processes access the display.

This very brief example presents the use of multiple processes that must be synchronized. Earlier we used flags in a common data area to accomplish this with the program that displayed 100 random boxes at once. The use of semaphores is a more formal and elegant way to achieve synchronization and does not require a constant polling of the flag to check for process completion. The system does this for us.

```
PAGE 55,132
TITLE OS2P2 -- Check thread generation (os2p2.asm)
;
;       DESCRIPTION: This routine verifies that a 2nd process is
;       generated.  It uses semaphores for synchronization.
;
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
;
error1  macro
        local ERROR12
        or ax,ax
        jz ERROR12
           jmp ERROR11
ERROR12:
        endm
;
dgroup  GROUP   data1
;
STACK1  SEGMENT WORD    STACK   'STACK1'        ;Stack for 2nd process
        dw      1024 dup(?)
STACK1  ENDS
;
DATA1   SEGMENT WORD PUBLIC 'DATA1'
;
result  dw      0                               ;Exit code from process
action  equ     1                               ;Action code from process
;
msg_p1  db      'This is a separate OS/2 process'
        db      0DH                             ;Carriage return
        db      0AH                             ;Line feed
lmsg_p1 equ     $-msg_p1                        ;Length message one
;
viohdl  equ     0                               ;Video handle
;
freq            dw      5000                    ;5000 Hz
duration        dw      500                     ;500 msec
;
;               -----------------------------------------
;                       Semaphore parameters
;               -----------------------------------------
;
asem1           db      '\SEM\SDAT.DAT',0       ;Semaphore name
sem_hdl1        dd      0                       ;Address thd_sem1
no_to           dd      -1                      ;No time out
;
;               -----------------------------------------
;
DATA1   ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK1
OS21    PROC    FAR
;
        @DosOpenSem     sem_hdl1,asem1          ;Open system semaphore
        error1
;
        @DosBeep        freq,duration           ;Beep speaker
        error1
```

Figure 3.21b Child process, illustrating synchronization using system semaphores.

```
;
        @VioWrtTTY      msg_p1,lmsg_p1,viohdl   ;Write message
        error1
;
        @DosSemClear    sem_hdl1                ;Clear semaphore
;
ERROR11:
        @DosExit action,result                  ;Exit process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.21b *(Concluded)*

3.4 INTERPROCESS COMMUNICATIONS

We have seen examples of interprocess communication (IPC) using shared memory and semaphores. OS/2 has three additional mechanisms for achieving such communication: pipes, queues, and signals. Signals basically act like a hardware interrupt and tend to reflect rather specialized interprocess communications [5]. We only mention them. The dominant mechanisms we focus on in this book are the remaining four. Synchronization is a major requirement for processes competing for serial mechanisms [6]. OS/2 solves this problem in a number of ways.

3.4.1 Pipes and Queues

A flowchart for a parent program that passes messages via pipes is illustrated in Figure 3.22a. The code associated with the parent is presented in Figure 3.22b. This program employs several IPC mechanisms in addition to pipes: semaphores, for achieving synchronization, and shared memory, for passing the pipe handle and message length. First the screen is cleared with a call to cls. Next, the shared segment is created and this segment is arbitrarily large (512 words). The call to @DosMakePipe creates the pipe with a read handle, read_hdl, and a write handle, write_hdl. The parameter pflag specifies the pipe length in bytes.

Note that a pipe is anonymous in this context and serves simply as a high-speed buffer area with no name. A system semaphore is created, the speaker beeped, the message msg_p0 written to the pipe buffer using write_hdl, the semaphore set, and a child process executed. The parent then waits for the child to execute and clear the semaphore before it terminates the child and exits.

Figure 3.23a presents the flowchart for the child process. Figure 3.23b contains the code for this process. Initially, the child opens the semaphore, beeps the speaker, and gets a selector to the shared segment. This shared segment is used to obtain a

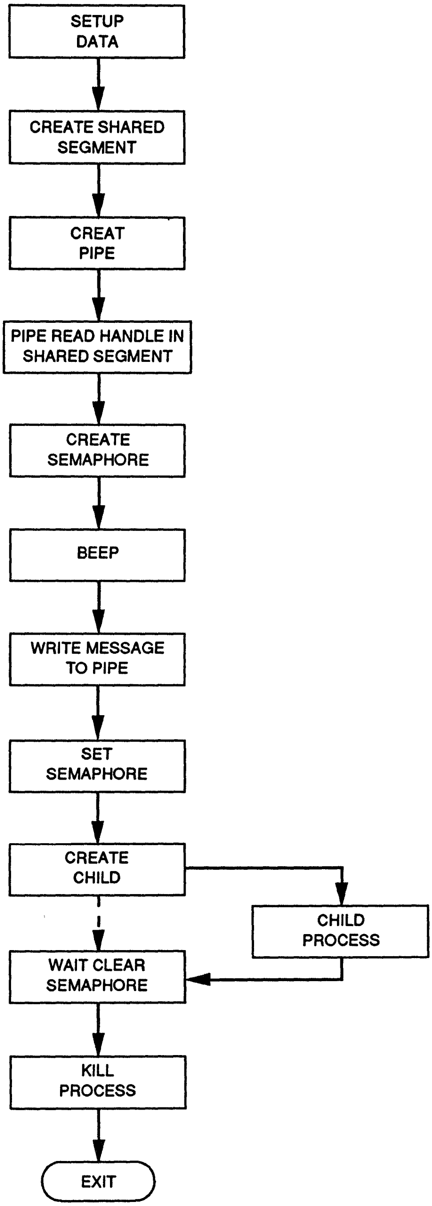


Figure 3.22a Flowchart for a main program that examines pipes for interprocess communication.

```
PAGE 55,132
TITLE PIPEST -- Check pipe generation (pipest.asm)
;
;       DESCRIPTION: This routine verifies that a pipe is
;       generated.
;
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
errorl  macro
        local ERROR12
        or ax,ax
        jz ERROR12
           jmp ERROR11
ERROR12:
        endm
;
dgroup  GROUP   data
;
STACK   SEGMENT WORD    STACK   'STACK'         ;Stack for main program
        dw      1024 dup(?)
STACK   ENDS
;
DATA    SEGMENT WORD PUBLIC 'DATA'
;
result  dw      0                               ;Exit code from main
action  equ     1                               ;Action code from main
;
msg_p0  db      'This is the OS/2 pipe message'
        db      0DH                             ;Carriage return
        db      0AH                             ;Line feed
lmsg_p0 equ     $-msg_p0                        ;Length message zero
;
viohdl  equ     0                               ;Video handle
;
freq            dw      4000                    ;4000 Hz
duration        dw      500                     ;500 msec
;
;               -----------------------------------------
;                       Semaphore 1 parameters
;               -----------------------------------------
;
no_excl         dw      1                       ;no exclusive
aseml           db      '\SEM\SDAT.DAT',0       ;Name system semaphore
sem_hdl1        dd      0                       ;Address
no_to           dd      -1                      ;No time out
;               -----------------------------------------
;
tr              dw      0                       ;Top row screen
lc              dw      0                       ;Left corner
br              dw      23                      ;Bottom row
rc              dw      79                      ;Right corner
no_line         dw      25                      ;Number blanked lines
blank           dw      0007H                   ;Blank attribute
;
;
;               -----------------------------------------
;                       Process Created Parameters
;               -----------------------------------------
;
```

Figure 3.22b Pipe main program.

```
obj_name_buf    dd      10 dup(?)               ;Process name buffer
lobj_name_buf   dw      $-obj_name_buf          ;length buffer
async           dw      1                       ;asynchronous operation
argptr          dw      0                       ;pointer arguments
envptr          dw      0                       ;environment pointer
pid             dw      ?                       ;process ID
                dw      ?
prgm_nm         db      'PIPECL.EXE',0          ;process name
;
;               -----------------------------------------
;
;               -----------------------------------------
;                       Pipe Parameters
;               -----------------------------------------
;
read_hdl        dw      ?                       ;Pipe read handle
write_hdl       dw      ?                       ;Pipe write handle
pflag           dw      256                     ;Pipe length in bytes
bytes_written   dw      ?                       ;bytes written to pipe
;
;               -----------------------------------------
;
;               -----------------------------------------
;                       Shared Memory Parameters
;               -----------------------------------------
;
msize   dw      512                             ;Shared buffer size
msell   dw      ?                               ;Shared selector
shrname db      '\SHAREMEM\SDAT1.DAT',0         ;Shared buffer name
;
;               -----------------------------------------
;
zero    dw      0
one     dw      1
;
DATA    ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK
OS21    PROC    FAR
;
        call    cls                             ;Clear screen
;
        @DosAllocShrSeg msize,shrname,msell
        error1
        push msell                              ;preserve selsctor
        pop es                                  ;selector in extra segment
        mov bp,zero                             ;index equal 0
;
        @DosMakePipe read_hdl,write_hdl,pflag   ;Create pipe
        error1
        mov ax,read_hdl                         ;transfer read handle
        mov es:[bp+2],ax                        ;handle in extra segment
;
        @DosCreateSem   no_excl,sem_hdl1,aseml  ;Create system semaphore
        error1
;
        @DosBeep        freq,duration           ;Beep speaker
;
        @DosWrite write_hdl,msg_p0,lmsg_p0,bytes_written
        error1
        mov ax,bytes_written                    ;transfer message length
        mov es:[bp+4],ax                        ;length in buffer
;
        @DosSemSet      sem_hdl1                ;Set semaphore
        error1
;
```

Figure 3.22b (*Continued*)

```
                                                ;Create child process
        @DosExecPgm obj_name_buf,lobj_name_buf,async,argptr,envptr,pid,prgm_nm
        error1
;
        @DosSemWait     sem_hdl1,no_to          ;Wait for semaphore clear
;
        @DosKillProcess 1,pid                   ;Terminate child process
;
ERROR11:
        @DosExit action,result                  ;Exit
;
OS21    ENDP
;
cls     PROC    NEAR
;
                                                ;Clear screen
        @VioScrollUp    tr,lc,br,rc,no_line,blank,viohdl
;
        ret
cls     ENDP
;
CSEG    ENDS
        END     OS21
```

Figure 3.22b (*Concluded*)

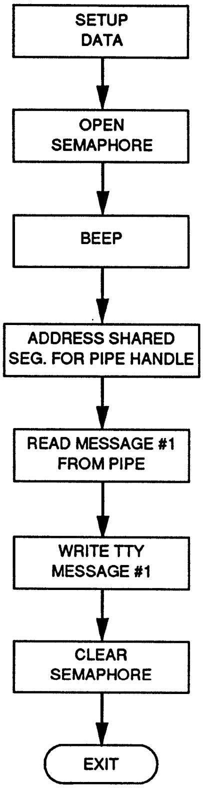


Figure 3.23a Flowchart for a child process that illustrates pipes used for interprocess communications.

```
PAGE 55,132
TITLE PIPECL -- Check pipe generation (pipecl.asm)
;
;       DESCRIPTION: This routine verifies that a pipe is
;       generated.  It uses semaphores for synchronization.
;
;
        .sall
;
        .xlist
                INCL_BASE equ 1
                include os2def.inc
                include bse.inc
        .list
;
error1  macro
        local ERROR12
        or ax,ax
        jz ERROR12
           jmp ERROR11
ERROR12:
        endm
;
dgroup  GROUP   data1
;
STACK1  SEGMENT WORD    STACK   'STACK1'        ;Stack for 2nd process
        dw      1024 dup(?)
STACK1  ENDS
;
DATA1   SEGMENT WORD PUBLIC 'DATA1'
;
result  dw      0                               ;Exit code from process
action  equ     1                               ;Action code from process
viohdl  equ     0                               ;Video handle
;
freq            dw      5000                    ;5000 Hz
duration        dw      500                     ;500 msec
;
;               ----------------------------------------
;                       Semaphore parameters
;               ----------------------------------------
;
aseml           db      '\SEM\SDAT.DAT',0       ;Semaphore name
sem_hdl1        dd      0                       ;Address
no_to           dd      -1                      ;No time out
;
;               ----------------------------------------
;
zero    dw      0
;
;               ----------------------------------------
;                       Shared Buffer Parameters
;               ----------------------------------------
;
shrsel  dw      ?                               ;selector
shrname db      '\SHAREMEM\SDAT1.DAT',0         ;buffer name
;
;               ----------------------------------------
;
;               ----------------------------------------
;                       Pipe Parameters
;               ----------------------------------------
;
read_hdl        dw      ?                       ;read handle
lmsg            dw      ?                       ;length message
buffer          db      256 dup(?)              ;buffer length
```

Figure 3.23b Routine for a child process, illustrating pipes for interprocess communications.

```
bytes_read      dw      ?                       ;actual bytes read
;
;               ----------------------------------------
;
DATA1   ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK1
OS21    PROC    FAR
;
        @DosOpenSem     sem_hdl1,aseml          ;Open system semaphore
        error1
;
        @DosBeep        freq,duration           ;Beep speaker
        error1
;
        @DosGetShrSeg   shrname,shrsel          ;shared segment
        error1
        push shrsel                             ;preserve selector
        pop es                                  ;load extra segment
        mov bp,zero                             ;initialize index
        mov ax,es:[bp+2]                        ;read handle
        mov read_hdl,ax                         ;specified
        mov ax,es:[bp+4]                        ;message length
        mov lmsg,ax                             ;specified
;
        @DosRead read_hdl,buffer,lmsg,bytes_read
        error1
;
        @VioWrtTTY      buffer,lmsg,viohdl      ;Write message
        error1
;
        @DosSemClear    sem_hdl1                ;Clear semaphore
;
ERROR11:
        @DosExit action,result                  ;Exit process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.23b *(Concluded)*

read handle and the message length for the pipe. The pipe is then read using @DosRead and the message loaded into buffer. The display is then updated with the message content using

```
@VioWrtTTY buffer, lmsg, yiohdl
```

where lmsg is the message length in bytes and viohdl the display handle. It is this access of the display that requires synchronization with the parent.

Following the message write to the screen, the semaphore is cleared and the child process terminates execution. The key step in this code was to ensure that common pipe link exists between the routines. Unlike the semaphore link, which uses a commonly named area ('\SEM\SDAT.DAT',0) across both the child and parent, the pipe handle was passed via a common memory area ('\SHAREMEM\SDAT1.DAT',0).

Figure 3.24a illustrates the main process for a set of programs that demonstrate queue operation. Figure 3.24b presents the associated code for this parent process. The process opens with a call to @DosCreateQueue. The common link between the child and parent is the queue name, '\QUEUES\QDAT.DAT',0. This call returns a queue handle, q_hdl. The speaker is beeped and a child process (named 'QUEUECL.EXE') turned on. The queue is used to pass a 32-bit buffer address from the child process to the parent. The selector value of this address is loaded into es and the offset into bx. This buffer address is contained in the double word buffer1.

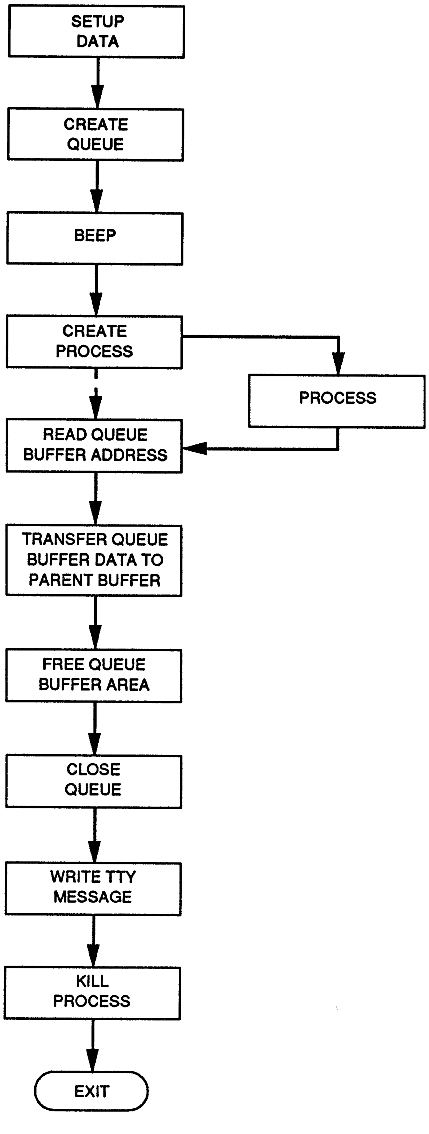


Figure 3.24a Flowchart for a main program, illustrating queues for interprocess communications.

```
PAGE 55,132
TITLE QUEUEST -- Check queue generation (queuest.asm)
;
;       DESCRIPTION: This routine verifies that a queue is
;       generated.
;
;
        .sall                                   ;suppresses macro lists
;
        .xlist                                  ;suppresses source list
                INCL_BASE equ 1                 ;sets IBM macro flag
                include os2def.inc              ;os2 definitions
                include bse.inc                 ;Dos,Vio,Mou, & Kbd
        .list                                   ;turns list on
;
errorl  macro                                   ;exit macro
        local ERROR12                           ;local macro label
        or ax,ax                                ;set ax bits
        jz ERROR12                              ;jump if zero next instruction
           jmp ERROR11                          ;otherwise exit process
ERROR12:
        endm                                    ;end macro
;
dgroup  GROUP   data                            ;data and extra group
;
STACK   SEGMENT WORD    STACK   'STACK'         ;Stack for main program
        dw      1024 dup(?)
STACK   ENDS
;
DATA    SEGMENT WORD PUBLIC 'DATA'
;
result  dw      0                               ;Exit code from main
action  equ     1                               ;Action code from main
;
viohdl  equ     0                               ;Video handle
;
freq            dw      4000                    ;4000 Hz
duration        dw      500                     ;500 msec
selector        dw      ?                       ;allocated segment selector
;
;               -----------------------------------------
;                       Process Created Parameters
;               -----------------------------------------
;
obj_name_buf    dd      10 dup(?)               ;Process name buffer
lobj_name_buf   dw      $-obj_name_buf          ;length buffer
async           dw      1                       ;asynchronous operation
argptr          dw      0                       ;pointer arguments
envptr          dw      0                       ;environment pointer
pid             dw      ?,?                     ;process ID
prgm_nm         db      'QUEUECL.EXE',0         ;process name
;
;               -----------------------------------------
;
;               -----------------------------------------
;                       Queue Parameters
;               -----------------------------------------
;
q_hdl           dw      ?                       ;Queue handle
q_prty          dw      0                       ;Queue ordering priority
q_name          db      '\QUEUES\QDAT.DAT',0    ;name
request         dd      0                       ;Read request parameter
el_prty         dw      0                       ;Element read priority
asemi           dd      0                       ;semaphore
el_code         dw      0                       ;element code
```

Figure 3.24b Main program illustrating queues.

```
no_wait         dw      0                          ;wait processing
lmsg            dw      ?                          ;message length--read
bufferl         dd      0                          ;queue buffer address
buffer          db      256 dup(?)                 ;read buffer
;
;               ----------------------------------------
;
DATA    ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK
OS21    PROC    FAR
;
        @DosCreateQueue q_hdl,q_prty,q_name        ;Create queue
        errorl
;
        @DosBeep        freq,duration              ;Beep speaker
;
                                                   ;Create child process
        @DosExecPgm obj_name_buf,lobj_name_buf,async,argptr,envptr,pid,prgm_nm
        errorl
;
                                                   ;read queue buffer area
        @DosReadQueue q_hdl,request,lmsg,bufferl,el_code,no_wait,el_prty,asemi
        errorl
;
        mov bx,word ptr bufferl                    ;child buffer 32-bit address
        mov ax,word ptr bufferl+2                  ;selector
        mov selector,ax
        push selector
        pop es                                     ;extra segment register
        lea bp,buffer                              ;load data buffer address
        mov cx,lmsg                                ;count limit
        mov di,0                                   ;count index
LOOP1:
        mov al,es:[bx+di]                          ;transfer from queue area
        mov ds:[bp+di],al                          ;transfer to ds buffer
        inc di                                     ;increment index
        loop LOOP1
;
        @DosFreeSeg     selector                   ;free allocated segment
        errorl
;
        @DosCloseQueue  q_hdl                      ;close queue
;
        @VioWrtTTY buffer,lmsg,viohdl              ;write message to screen
        errorl
;
        @DosKillProcess 1,pid                      ;Terminate child process
;
ERROR11:
        @DosExit action,result                     ;Exit
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.24b (*Concluded*)

Next, the length of the buffer is loaded into a loop counter. This value, lmsg, was retrieved from the child process along with bufferl using an @DosReadQueue call. The contents of the buffer pointed to by bufferl are loaded into the buffer, and the segment area pointed at by selector, the segment address associated with the pointer, bufferl, is released. This segment was allocated previously, during execution of the child process, as we shall see shortly. The queue is closed and the message in buffer written to the screen. Finally, the child is terminated and the parent exits back to OS/2.

Figure 3.25a shows the flowchart for the associated child process and Figure 3.25b the corresponding code. The child opens with a beep to the speaker alerting the user that the child has started. Next, the queue is opened (note that the queue name represents the common link between the two processes. At this point the child allocates a segment using @DosAllocSeg. The size of the segment equals the message length and it is a giveable segment (aseg_give=1). The segment allocated has a selector returned (q_w) which is loaded into es and the contents of the message written to this buffer. The speaker is beeped with a slightly different tone, @DosGiveSeg executed (which returns a selector that can be given back to the parent, q_rr), and the 16-bit read selector loaded into the segment portion of a 32-bit pointer to the giveable segment (q_r).

A macro @DosWriteQueue1 has been defined at the beginning of the program and this macro has the fifth statement as pushing a 32-bit value (not address) onto the stack prior to the call to DOSWRITEQUEUE. This macro is called to transfer

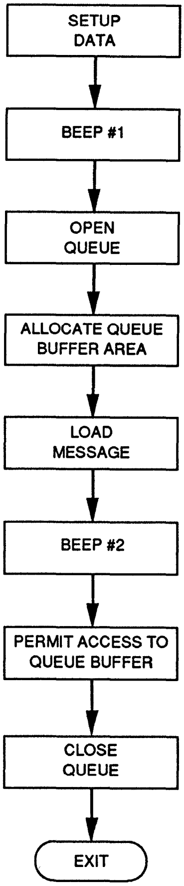


Figure 3.25a Flowchart for a child process, illustrating queues for interprocess communications.

```
PAGE 55,132
TITLE QUEUECL -- Check queue generation (queuecl.asm)
;
;       DESCRIPTION: This routine verifies that a queue is
;       generated.  It uses semaphores for synchronization.
;
;
;
        .sall                                   ;suppress macro listing
;
        .xlist                                  ;suppress source list
                INCL_BASE equ 1                 ;set IBM macro flag
                include os2def.inc              ;include os2 macros
                include bse.inc                 ;Dos,Vio,Mou, & Kbd
        .list                                   ;turn list on
;
errorl  macro                                   ;exit macro
        local ERROR12                           ;local macro label
        or ax,ax                                ;set ax
        jz ERROR12                              ;jump to next instruction
           jmp ERROR11                          ;exit
ERROR12:
        endm                                    ;end macro
;
                                                ;Corrected macro
@DosWriteQueue1 macro   handle,request,length,data,prty
                @define DOSWRITEQUEUE           ;;define API call
                @pushw  handle                  ;;push word handle
                @pushw  request                 ;;push word request
                @pushw  length                  ;;push buffer length
                @pushd  data                    ;;push 32-bit address
                @pushw  prty                    ;;push priority
                call    far ptr DOSWRITEQUEUE   ;;call API function
                endm
;
dgroup  GROUP   data1                           ;load ds and es
;
STACK1  SEGMENT WORD    STACK   'STACK1'        ;Stack for 2nd process
        dw      1024 dup(?)
STACK1  ENDS
;
DATA1   SEGMENT WORD PUBLIC 'DATA1'
;
result  dw      0                               ;Exit code from process
action  equ     1                               ;Action code from process
;
freq            dw      5000                    ;5000 Hz
freq1           dw      2000                    ;2000 Hz
duration        dw      500                     ;500 msec
;
;               ----------------------------------------
;                       Queue Parameters
;               ----------------------------------------
;
q_hdl           dw      ?                       ;queue handle
q_pid           dw      ?                       ;process ID--queue creator
q_name          db      '\QUEUES\QDAT.DAT',0    ;name
request         dw      0                       ;write request parameter
prty0           dw      0                       ;priority message 1
;
;               ----------------------------------------
;
msg_p0  db      'This is a priority 1 message',0DH,0AH
lmsg_p0 dw      $-msg_p0                        ;length
;
```

Figure 3.25b The child process, illustrating queues for interprocess communications.

```
;               ----------------------------------------
;                  Allocated Segment/Queue Parameters
;               ----------------------------------------
;
q_w             dw      0                       ;queue write selector
aseg_give       dw      1                       ;allocated segment giveable
q_r             dd      0                       ;queue read 32-bit pointer
q_rr            dw      ?                       ;queue read selector
;
;               ----------------------------------------
;
DATA1   ENDS
;
CSEG    SEGMENT WORD PUBLIC 'CODE'
        assume  cs:CSEG,ds:dgroup,ss:STACK1
OS21    PROC    FAR
;
        @DosBeep        freq,duration           ;Beep speaker
        errorl
;
        @DosOpenQueue q_pid,q_hdl,q_name        ;open queue
        errorl
;
        @DosAllocSeg    lmsg_p0,q_w,aseg_give   ;allocate segment
        errorl
;
        push q_w                                ;allocated segment selector
        pop es                                  ;pop to extra segment register
        lea bx,msg_p0                           ;offset of message
        mov cx,lmsg_p0                          ;loop count=message length
        mov di,0                                ;zero index
LOOP1:
        mov al,ds:[bx+di]                       ;transfer message
        mov es:[di],al                          ;message to extra segment
        inc di                                  ;increment index
        loop LOOP1
;
        push ds                                 ;reload ds to stack
        pop es                                  ;es=ds
;
        @DosBeep        freq1,duration          ;2nd beep
        errorl
;
        @DosGiveSeg     q_w,q_pid,q_rr          ;get selector
        errorl
;
        lea bx,q_r                              ;32-bit read address
        mov ax,q_rr                             ;16-bit read selector
;       mov ds:[bx+2],ax                                ;load read address
                                                ;write address to queue
        @DosWriteQueuel q_hdl,request,lmsg_p0,q_r,prty0
        errorl
;
        @DosFreeSeg     q_w                     ;free allocated segment
        errorl
;
        @DosCloseQueue  q_hdl                   ;close queue
        errorl
;
ERROR11:
        @DosExit action,result                  ;Exit process
;
OS21    ENDP
CSEG    ENDS
        END     OS21
```

Figure 3.25b (*Concluded*)

the 32-bit pointer to the allocated segment, back to the parent process via the queue. Also transferred with this call is the message length. Finally, the segment is set free for selector q_w. The queue is closed next and the child exits back to the OS/2.

3.4.2 Shared Memory Segments

In several cases among the preceding examples the processes involved employed shared memory segments. There were generally two types of mechanisms employed: giveable segments created using @DosAllocSeg or true shared segments created using @DosAllocShrSeg. Both of these approaches lead to a common sharing of a memory segment. These segments were used to transfer commonly needed information so that two independent (although possibly synchronized) processes could establish a link. It is this need for some sort of common memory reference that characterizes all IPC, and shared memory is a very effective way to achieve this.

3.5 SUMMARY

In this chapter we have looked at memory management, multitasking, and interprocess communications. The goal has been to establish for the reader an introduction to these techniques, with representative examples used to illustrate the mechanisms involved. A major attribute of OS/2 is the ability to access huge segments (greater than 64K). This was demonstrated in Section 3.2.4.

Multitasking is a cornerstone of the operating system. As programming strategies change from the single-threaded way of doing business common throughout the 1980s to more parallel approaches, OS/2 can be expected to move to the forefront of microcomputer operating systems. It must be recognized that multitasking requires a rethinking of how programs are structured in order to be able to take advantage of this feature. Programmers must begin to think in terms of how a given application can be subdivided so that the application can be run efficiently in a multitasked environment. This is a very nontrivial change in programming concept. Without the common availability of well-supported multitasking operating systems, it is, of course, impossible to begin the process of rethinking program structure to fit the multitasking mold. Hence OS/2 truly represents a transition in programming philosophy for the applications programmer. Fortunately, it has a great deal of commonality with earlier systems such as DOS and the Windows executive, and consequently, represents a relatively fluid vehicle for many programmers to enter the world of multitasking.

In Section 3.3 we discussed semaphores, multiple threads, and multiple processes, all essential to a comprehensive multitasking environment. The semaphores treated consisted of two types: RAM and system (with a third being fast-RAM). In Section 3.4 we described the basic vehicles for interprocess communications, which had been alluded to earlier, and illustrated the use of pipes and queues. Shared memory segments were discussed throughout the chapter and signals were mentioned briefly.

REFERENCES

1. *Intel iAPX 286 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, 1985.
2. Motorola Corporation, *MC68000/MC68008/MC68010/MC68HC000 8-/16-/32-Bit Microprocessors User's Manual*, 6th ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
3. McCracken, D. D., *A Guide to PLM Programming for Microcomputer Applications*, Addison-Wesley Publishing Company, Reading, MA, 1978, p. 43.
4. Godfrey, J. T., *Applied C: The IBM Microcomputers*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
5. Duncan, R., *Advanced OS/2 Programming*, Microsoft Corporation, Redmond, WA, 1989, p. 271.
6. Tanenbaum, A. S., *Operating Systems: Design and Implementation*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987, p. 51.

PROBLEMS

3.1 IBM supplies a number of device drivers with the OS/2 libraries. They have extension .SYS. The floppy and fixed disk driver for the IBM PC/AT is called DISK01.SYS, and the driver for the inport Microsoft Mouse is MOUSEA04.SYS, for example (see reference 5, p. 14). What level of protection would you expect these drivers to have? Why?

3.2 Throughout this book the IBM (and Microsoft) macros have been used to access the API services. Typical of these is the call

```
@DosExit  action, result
```

which executes the macro code

```
@DosExit  macro    action, result
          @define  DOSEXIT
          @pushw   action
          @pushw   result
          call     far ptr DOSEXIT
          endm
```

where @define and @pushw are defined as

```
@define   macro    callname
          ifndef   callname
          extrn    callname:far
          endif
          endm
```

and

```
@pushw     macro     parm
           mov       ax,parm
           push      ax
           endm
```

What are the advantages and disadvantages of this approach. Consider, for example, maintenance and clarity of code.

3.3 When using @DosAllocSeg, what must occur for the segment to be created to be giveable? To be discardable?

3.4 When accessing a huge segment, what is essential to achieving operation that ensures no violation of protection?

3.5 What is the dominant feature of interprocess communications that must hold in any multitasking implementation?

3.6 When would you be likely to use RAM semaphores for interprocess communications? To use system semaphores?

3.7 In @DosWriteQueue why must the fourth macro parameter be pushed with @pushd not @pushs? Here

```
@pushs     macro parm
           mov ax,SEG parm
           push ax
           lea ax,parm
           push ax
           endm
```

while

```
@pushd     macro parm
           push ds
           push bx
           mov ax, SEG parm
           mov bx,OFFSET parm
           push word ptr [bx]
           mov ax, [bx+2]
           push bp
           push sp
           pop bp
           xchg [bp+6],ax
           pop bp
           mov ds,ax
           pop ax
           pop bx
           push ax
           endm
```

3.8 When a child process, that is, using semaphores for interprocess communications, completes the execution of a critical area of code, how does it signal the parent?

3.9 Suppose that two processes involve no IPC and contain such code fragments as:

Process 1

```
...
@DosExecPgm...
@error1
;
@DosBeep  freq1,duration

...
```

and

Process 2

```
...
OS21      PROC      FAR
;
          @DosBeep  freq2,duration
          @DosExit  action,result
OS21      ENDP
          ...
```

What are the potential consequences of such code?

3.10 What is the major difference between a pipe and a queue as used in this chapter?

3.11 Compare the various IPC mechanisms.

3.12 When would you use a shared segment as opposed to a giveable segment?

3.13 Outline the API calls for pipe operation using a child process.

3.14 Outline the API calls for queue operation using a child process.

3.15 How is an intraprocess thread differentiated from an interprocess thread? What is preferred for a second task?

3.16 Discuss the usage of DosAllocSeg and DosReAllocSeg in comparison to the use of DosSubAlloc.

PART III
Advanced OS/2 Kernal Programming

4 OS/2 and C

There are a number of C compilers available; however, two that run under OS/2 Protected Mode are the Microsoft C 5.1 Optimizing Compiler [1] and IBM's C/2 Version 1.1 [2]. In this book we use the former compiler. Together with the include files for the assembler (the .inc files), the IBM Toolkit [3] provides a set of C include files (.h files) that contain macros for accessing the Applications Program Interface (API) and Presentation Manager (PM) services. There are some significant difference's between the Version 1.0 and 1.1 Toolkit files, particularly in the definition of the structures used by API service calls. We will adhere to the Version 1.1 definitions; the interested reader is referred elsewhere for the Version 1.0 definitions [4]. The purpose of this chapter is to introduce C programming in the Protected Mode context.

4.1 HIGHER LEVELS OF ABSTRACTION

C, by its very nature as a high-level language (HLL), is more abstract than assembler. This has distinct advantages when developing modular programs because the resulting code is more compact and easier to follow, assuming that the programmer has the language background. It does not necessarily facilitate optimum access to system hardware because the programmer must rely on the C compiler developer to provide these underlying service routines. In many cases, of course, these services are very optimized, but they must have some general-purpose features that could be avoided if tailored assembler code were provided. This book assumes that the reader has a basic familiarity with the C language, as it did for assembler, and Appendix B reviews the C syntax in the Kernighan and Ritchie mold [5].

What do we mean by more abstract? Multiplication is an example. To square the variable x in C, one merely writes

```
x = x * x;
```

To square the same variable in assembler (assuming that x is of word length), one has

```
mov ax, x       ; load accumulator
mov dx, 0       ; clear upper multiplicand
mul x           ; multiply
mov x, ax       ; reload x variable
```

which is a bit more cumbersome. An even more exaggerated example is the line of C code

```
y = (float)(sin(2.*PI*f*t));
```

The conversion from double precision to floating point, alone, is a major system call, as is the reference to the sine mathematical function. These calls would encompass many lines of assembler code to accomplish the same algorithm.

Hence abstraction can be a desirable feature as the programmer moves away from low-level system services and the hardware. Assuming that a programmer's span of attention is limited to some rough measure of lines of code, the HLL allows a more efficient usage of this feature.

4.1.1 The C Include Files

The Toolkit has a number of include files used to set up the API calls and associated variables, types, and structures used by these calls. The Toolkit is highly recommended for users who quickly wish to begin programming OS/2 Protected Mode C, with its function-like interface. The major Toolkit include file is

```
OS2.h
```

which calls

```
#include <OS2def.h>
#include <bse.h>
```

and requires a beginning program statement

```
#define INCL_BASE
```

Hence the first line of code prior to any API call would be

```
#define INCL_BASE
#include <OS2.h>
...
```

Note that OS2.h also has provisions to call pm.h, which loads the PM include routines.

The file bse.h checks to see if INCL_BASE is set and then sets three symbols;

```
INCL_DOS
INCL_SUB
INCL_DOSERRORS
```

and (loads)

```
#include <bsedos.h>    /*Dos calls*/
#include <bsesub.h>    /*Vio,Kbd,Mon calls*/
#include <bseerr.h>    /*Error calls*/
```

where the first of these sets up the Dos prefix API calls. The second loads the Vio, Kbd, and Mon prefix API calls and bseerr.h loads the error calls.

It is worthwhile pointing out a typical difference between the Version 1.0 Toolkit and the Version 1.1. Consider the structure definition for getting the physical buffer:

Version 1.0

```
struct PhysBufData {
unsigned long   buf_start;      /*start byte*/
unsigned long   buf_length;     /*buffer length*/
unsigned        selectors[2];   /*selector*/
};
```

Version 1.1

```
typedef struct  _VIOPHYSBUF{
PBYTE pBuf;      /*pointer to start byte*/
ULONG cb;        /*buffer length*/
SEL asel[1];     /*selector*/
}VIOPHYSBUF;
```

Clearly, to access these two structures, which serve the same purpose, requires radically different calling schemes. The programmer can expect to encounter this type of problem when converting Version 1.0 Protected Mode code to Version 1.1; however, it is generally desirable to use the Toolkit routines because of the abstraction features intrinsic to these calls.

4.1.2 The Low-Level Nature of the API

We know that standard C code can be used for output to the display with calls of the type:

```
printf("This is a display message.\n");
```

This can be accomplished with the standard I/O include file stdio.h and works in Protected Mode as well as Real Mode. To use the API call in equivalent fashion, we would need the more structured statements

```
...
unsigned vio_hdl = 0;          /*video handle*/
char *msg_p = "This is a display message./n";
unsigned lmsg_p = 0;

lmsg_p = strlen(msg_p);
VioWrtTTy((char far*)msg_p,lmsg_p,vio_hdl);
```

Thus the reader can see that the API calls tend to be more cumbersome than standard C code and more low level in nature. Clearly, as the example above highlights, the programmer would want to use the standard I/O routines in this case. Frequently, however, services will be required in Protected Mode that cannot be accomplished using the standard C functions. It is these activities that must access the API directly in low-level fashion. A very good example of this is the screen graphics modes, which require locking the screen and accessing the physical buffer all in conjunction with the mode set. These activities fit well with the notion of low-level calls in C. They correspond to low-level services: accessing the system resources directly.

Generally, many of the API services are of this low-level nature. The reader is cautioned to use the standard C syntax where possible but recognize that the API services are designed to work in a multitasking environment and that some low-level interfacing will therefore always be necessary.

4.1.3 Comparison of C with Assembler

We have already seen several examples of the differences between C syntax and assembly language syntax when used to accomplish the same task. Typically, the assembler is much more detailed and incremental (each instruction accomplishes a much smaller piece of the overall task). As a further simplified example, consider addition in C:

```
y = x1 + x2;
```

To accomplish this same syntax in assembler the following code is required:

```
mov  ax, x2
add  ax, x1
mov  y, ax
```

Again, this assumes word integer arithmetic. If floating-point operations, for example, are to be implemented, the overhead increases dramatically.

Why, then, have we spent time learning the assembler interface in OS/2? A major reason is to understand the low-level nature of the API interface. In order to program the API from any language, the programmer must have a feeling for the syntax at a very basic level. Frequently, access is byte oriented and in order to get C code to function properly, the programmer must have this very basic understanding. The structures and parameter definitions for C calls to the API rely on a low-level interpretation, as found in the assembler calls. When problems arise in the C debugging process, assembler-level understanding of the API services provides invaluable insight into the C function calls.

4.2 INTRODUCTORY C PROGRAMMING WITH OS/2

Many application programs require a reasonable level of mathematical sophistication to achieve their intended computational goals. Generally speaking, assembly language is not the desired vehicle to achieve such sophistication. Modern languages have evolved such that a great deal can be accomplished within a single language to span the requirement of sophistication yet retain the ability to implement low-level services. The C language is such an implementation, and from this point on we shall concentrate on programming for OS/2 in the C context. Of course, we will make an occasional sojourn back to assembly language when the need arises.

4.2.1 C Program Architecture and Structure

Perhaps the easiest way to present the structure of a C program is with a simple example. Figure 4.1 contains a C program that prints the message

```
Input word integer less than 32,768
```

reads the input word integer value, and calls a function times_2(). The function times_2( ) has a single formal parameter that it doubles and converts from integer to floating point. Then the function prints the floating-point value of twice the initial integer to the display with the message

```
2 times the integer value =
```

with the equal sign followed by the value.

What is typical about this code? First, a comment line has been offset with the following form:

```
/* ... */
```

Next, the C files needed by the program have been specified. In this program there is only one, stdio.h, and it is included with the statement

```
#include <stdio.h>
```

```
/* A simple C program to illustrate Protected Mode I/O -- ioprgm.c */

#include <stdio.h>

main()
        {
        int x;                                  /* input variable */

        printf("Input word integer less than 32,768 \n");
        scanf("%d",&x);

        times_2(x);                             /* function */
        }

times_2(y)
        int y;                                  /* formal parameter */
        {
        float z;                                /* floating point */

        z = (float)(2.*y);                      /* double */

        printf("2 times the integer value = %f\n",z);
        }
```

Figure 4.1 The program ioprgm.c, illustrating typical program formatting for C code.

Following this preprocessor area, the main function (called main()) appears and the code contained in this function is subtended within the curly brackets: {...}. The first line of code is a type declaration for x to be of type integer: int. Next the C standard routine, printf(), is called, asking for the word integer input. The string contained in quotations is written to the display and terminated by the escape character, \n, which generates a carriage return and line feed. The scanf() routine is called to read an integer value (%d) into the location (using the address operator,&) specified by x. Finally, the function times_2() is called with x passed as a parameter and main() is then ended.

In the function times_2() the formal parameter, y, is declared to be of type integer and this is declared outside the body of the function. Within the body of the function all variables are locally defined. Here, for example, x is local to times_2() and is of type float (floating point). The value of y is doubled and converted to floating point with the cast: (float). This is used to define z. Next the value of z is output following the message. Note that the parameter specification (%f) corresponds to a floating-point output, while earlier we had (%d) to correspond to an integer format.

Figure 4.2 illustrates the MAKE utility file used to compile and link the C code. In general the reader is referred to his or her compiler manual to understand the nature of this, but briefly the command

```
cl  -c  -Zi  -Os  -FPc  /Fcioprgm.cod  ioprgm.c
```

compiles the program (-c indicates do not link yet) and sets it up for the CodeView debugger (-Zi). The -Os parameter tends to reduce code size during optimization and

```
        ioprgm.obj: ioprgm.c
        cl -c -Zi -Os -FPc /Fcioprgm.cod ioprgm.c

ioprgm.exe: ioprgm.obj
        link /CO ioprgm.obj,ioprgm,ioprgm,slibce.lib/NOE os2.lib/NOE,,
```

Figure 4.2 MAKE file for ioprgm.c.

-FPc generates floating-point calls and selects the emulator math package. The statement

```
/Fcioprgm.cod
```

generates a mixed assembler and C code output file, and ioprgm.c indicates the C source file.

The next set of lines in the MAKE file corresponds to the link operation. The /CO sets up CodeView. The first field contains the object modules(s); the second field contains ioprgm, where the default extension is .exe the run filename; and the third field contains ioprgm, the map filename with default .map. Next, the libraries are indicated, with the /NOE option that prevents multiple definitions of the same name.

Figure 4.3 contains the list file for the mixed assembler and C source code (the .COD file). It is important to examine this file because it establishes the complexity of the C code in relation to the required assembly language instructions needed to represent each line of this C code. Note the large number of external routines called to implement the program appearing in Figure 4.1: __acrtused, _printf, _scanf, __chkstk, __fldw, __fmnld, __fstsp, __flds, __fstdp, and __fltused. The text segment is _TEXT and the data segment _DAT. The data segment contains the strings of text and the integer formal specifier, %d. Aside from the initial setup for the routine _main, the print output asking for the integer less than 32,768 is accomplished with the assembly code following the designation for line 9. Next the integer is read in and a call made to times_2(). Finally, the main procedure ends. The times_2() code follows as a NEAR procedure with a number of calls to floating-point routines that emulate the coprocessor. These routines all begin with "f ".

The code in Figure 4.3 is instructive in that it illustrates the general techniques for generating assembly language instructions from C syntax. Note that no obvious Protected Mode calls were evident. These are all buried in the routines _printf and _scanf. The basic C compiler template, however, is evident using _TEXT, _DATA, and DGROUP.

4.2.2 Accessing the API from C

The API is accessed in much the same fashion from C as it is from assembly language. Using the Toolkit definitions it is possible to set up the C function calls in a comfortable style for usage. Consider, for example, the prototyping for DosExit:

```
;       Static Name Aliases
;
        TITLE   ioprgm.c
        NAME    ioprgm

        .8087
_TEXT   SEGMENT  WORD 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
$$SYMBOLS       SEGMENT  BYTE PUBLIC 'DEBSYM'
$$SYMBOLS       ENDS
$$TYPES SEGMENT  BYTE PUBLIC 'DEBTYP'
$$TYPES ENDS
DGROUP  GROUP   CONST, _BSS, _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP
EXTRN   __acrtused:ABS
EXTRN   _printf:NEAR
EXTRN   _scanf:NEAR
EXTRN   __chkstk:NEAR
EXTRN   __fldw:NEAR
EXTRN   __fmuld:NEAR
EXTRN   __fstsp:NEAR
EXTRN   __flds:NEAR
EXTRN   __fstdp:NEAR
EXTRN   __fltused:NEAR
_DATA      SEGMENT
$SG159  DB      'Input word integer less than 32,768 ',  0aH,  00H
$SG160  DB      '%d',  00H
$SG165  DB      '2 times the integer value = %f',  0aH,  00H
_DATA      ENDS
_TEXT      SEGMENT
        ASSUME  CS: _TEXT
;|*** /* A simple C program to illustrate Protected Mode I/O -- ioprgm.c */
;|***
;|*** #include <stdio.h>
;|***
;|*** main()
;|***    {
; Line 6
        PUBLIC  _main
_main   PROC NEAR
        *** 000000      55                      push    bp
        *** 000001      8b ec                   mov     bp,sp
        *** 000003      b8 02 00                mov     ax,2
        *** 000006      e8 00 00                call    __chkstk
;       x = -2
;|***   int x;                                  /* input variable */
;|***
;|***   printf("Input word integer less than 32,768 \n");
; Line 9
        *** 000009      b8 00 00                mov     ax,OFFSET DGROUP:$SG159
        *** 00000c      50                      push    ax
        *** 00000d      e8 00 00                call    _printf
        *** 000010      83 c4 02                add     sp,2
;|***   scanf("%d",&x);
; Line 10
        *** 000013      8d 46 fe                lea     ax,WORD PTR [bp-2]
;x
        *** 000016      50                      push    ax
        *** 000017      b8 26 00                mov     ax,OFFSET DGROUP:$SG160
```

Figure 4.3 The >COD file for ioprgm.c.

```
        *** 00001a      50                          push    ax
        *** 00001b      e8 00 00                    call    _scanf
        *** 00001e      83 c4 04                    add     sp,4
;|***
;|***   times_2(x);                                 /* function */
; Line 12
        *** 000021      ff 76 fe                    push    WORD PTR [bp-2] ;x
        *** 000024      e8 00 00                    call    _times_2
        *** 000027      83 c4 02                    add     sp,2
;|***   }
; Line 13
        *** 00002a      8b e5                       mov     sp,bp
        *** 00002c      5d                          pop     bp
        *** 00002d      c3                          ret

_main   ENDP
;|***
;|*** times_2(y)
;|***   int y;                                      /* formal parameter */
; Line 16
_TEXT      ENDS
CONST      SEGMENT
$T20002 DQ      04000000000000000r    ; 2.000000000000000
CONST      ENDS
_TEXT      SEGMENT
        ASSUME  CS: _TEXT
        PUBLIC  _times_2
_times_2        PROC NEAR
        *** 00002e      55                          push    bp
        *** 00002f      8b ec                       mov     bp,sp
        *** 000031      b8 04 00                    mov     ax,4
        *** 000034      e8 00 00                    call    __chkstk
        *** 000037      56                          push    si
;|***   {
; Line 17
;       y = 4
;       z = -4
;|***   float z;                                    /* floating point */
;|***
;|***   z = (float)(2.*y);                          /* double */
; Line 20
        *** 000038      8d 5e 04                    lea     bx,WORD PTR [bp+4]
;y
        *** 00003b      e8 00 00                    call    __fldw
        *** 00003e      8d 1e 00 00                 lea     bx,WORD PTR $T20002
        *** 000042      e8 00 00                    call    __fmuld
        *** 000045      8d 5e fc                    lea     bx,WORD PTR [bp-4]
;z
        *** 000048      e8 00 00                    call    __fstsp
;|***
;|***   printf("2 times the integer value = %f\n",z);
; Line 22
        *** 00004b      8d 5e fc                    lea     bx,WORD PTR [bp-4]
;z
        *** 00004e      e8 00 00                    call    __flds
        *** 000051      83 ec 08                    sub     sp,8
        *** 000054      8b dc                       mov     bx,sp
        *** 000056      e8 00 00                    call    __fstdp
        *** 000059      b8 29 00                    mov     ax,OFFSET DGROUP:$SG165
        *** 00005c      50                          push    ax
        *** 00005d      e8 00 00                    call    _printf
        *** 000060      83 c4 0a                    add     sp,10
;|***   }; Line 23
        *** 000063      5e                          pop     si
        *** 000064      8b e5                       mov     sp,bp
        *** 000066      5d                          pop     bp
```

Figure 4.3 (*Continued*)

```
        *** 000067      c3                  ret

_times_2        ENDP
_TEXT   ENDS
END
```

Figure 4.3 *(Concluded)*

```
VOID APIENTRY DosExit(USHORT,USHORT)
```

Here the Toolkit definitions are

```
#define   VOID void
#define   APIENTRY pascal far
Typedef   unsigned int USHORT;
```

The definitions have their usual meaning in C, and the only one needing an explanation is the type pascal. Pascal refers to the calling convention used in accessing the function DosExit. When a C function is called the formal parameters are loaded on the stack, starting with the last parameter first. In the pascal convention the last parameter is loaded last. In the assembler instruction set, the PUSH instruction puts its operand on the stack and decrements the stack pointer. Hence, the initiating assembly language code sequence (at the beginning of each function)

```
...
push bp
mov bp,sp
sub sp,N
...
```

causes the previous module's (or calling module's) base pointer, bp, to be saved on the stack and the current stack pointer value, sp, placed in bp. Prior to execution of this code, the *calling program* caused the called function (procedure) parameters to be placed on the stack along with a return address (invoked at the CALL) for the C convention. The *called program* places the parameters on the stack along with a return address for the pascal convention. The "sub sp,N" instruction above simply reserves N bytes of stack for local usage (N should be even).

The API definitions include a large class of type definitions, such as PCHAR, for a pointer to a character where

```
typedef CHAR FAR *PCHAR;
```

Similarly,

```
typedef ULONG FAR *PULONG
```

defines a pointer to a LONG variable.

A particularly important definition, is one that allows the programmer to establish a FAR pointer:

```
#define MAKEP(sel,off) ((PVOID)MAKEULONG(off,sel))
```

where

```
typedef VOID FAR *PVOID;
```

and

```
#define MAKEULONG(l,h) ((ULONG)(((USHORT)(l)) |
((ULONG)((USHORT))(h)))<<16))
```

Access of the video context, for example, is via the following sequence:

```
...
VioSetMode (((struct_VIOMODEINFO far*)&CGAm),vio_hdl);
VioScrLock (wait2, (char far *)dstat1,vio_hdl);
...
VioScrUnLock(vio_hdl);
...
```

Here the structure _VIOMODEINFO is defined as CGAm and must satisfy the API constraints for structure members. It is defined in the Toolkit context as

```
typedef struct _VIOMODEINFO {
  USHORT cb;
  UCHAR fbType;
  UCHAR color;
  USHORT col;
  USHORT row;
  USHORT hres;
  USHORT vres;
  UCHAR fmt_ID;
  UCHAR attrib;
  } VIOMODEINFO;
```

Note that the structure above has the same form as the structure used in the assembly language programs except for the two added parameters, which are reserved and of no significance to the current programs. The video handle is short and declared with

```
SHANDLE vio_hdl = 0;
```

where

```
typedef unsigned short SHANDLE;
```

The remaining parameters are defined as would be expected in the usual C convention.

This, then, is how the API access is achieved based on the Toolkit definitions. The programmer can, of course, choose to develop his or her own definitions; however, it can be expected that they would be similar to those found in the Toolkit. Note that the function prototyping follows the assembler conventions established in the API library by IBM. The parameter setup in these calls is similar to that established for the basic assembler routines except that a pascal convention has been used.

4.2.3 Graphics Using C and OS/2

Figure 4.4 illustrates the MAKE file for a program swave.c, which plots a dynamically varying sine wave. Figure 4.5a is the flowchart for swave.c and Figure 4.5b the actual program code for this module. This code opens with the needed include file accesses and then sets up the keyboard buffer, an associated structure, several integer and character parameters, and the floating point arrays x[] and y[]. Next the principal calling routine, the function main(), is established.

```
                                                          swave.obj:
        cl -c -Zi -Os -FPc swave.c

gphrout.obj: gphrout.c
        cl -c -Zi -Os -FPc gphrout.c

swave.exe: swave.obj gphrout.obj
        link /CO swave.obj+gphrout.obj,swave,,slibce.lib/NOE os2.lib/NOE
```

Figure 4.4 MAKE file for swave.c.

In the beginning of main(), a number of structures are defined: the physical buffer (PVBPrt2) and two video mode structures (CGAm and STDm). These all follow the convention of the Toolkit and the OS/2 parameter definitions for the API calls [6]. The sine wave is to be iteratively displayed: Each iteration is incremented a finite time interval to simulate motion. The total number of iterations read is followed by the setup for the video CGA mode. The first VioSetMode parameter could have been specified as

```
(PVIOMODEINFO)&CGAm
```

instead of using the structure-oriented syntax. The screen is locked and the physical buffer accessed, using VioGetPhysBuf(). Note that this access allows the program to return a selector to the buffer, PVBPrt2.asel[0]. The call to sine_wave() includes specification of the number of iterations and the selector to the physical buffer. Once the return from sine_wave() takes place, the screen is unlocked and a keyboard hesitate implemented. This is followed by a reset of the mode to 80 x 25 (STDm) and the program exit.

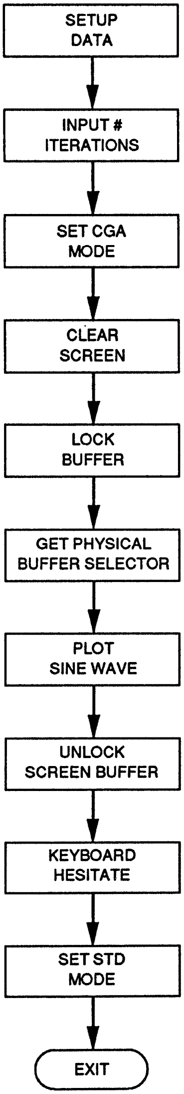


Figure 4.5a Flowchart for swave.c.

The sine_wave() program generates an array of 199 points of the sine wave of frequency 1 Hz and time interval of 0.02 second. This figure is plotted on the display and then removed [with upltpt()]. After each plot followed by removal, the time value is incremented by one-tenth of a second. The routine cclsCGA() clears the CGA screen. Note that it functions similarly to the earlier main() calls to lock, unlock, and get the screen buffer. This routine calls clrCGA(), which actually writes

```
/* This routine sets & clears CGA mode with screen clear--swave.c
 *  The generalized nomenclature is used.
 *  A dynamic sine wave has been added to the CGA mode output.
 *  This sine wave gradually moves across the screen
 *  The routine calls gphrout.c graphics functions. */

#define INCL_BASE                                  /* Conditional load */
#include <os2.h>
#include <math.h>

struct _STRINGINBUF lkbd_buf;                      /* keyboard buf len */
CHAR kbd_buf[80];                                  /* keyboard buffer */

UINT action = 0;                                   /* end thread */
UINT error_code = 0;                               /* result code */
UINT wait = 1;                                     /* reserved word */

CHAR dstat[1];                                     /* lock status */
CHAR dstat1[1];                                    /* lock status */

float x[250],y[250];                               /* screen coords */

main()
        {
        SHANDLE vio_hdl = 0;                       /* video handle */
        SHANDLE kbd_hdl = 0;                       /* keyboard handle */
        UINT wait2 = 1;                            /* reserved */
        UINT xb = 75,xe = 150,yb = 25,ye = 175;    /* box points */
        SEL MM1;                                   /* selector */

        int no_iter;                               /* number iteration */

        struct _VIOPHYSBUF PVBPrt2;                /* physical buffer */
        struct _VIOMODEINFO CGAm;                  /* CGA structure */
        struct _VIOMODEINFO STDm;                  /* 80 x 25 struct */

        PVBPrt2.pBuf = (BYTE far *)(0xB8000);      /* buffer start */
        PVBPrt2.cb = 0x4000;                       /* buffer size */

        CGAm.cb = 12;                              /* struct length */
        CGAm.fbType = 7;                           /* CGA mode */
        CGAm.color = 2;                            /* CGA color */
        CGAm.col = 40;                             /* text columns */
        CGAm.row = 25;                             /* text rows */
        CGAm.hres = 320;                           /* CGA hor res */
        CGAm.vres = 200;                           /* CGA vert res */

        STDm.cb = 12;                              /* struct length */
        STDm.fbType = 1;                           /* 80 x 25 mode */
        STDm.color = 4;                            /* STD color */
        STDm.col = 80;                             /* text columns */
        STDm.row = 25;                             /* text rows */
        STDm.hres = 720;                           /* STD hor res */
        STDm.vres = 400;                           /* STD vert res */

        lkbd_buf.cb = 80;                          /* buffer size */
        printf("Input number of iterations\n");
        scanf("%d",&no_iter);                      /* no. updates */

                                                   /* set CGA mode */
        VioSetMode(((struct _VIOMODEINFO far *)&CGAm),vio_hdl);

        cclsCGA(vio_hdl);                          /* clear CGA screen */

        VioScrLock(wait2,(char far *)dstat1,vio_hdl);  /* lock screen */
                                                   /* physical buffer */
```

Figure 4.5b The program swave.c, which plots a dynamic sine wave.

```
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt2,vio_hdl);

        MM1 = PVBPrt2.asel[0];                          /* selector */

        sine_wave(no_iter,MM1);                         /* sine wave */

        VioScrUnLock(vio_hdl);                          /* unlock screen */
                                                        /* hesitate screen */
        KbdStringIn((char far *)kbd_buf,
                ((struct _STRINGINBUF far *)&lkbd_buf),
                wait,kbd_hdl);
                                                        /* set STD mode */
        VioSetMode(((struct _VIOMODEINFO far *)&STDm),vio_hdl);

        DosExit(action,error_code);
        }

sine_wave(NN,MM1)
        int NN;
        SEL MM1;
        {
        float scale=35.,mid=100.;                       /* plot parameters */
        int mmid=100,zero=0,end=200,npts=199,n1,n;
        double PI = 3.141592654,t;

        t = 0.0;                                        /* start time */

        for(n1=1;n1 <= NN;n1++)                         /* loop screens */
           {
           for(n=1;n <= npts;n++)                       /* loop array pts */
              {
              y[n]=scale*(float)(sin(2.*PI*t));         /* sine wave */
              y[n]=mid-y[n];                            /* adjust plot */
              x[n]=(int)(n);                            /* col coordinate */
              t = t +.02;                               /* increment time */
              }

           for(n=1;n <= (npts-1);n++)
              pltpt(x[n],x[n+1],y[n],y[n+1],MM1);       /* plot points */

           for(n=1;n <= (npts-1);n++)
              upltpt(x[n],x[n+1],y[n],y[n+1],MM1);      /* unplot points */

           t=t+0.1;                                     /* major shift */
           }
        }

cclsCGA(vio_hdl1)
        SHANDLE vio_hdl1;
        {
        SEL MM;
        UINT wait1 = 1;
        struct _VIOPHYSBUF PVBPrt1;                     /* physical buffer */

        PVBPrt1.pBuf = (BYTE far *)(0xB8000);           /* phys buf start */
        PVBPrt1.cb = 0x4000;                            /* buffer length */

        VioScrLock(wait1,(char far *)dstat,vio_hdl1);   /* lock screen */
                                                        /* physical buffer */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt1,vio_hdl1);

        MM = PVBPrt1.asel[0];                           /* selector */

        clrCGA(MM);                                     /* CGA clear */
```

Figure 4.5b *(Continued)*

```
        VioScrUnLock(vio_hdl1);                        /* unlock screen */
        }

clrCGA(MM)
        SEL MM;
        {
        INT n;
        INT N1 = 0x1F3F;                               /* end odd buffer */
        INT DM = 0x2000;                               /* even offset */
        PCHAR ptr;                                     /* pointer scr buf */

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,n);                          /* odd far pointer */
           *ptr = 0;                                   /* clear odd buffer */
           }
        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,DM+n);                       /* even far pointer */
           *ptr = 0;                                   /* clear even buffer */
           }
        }
```

Figure 4.5b *(Concluded)*

a zero to each byte in the CGA buffer area. The function MAKEP() is used to generate the pointer to the physical buffer values, with the selector value passed from the earlier VioGetPhysBuf() call and the offset generated internally.

Figure 4.6 illustrates the graphics routines used in the calls to generate the sine wave graphics: wdot(), uwdot(), pltpt(), and upltpt(). In addition, a box drawing routine bboxx() and the vertical and horizontal line drawing routines are included.

4.2.4 Low-Level Access for Printer Graphics

Figure 4.7 illustrates the MAKE file for a program, prtwave.c, which plots a sine wave on the graphics printer (in this case an Epson FX-85 dot matrix printer). The MAKE file specifies three C source code files: prtwave.c, the actual sine wave generator; pprtscr.c, a C source code file that contains the code to drive the printer; and gphrout.c, the graphics routines specified in Figure 4.6.

Figure 4.8a illustrates the flowchart for prtwave.c and Figure 4.8b the corresponding source code for prtwave.c. This code is very similar to the program swave.c except that the printer routine arrays have been declared in the preprocessor area and a new sine wave routine, ssine_wave(), is called. This routine plots a single sine wave of 199 points at a selected frequency read in from main(). A call to prtscr() causes the screen to print on the printer.

Figure 4.9a illustrates the flowchart for the print-screen CGA mode routine, pprtscr.c. Figure 4.9b contains the code for this routine. Note that this routine performs exactly like its counterpart, prtscr.asm, in assembly language. The only difference is that an intermediate 16,000-byte buffer is not used. The same ESC character outputs are sent to the printer via DosWrite() once the printer is opened as a file device using DosOpen(). The data from the CGA screen is output as 25 blocks of 640 bytes (eight rows by 80 bytes per row). The routine ldarray is used to load

```
/* Graph routines Protected Mode--gphrout.c */

#define INCL_BASE
#include <os2.h>

bboxx(xb,xe,yb,ye,MM1)
        UINT xb,xe,yb,ye;
        SEL MM1;
        {
        lineh(yb,xb,xe,MM1);                          /* top line */
        lineh(ye,xb,xe,MM1);                          /* bottom line */
        linev(xb,yb,ye,MM1);                          /* right line */
        linev(xe,yb,ye,MM1);                          /* left line */
        }
lineh(y,x1,x2,MM1)
        UINT y,x1,x2;
        SEL MM1;
        {
        UINT n;
        for(n = x1;n <= x2;n++)
           wdot(n,y,MM1);                             /* hor line */
        }
linev(x,y1,y2,MM1)
        UINT x,y1,y2;
        SEL MM1;
        {
        UINT n;
        for(n = y1;n <= y2;n++)
           wdot(x,n,MM1);                             /* vertical line */
        }
wdot(x,y,MM1)
        UINT x,y;
        SEL MM1;                                      /* x=col,y=row */
        {
        PCHAR ptr;
        UINT DM = 0x0000;
        CHAR MASK1 = 0x01;                            /* set dot */

        if(y & 0x01)
           DM = 0x2000;                               /* even buffer */
        ptr = MAKEP(MM1,DM+(80*(y >> 1) + (x >> 2))); /* dot location */
        *ptr =(*ptr | (MASK1 << (2*(3 - x % 4))));    /* "OR" dot */
        }
uwdot(x,y,MM1)
        UINT x,y;
        SEL MM1;
        {
        PCHAR ptr;
        UINT DM = 0x0000;
        CHAR MASK1 = 0x00;                            /* clear dot */

        if(y & 0x01)
           DM = 0x2000;                               /* even buffer */
        ptr = MAKEP(MM1,DM+(80*(y >> 1) + (x >> 2))); /* dot location */
        *ptr = (MASK1 << (2*(3 - x % 4)));            /* write undot */
        }
pltpt(x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;                                      /* slope */
        int row;
        int col;

        if(x1 == x2)
           m = 1000.;                                 /* zero divide */
```

Figure 4.6 Graph routines used in the library cgraph.lib and taken from gphrout.c.

```
        else
           m = (y2-y1)/(x2-x1);                          /* normal slope */
        if( x2 > x1)
           {
           for(col = (int)(x1)+1;col <= (int)(x2);col++)
              {
              row =(int)(y1 + m*(col - x1));             /* line equation */
              wdot(col,row,MM1);                         /* write dot */
              }
           }
        else
           {
           if(x2 < x1)
              {
              for(col =(int)(x2)+1;col <= (int)(x1);col++)
                 {
                 row=(int)(y2 + m*(col - x2));
                 wdot(col,row,MM1);                      /* write dot */
                 }
              }
           else
              {
              col = (int)(x1);                           /* verticle line */
              if(y1 > y2)
                 {
                 for(row=(int)(y2)+1;row <= (int)(y1);row++)
                   wdot(col,row,MM1);
                 }
              else
                 {
                 for(row=(int)(y1)+1;row <= (int)(y2);row++)
                    wdot(col,row,MM1);
                 }
              }
           }
        }

upltpt(x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;                                         /* slope */
        int row;
        int col;

        if(x1 == x2)                                     /* zero divide */
           m = 1000.;
        else
           m = (y2-y1)/(x2-x1);                          /* normal slope */
        if(x2 > x1)
           {
           for(col = (int)(x1);col <= (int)(x2);col++)
              {
              row = (int)(y1 + m*(col - x1));            /* line segment */
              uwdot(col,row,MM1);                        /* erase dot */
              }
           }
        else
           {
           if(x2 < x1)
              {
              for(col = (int)(x2)+1;col <= (int)(x1);col++)
                 {
                 row=(int)(y2 + m*(col -x2));
                 uwdot(col,row,MM1);                     /* erase dot */
                 }
              }
           else
              {
              col = (int)(x1);
              if(y1 > y2)
                 {
                 for(row=(int)(y2)+1;row <= (int)(y1);row++)
                    uwdot(col,row,MM1);                  /* erase dot */
                 }
              else
                 {
                 for(row=(int)(y1)+1;row <= (int)(y2);row++)
                    uwdot(col,row,MM1);                  /* erase dot */
                 }
              }
           }
        }
```


Figure 4.6 (*Concluded*)

```
prtwave.obj: prtwave.c
        cl -c -Zi -Os -FPc prtwave.c

pprtscr.obj: pprtscr.c
        cl -c -Zi -Os -FPc pprtscr.c

gphrout.obj: gphrout.c
        cl -c -Zi -Os -FPc gphrout.c

prtwave.exe: prtwave.obj pprtscr.obj gphrout.obj
        link /CO prtwave.obj+pprtscr.obj+gphrout.obj,prtwave,,\
                slibce.lib/NOE os2.lib/NOE,
```

Figure 4.7 MAKE file for prtwave.c, which plots a sine wave on the printer.

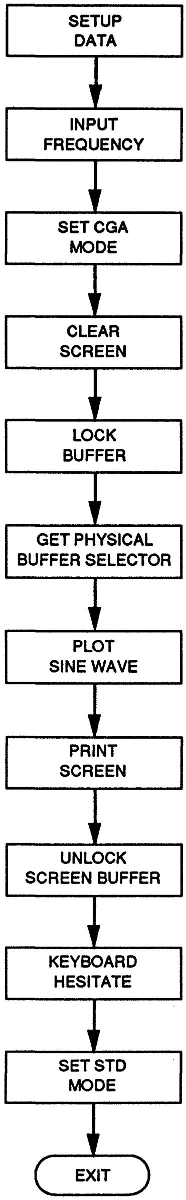


Figure 4.8a Flowchart for a program prtwave.c, which plots a sine wave on the printer.

```
/* This routine sets & clears CGA mode with screen clear--prtwave.c
*  The generalized nomenclature is used.
*  A sine wave has been added to the CGA mode output.
*  The routine calls gphrout.c graphics functions.
*  It prints the plot. */

#define INCL_BASE                                   /* Conditional load */
#include <os2.h>
#include <math.h>

struct _STRINGINBUF lkbd_buf;                       /* keyboard buf len */
CHAR kbd_buf[80];                                   /* keyboard buffer */

UINT action = 0;                                    /* end thread */
UINT error_code = 0;                                /* result code */
UINT wait = 1;                                      /* reserved word */

CHAR dstat[1];                                      /* lock status */
CHAR dstat1[1];                                     /* lock status */

float x[250],y[250];                                /* screen coords */
BYTE col1[320];                                     /* column array */
BYTE MM[4] = {0x40,0x10,0x04,0x01};                 /* mask */
BYTE w[8] = {128,64,32,16,8,4,2,1};                 /* weights */
BYTE s[4];                                          /* dummy */
BYTE shift1[4] = {6,4,2,0};                         /* shift count */
BYTE in_buffer1[4] = {0x1B,0x4B,64,1};              /* ESC K (320-256) */
BYTE in_buffer2[2] = {0x0D,0x0A};                   /* CR,LF */
BYTE in_buffer3[3] = {0x1B,0x41,8};                 /* ESC A 8/72 */
BYTE in_buffer4[2] = {0x1B,0x32};                   /* ESC 2 */
BYTE dev_name[5] = {'L','P','T','1',0};             /* device */

main()
        {
        extern prtscr();                            /* PrtSc routine */

        SHANDLE vio_hdl = 0;                        /* video handle */
        SHANDLE kbd_hdl = 0;                        /* keyboard handle */
        UINT wait2 = 1;                             /* reserved */
        UINT xb = 75,xe = 150,yb = 25,ye = 175;     /* box points */
        SEL MM1;                                    /* selector */

        float freq;                                 /* frequency */

        struct _VIOPHYSBUF PVBPrt2;                 /* physical buffer */
        struct _VIOMODEINFO CGAm;                   /* CGA structure */
        struct _VIOMODEINFO STDm;                   /* 80 x 25 struct */

        PVBPrt2.pBuf = (BYTE far *)(0xB8000);       /* buffer start */
        PVBPrt2.cb = 0x4000;                        /* buffer size */

        CGAm.cb = 12;                               /* struct length */
        CGAm.fbType = 7;                            /* CGA mode */
        CGAm.color = 2;                             /* CGA color */
        CGAm.col = 40;                              /* text columns */
        CGAm.row = 25;                              /* text rows */
        CGAm.hres = 320;                            /* CGA hor res */
        CGAm.vres = 200;                            /* CGA vert res */

        STDm.cb = 12;                               /* struct length */
        STDm.fbType = 1;                            /* 80 x 25 mode */
        STDm.color = 4;                             /* STD color */
        STDm.col = 80;                              /* text columns */
        STDm.row = 25;                              /* text rows */
        STDm.hres = 720;                            /* STD hor res */
        STDm.vres = 400;                            /* STD vert res */
```

Figure 4.8b The program prtwave.c.

```
        lkbd_buf.cb = 80;                                /* buffer size */

        printf("Input frequency (Hz)\n");
        scanf("%f",&freq);
                                                         /* set CGA mode */
        VioSetMode(((struct _VIOMODEINFO far *)&CGAm),vio_hdl);

        cclsCGA(vio_hdl);                                /* clear CGA screen */

        VioScrLock(wait2,(char far *)dstat1,vio_hdl);    /* lock screen */
                                                         /* physical buffer */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt2,vio_hdl);

        MM1 = PVBPrt2.asel[0];                           /* selector */

        ssine_wave(freq,MM1);                            /* sine wave */

        prtscr(MM1);                                     /* print screen */

        VioScrUnLock(vio_hdl);                           /* unlock screen */
                                                         /* hesitate screen */
        KbdStringIn((char far *)kbd_buf,
                   ((struct _STRINGINBUF far *)&lkbd_buf),
                   wait,kbd_hdl);
                                                         /* set STD mode */
        VioSetMode(((struct _VIOMODEINFO far *)&STDm),vio_hdl);

        DosExit(action,error_code);
        }

ssine_wave(freq,MM1)
        float freq;
        SEL MM1;
        {
        float scale=35.,mid=100.;                        /* plot parameters */
        int mmid=100,zero=0,end=200,npts=199,n1,n;
        double PI = 3.141592654,t;

        t = 0.0;                                         /* start time */
        for(n=1;n <= npts;n++)                           /* loop array pts */
           {
           y[n]=scale*(float)(sin(2.*PI*freq*t));        /* sine wave */
           y[n]=mid-y[n];                                /* adjust plot */
           x[n]=(int)(n);                                /* col coordinate */
           t = t +.02;                                   /* increment time */
           }
        for(n=1;n <= (npts-1);n++)
           pltpt(x[n],x[n+1],y[n],y[n+1],MM1);           /* plot points */

        }

cclsCGA(vio_hdl1)
        SHANDLE vio_hdl1;
        {
        SEL MM2;
        UINT wait1 = 1;
        struct _VIOPHYSBUF PVBPrt1;                      /* physical buffer */

        PVBPrt1.pBuf = (BYTE far *)(0xB8000);            /* phys buf start */
        PVBPrt1.cb = 0x4000;                             /* buffer length */

        VioScrLock(wait1,(char far *)dstat,vio_hdl1);    /* lock screen */
                                                         /* physical buffer */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt1,vio_hdl1);

        MM2 = PVBPrt1.asel[0];                           /* selector */
```

Figure 4.8b (*Continued*)

```
        clrCGA(MM2);                                  /* CGA clear */

        VioScrUnLock(vio_hdl1);                       /* unlock screen */
        }

clrCGA(MM3)
        SEL MM3;
        {
        INT n;
        INT N1 = 0x1F3F;                              /* end odd buffer */
        INT DM = 0x2000;                              /* even offset */
        PCHAR ptr;                                    /* pointer scr buf */

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM3,n);                        /* odd far pointer */
           *ptr = 0;                                  /* clear odd buffer */
           }
        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM3,DM+n);                     /* even far pointer */
           *ptr = 0;                                  /* clear even buffer */
           }
        }
```

Figure 4.8b *(Concluded)*

these eight rows in the following manner: For each of the 80 bytes in a row, the byte is unfolded into its four pixel values (to make the 320 pixels per CGA row). The single byte containing the four pixel values is stored in four locations: s[0], s[1], s[2], and s[3]. These values are in turn masked, shifted, and weighted according to their row position. A running total is generated across all rows for each pixel position and stored in col1[]. The col1[] values are returned to prtscr() (the pprtscr.c function that accomplishes the screen print). Each buffer row and column value is returned using

```
prt = MAKEP (MM1, DM + (80 *(row>>1) + (col>>2)));
```

Here MM1 is the selector value and DM = 0x2000 if the row value is odd. Figure 4.10 illustrates a typical sine wave plotted using this program.

4.3 MEMORY MANAGEMENT AND MULTITASKING WITH C

The API services offer a wide variety of multitasking and memory management options, as we have seen. There is an important constraint to consider when employing programs that run in a multitasking environment: The code is reentrant if differ-

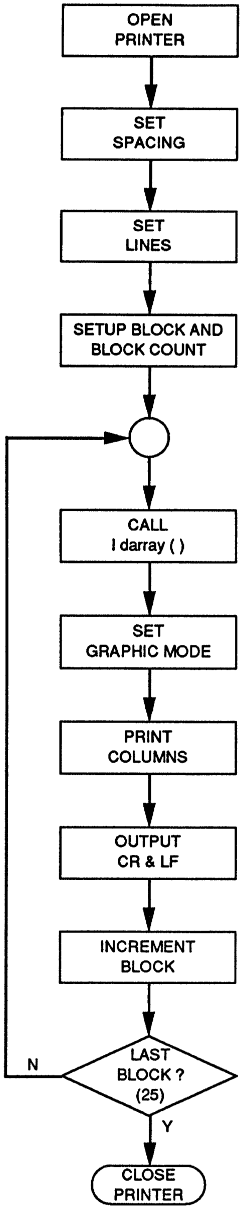


Figure 4.9a Flowchart for the program pprtscr.c, which performs the print operation in CGA mode.

```
/* This is a C print screen routine -- pprtscr.c */

#define INCL_BASE
#include <os2.h>

prtscr(MM1)
        SEL MM1;                                        /* selector */
        {
        /*      ---------------------------------
        *               Printer parameters
        *       --------------------------------- */

        extern BYTE in_buffer1[];                     /* ESC K (320-256) r */
        extern BYTE in_buffer2[];                       /* CR,LF */
        extern BYTE in_buffer3[];                       /* ESC A 8/72 */
        extern BYTE in_buffer4[];                       /* ESC 32 -- 1/6 */
                                                        /* byte counts */
        USHORT bytesin=320,bytesout=0,bytesin1=4,bytesin2=2,bytesin3=3;

        extern BYTE dev_name[];                         /* device name */
        HFILE dev_hand = 0;                             /* handle */
        USHORT dev_act = 0;                             /* action */
        ULONG dev_size = 0;                             /* size */
        USHORT dev_attr = 0;                            /* attribute */
        USHORT dev_flag = 1;                            /* open file */
        USHORT dev_mode = 0x00C1;                       /* private,nodeny */
        ULONG dev_rsv = 0;                              /* reserved */

        extern BYTE col1[];                             /* column array */

        UINT N;                                         /* block count */
        int blk_cnt,sixforty=640,n;

                                                        /* open printer */
        DosOpen(dev_name,(PHFILE)&dev_hand,(PUSHORT)&dev_act,dev_size,
                                dev_attr,dev_flag,dev_mode,dev_rsv);
                                                        /* set spacing */
        DosWrite(dev_hand,in_buffer3,bytesin3,(PUSHORT)&bytesout);
                                                        /* set lines */
        DosWrite(dev_hand,in_buffer4,bytesin2,(PUSHORT)&bytesout);

        blk_cnt=0;                                      /* 640 block */
        for(n=1;n <= 25;n++)
           {
           N = blk_cnt * sixforty;                      /* block bytes */

           ldarray(N,MM1);                              /* load array */
                                                        /* graphics mode */
           DosWrite(dev_hand,in_buffer1,bytesin1,(PUSHORT)&bytesout);
                                                        /* print columns */
           DosWrite(dev_hand,col1,bytesin,(PUSHORT)&bytesout);
                                                        /* CR,LF */
           DosWrite(dev_hand,in_buffer2,bytesin2,(PUSHORT)&bytesout);

           blk_cnt++;                                   /* inc block count */
           }
           DosClose(dev_hand);                             /* close printer */
           }

    ldarray(N,MM1)
           UINT N;
           SEL MM1;
           {
           extern BYTE MM[];                               /* mask */
           extern BYTE  w[];                               /* weights */
           extern BYTE  s[];                               /* dummy */
           extern BYTE shift1[];                           /* shift */
```

Figure 4.9b The routine pprtscr.c.

```
        int n,n1,m,N4,row,col;
        extern BYTE col1[];                              /* column array */

        N4 = N/80;                                       /* block row */
        for(n = 0;n <= 79;n++)
           {
           for(m = 0;m <= 3;m++)
              col1[n*4+m] = 0;                           /* initialize */
           for(n1 = 0;n1 <= 7;n1++)
              {
              row = N4 + n1;
              for(m = 0;m <= 3;m++)
                 {
                 col = n*4;                              /* nearest byte */
                 s[m] = rbuf(row,col,MM1);               /* screen byte */
                 }
              for(m = 0;m <= 3;m++)
                 {
                 s[m] = (s[m] & MM[m]);                  /* mask */
                 s[m] = (s[m] >> shift1[m]);             /* shift rt */
                 s[m] = s[m] * w[n1];                    /* weight */
                 col = n*4 + m;                          /* column index */
                 col1[col] = col1[col] + s[m];           /* column value */
                 }
              }
           }
        }
rbuf(y,x,MM1)
        SEL MM1;                                         /* selector */
        int x,y;                                         /* x=col,y=row */
        {
        PCHAR   ptr;                                     /* buffer ptr */
        UINT DM = 0x0000;                                /* even/odd */

        if(y & 0x01)
           DM = 0x2000;                                  /* odd row */

        ptr = MAKEP(MM1,DM + (80*(y >> 1)+(x >> 2)));    /* byte pointer */
        return(*ptr);                                    /* return value */
        }
```

Figure 4.9b (*Concluded*)

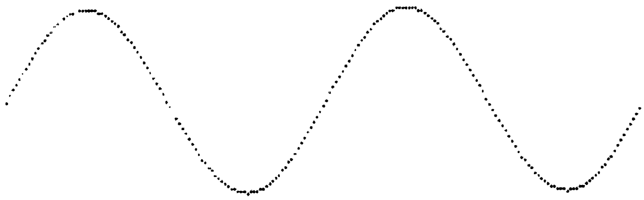


Figure 4.10 Representative sine wave output for the program prtwave.c.

ent threads call the same function. Hence in multithreaded applications the API services, which have code written for the multithreaded environment, are a more desirable way to write code than are the multithreaded versions of the standard C library routines. We always base our routines on the API calls.

In general, the variety of memory management services is large. These services include DosAllocSeg(), DosSubAllocSeg(), and DosAllocShrSeg(). In the next section we consider the creation of a shared segment. This is among the more complex variants of segment manipulation and constitutes a useful example. Similarly, the creation of a process or thread can be contingent on synchronization. For example, if both a parent and a child process access a shared segment, some form of synchronization is needed to ensure that one process does not write over the results of another before the data is properly used. In general we achieve synchronization using semaphores. This approach, using semaphores for synchronization and the API services for multitasking, allows easy development of both non-reentrant and reentrant code in the OS/2 multitasking environment. Memory management is, of course, a subset of this activity, particularly when sharing segments.

4.3.1 Creating and Accessing Segments

A good example of the use of memory management is the creation and access of shared segments. Figure 4.11a is a flowchart for the program pipestc.c. This program is the C version of the assembly language program, pipest.asm, which appears in Figure 3.22b. The program sets up a shared segment, creates a pipe, uses semaphores for synchronization, and passes a message to the pipe via the pipe's buffer area. A child process, pipeclc.c, then reads the pipe message and prints the message to the screen. The process is shown in Figure 4.11b, and the code for the child process appears in Figure 4.12. The MAKE file for pipestc.c is illustrated in Figure 4.13a, and the MAKE file for pipeclc.c is presented in Figure 4.13b.

The program pipestc.c opens with four string expressions of CHAR type. These expressions define the pointers msg_pd, prgm_nm, shrname, and aseml. Within the calling function, main(), a number of local variables have been defined using the standard OS/2 type casts. In the API service call to DosAllocShrSeg(), for example, the parameters are of the type

```
SEL msell;
CHAR FAR *shrname;
USHORT msize = 512;
```

where USHORT = unsigned short (16-bit word) and SEL denotes a selector: unsigned short SEL. The actual call in DosAllocShrSeg() specifies that the selector, msell, be specified as a pointer object:

```
(PSEL) &msell
```

Here the address of msell is treated as the pointer, as it should be.

When DosMakePipe() executes a read handle and write handle are returned for

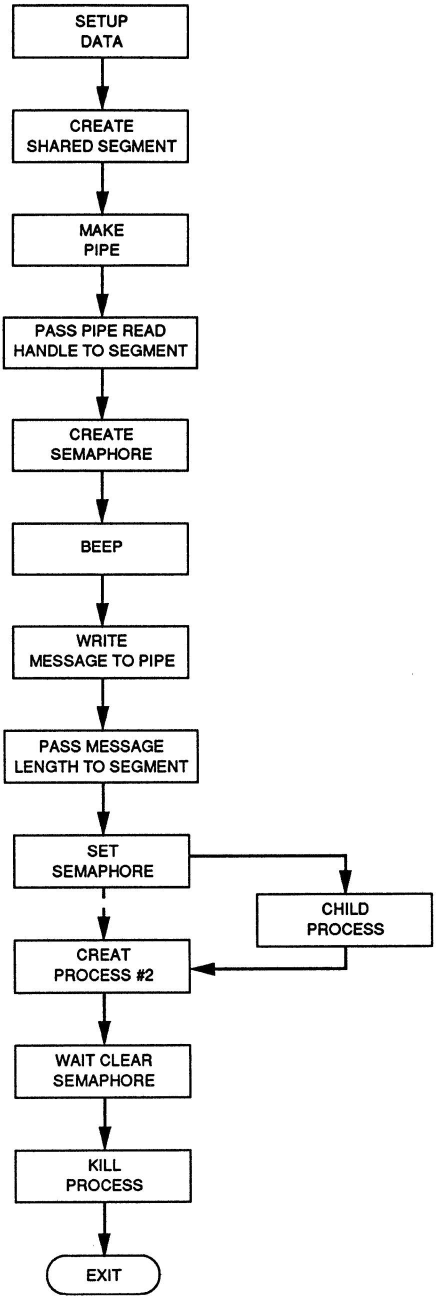


Figure 4.11a Flowchart for C program to emulate pipest.asm, a pipe and shared segment program.

```
/* Program to emulate pipest.asm -- pipestc.c
*  This routine sets up a child and accesses a shared
*  memory segment. */

#define INCL_BASE
#include <os2.h>
#include <string.h>

CHAR     *msg_p0  = "This is the OS/2 pipe message\n";
CHAR FAR *prgm_nm = "PIPECLC.EXE";
CHAR FAR *shrname = "\\SHAREMEM\\SDAT1.DAT";
CHAR FAR *aseml   = "\\SEM\\SDAT.DAT";
INT      blank[1] = {0x0007};                /* Scroll attribute */
size_t   lmsg_p0;                             /* length result */
main()
        {
/*-------------------------------------------------------------------------*/
/*                      Locally Defined Variables                          */
/*-------------------------------------------------------------------------*/

                                              /* Shared buffer */
        USHORT  msize = 512;                  /* Buffer size */
        SEL     msell;                        /* Selector */

                                              /* Pipe parameters */
        HFILE   read_hdl,write_hdl;           /* Pointer to pipe handles */
        USHORT  pflag = 256;                  /* Pipe size bytes */
        USHORT  bytes_written;                /* Length of write */

                                              /* Semaphore parameters */
        USHORT  no_excl = 1;                  /* No exclusive */
        HSEM    sem_hdl1;                     /* Semaphore handle */
        LONG    no_to = -1;                   /* No timeout */

                                              /* Beep */
        USHORT  freq = 5000;                  /* 5,000 Hertz */
        USHORT  duration = 500;               /* 500 millisec */

                                              /* Child process */
        CHAR    obj_nm_buf[40];               /* Failure buffer */
        USHORT  lobj_nm_buf = 40;             /* Length buffer */
        USHORT  async = 1;                    /* Child asynchronous */
        CHAR    argst = 0;                    /* NULL command parm */
        CHAR    envst = 0;                    /* NULL environment parm */
        RESULTCODES     PIDD;                 /* Structure-result codes */

        PUINT   ptr;                          /* Pointer */

        USHORT  action = 1;                   /* Terminate all threads */
        USHORT  result = 0;                   /* Completion code */

        USHORT  error2;                       /* Dummy error return */
/*-------------------------------------------------------------------------*/

        cls();                                /* Clear screen */

                                              /* Create shared segment */
        error2 = DosAllocShrSeg(msize,shrname,(PSEL)&msell);

                                              /* Check creation error */
        if(error2 != 0)
           {
           printf("Result code = %d",error2);
           exit(1);
           }
```

Figure 4.11b Program code for pipestc.c, illustrated in Figure 4.11a.

```
                                        /* Create pipe */
    DosMakePipe((PHFILE)&read_hdl,(PHFILE)&write_hdl,pflag);

    ptr = MAKEP(msel1,2);               /* Pointer to 2nd word */
    *ptr = read_hdl;                    /* Read handle */

                                        /* Create system sem */
    DosCreateSem(no_excl,(PHSEM)&sem_hdl1,(PCHAR)asem1);

    DosBeep(freq,duration);             /* Beep speaker */

    lmsg_p0 = strlen(msg_p0);           /* Length of message */

                                        /* Write to handle */
    DosWrite(write_hdl,(PVOID)msg_p0,lmsg_p0,
                 (PUSHORT)&bytes_written);

    ptr = MAKEP(msel1,4);               /* Pointer to 3rd word */
    *ptr = bytes_written;               /* Length of write */

    DosSemSet(sem_hdl1);                /* Set semaphore */

                                        /* Initiate child */
    error2 = DosExecPgm((PCHAR)obj_nm_buf,
                 lobj_nm_buf,async,
                 (PCHAR)&argst,
                 (PCHAR)&envst,
                 (PRESULTCODES)&PIDD,
                 prgm_nm);
                                        /* Check creation error */
    if(error2 != 0)
       {
       printf("Error on opening child");
       exit(1);
       }

    DosSemWait(sem_hdl1,no_to);         /* Wait on child */

                                        /* Terminate child */
    DosKillProcess(PIDD.codeTerminate,PIDD.codeResult);

    DosExit(action,result);             /* Terminate parent */
    }

cls()
    {
    USHORT  tr = 0;                     /* top row */
    USHORT  lc = 0;                     /* left column */
    USHORT  br = 23;                    /* bottom row */
    USHORT  rc = 79;                    /* right column */
    USHORT  no_line = 25;               /* no. lines */
    HVIO    vio_hdl;                    /* handle */

                                        /* Clear screen */
    VioScrollUp(tr,lc,br,rc,no_line,(PCHAR)blank,vio_hdl);
    }
```

Figure 4.11b (*Concluded*)

the pipe. The read handle, read_hdl, is written into the second word of the shared segment. A semaphore is created to synchronize the parent and child process. Essentially, we want the parent to wait on the child process until the child completes its write to the screen. The parent process (pipestc.c) writes the message to the pipe buffer using DosWrite(), and a return count of the length of the message written is placed in the shared segment buffer at offset 4. Next, the semaphore is set and the child process started. Once the child completes, the semaphore is cleared and the

```
/* This is the child process -- pipeclc.c
*  It writes the actual screen message using pipes */

#define INCL_BASE
#include <os2.h>

CHAR FAR   *asem1 = "\\SEM\\SDAT.DAT";
CHAR FAR *shrname = "\\SHAREMEM\\SDAT1.DAT";
CHAR    buffer[256];

main()
        {
/*      ---------------------------------------------------------------
*                       Local Variables
*       ---------------------------------------------------------------*/

                                        /* Exit parameters */
        USHORT  action = 1;             /* Terminates all threads */
        USHORT  result = 0;             /* Completion code */

        HVIO    vio_hdl;                /* Video handle */

                                        /* Beep */
        USHORT  freq = 4000;            /* 4,000 Hertz */
        USHORT  duration = 500;         /* 500 millisec */

        HSEM    sem_hdl1;               /* Semaphore handle */

        SEL     shrsel;                 /* Shared Segment Sel */

        HFILE   read_hdl;               /* Read handle */

        USHORT  lmsg;                   /* Length message */
        USHORT  bytes_read;             /* Bytes read */

        USHORT FAR *ptr;                /* 32-bit pointer */

/*      ---------------------------------------------------------------*/

                                        /* Open semaphore */
        DosOpenSem((PHSEM)&sem_hdl1,asem1);

        DosBeep(freq,duration);         /* Beep speaker */

        DosGetShrSeg(shrname,(PSEL)&shrsel);    /* Shared Segment */

        ptr = MAKEP(shrsel,2);          /* Pointer to 2nd word */
        read_hdl = *ptr;                /* Read handle */
        ptr = MAKEP(shrsel,4);          /* Pointer to 3rd word */
        lmsg = *ptr;                    /* Length of message */

                                        /* Read message */
        DosRead(read_hdl,buffer,lmsg,(PSHORT)&bytes_read);

        VioWrtTTy(buffer,lmsg,vio_hdl);         /* Write message */

        DosSemClear(sem_hdl1);          /* Clear semaphore */

        DosExit(action,result);         /* Terminate process */
        }
```

Figure 4.12 Program code for child process, pipeclc.c, used by pipestc.c.

```
        pipestc.obj: pipestc.c
        cl -c -Zi -Os -FPc pipestc.c

pipestc.exe: pipestc.obj
        link /CO pipestc.obj,pipestc,,\
                slibce.lib/NOE os2.lib/NOE,,
```

(a)

```
pipeclc.obj: pipeclc.c
        cl -c -Zi -Os -FPc pipeclc.c

pipeclc.exe: pipeclc.obj
        link /CO pipeclc.obj,pipeclc,,\
                slibce.lib/NOE os2.lib/NOE,,
```

(b)

Figure 4.13 (a) MAKE file for pipestc.c and (b) MAKE file for pipeclc.c.

wait state for the parent is terminated. The child process is terminated (along with any dependent processes), and the parent is completed with the termination DosExit(). The child process is illustrated in Figure 4.12.

4.3.2 Creating a Thread or Process

In Figure 4.11b the parent process initiates a child process specified using the pointer prgm_nm. This specification links the two processes and is the last parameter of DosExecPgm(). Figure 4.12 contains the program for the child. Note that the semaphore name and the shared segment name are the same as those specified by the parent. This constitutes the common link between the two processes. The speaker is beeped to indicate that the child is operating. The shared segment is retrieved and the pipe read handle and buffer length obtained. DosRead() is used to load buffer[] with the message in the pipe, and this message is printed using VioWvtTTY().

The return to the parent results in termination of the work semaphore. The child process is terminated by the parent, as well, with a full exit from the parent mode. Note that DosKillProcess() is not absolutely necessary because the child, pipeclc.exe, has been terminated but is good programming practice because it terminates all dependent processes started by the child, in addition to the child process itself, if needed. Figure 4.11b also illustrates the clear screen routine, which is a one-time call to VioScrollUp().

Figure 4.14 contains the MAKE file for a program ckthread.c, which creates a thread as opposed to a new process. Remember that a thread shares resources with

```
ckthred.obj: ckthred.c
        cl -c -Zi -Gs -FPc -F C00 -Lp ckthred.c

ckthred.exe: ckthred.obj
        link /CO ckthred.obj,ckthred,ckthred,slibce.lib/NOE os2.lib/NOE,,
```

Figure 4.14 MAKE file for ckthred.c. This program checks the formation of child threads.

the subordinate or child threads. This is unlike creation of a new process, where each subordinate process has its own encapsulated resources.

Since each thread has its own stack (a resource unique to the thread) a high-level-language compiler such as C will report stack overflow errors because the thread's unique stack space does not overlap within a given process. This usually generates an error message (in the Microsoft C Optimizing Compiler Version 5.1, for example). To avoid such error messages, a compiler option, -Gs, can be used to eliminate stack checking only when the stack space is known to be sufficient [7]. This will allow the compilation of programs that generate separate threads without generating an error condition. Each thread should allow a minimum stack size of 2048 bytes. In Figure 4.14 the compile and link allow generation of symbolic debugging information through options -Zi (compiler) and /CO (linker).

Figure 4.15 illustrates a simple OS/2 C language program which generates a thread that prints the message

```
"This is the subordinate thread"
```

to the display. A second message is printed following completion of the child thread action and return to the parent calling thread. The latter thread prints the message

```
"This is the main thread"
```

Both threads generate 500-millisecond tones. Synchronization is achieved using semaphores.

In the example program illustrated in Figure 4.15, no error checking exists following the API calls. This is because the program is fully debugged and the need for such diagnostics is minimized. During the debugging phase such diagnostics were included. Should a malfunction occur the user can, of course, set up such checking procedures. Note that in the call DosCreateThread(), the third parameter references byte 2047 in the stack as the stack start. This is because of the way the stack pointer is changed following a push to the stack. Element 2047 is the top of the stack in terms of address, and each push decrements the stack pointer to a lower address. Throughout this example the Toolkit nomenclature has been used. The type casting is in keeping with the Toolkit defined types and the actual API calls reflect the IBM Toolkit definitions for setup of the API functions.

```
                                                                  /* Thre
#define INCL_BASE
#include <os2.h>

#include <string.h>
#define SSIZE 2048

void FAR thread1(void);

CHAR *msg_p1 = "\n This is the main thread \n";
CHAR *msg_p2 = "This is the subordinate thread \n";
CHAR FAR *asem1 = "\\SEM\\SDAT.DAT";
CHAR stack1[SSIZE];                              /* Stack for thread1 */
HVIO vio_hdl;
HSEM sem_hdl1;

size_t lmsg_p1;
size_t lmsg_p2;

main()
        {
        USHORT  freq = 3000, duration = 500;
        USHORT  action = 1, result = 0, no_excl = 1;

        TID     threadID;                        /* Thread ID */
        LONG    no_to = -1;                      /* no timeout */


        DosBeep(freq,duration);                  /* Beep Speaker */
                                                 /* Create semaphore */
        DosCreateSem(no_excl,(PHSEM)&sem_hdl1,(PCHAR)asem1);

        DosSemSet(sem_hdl1);                     /* Set semaphore */

        DosCreateThread(thread1,(PTID)&threadID,(PBYTE)&stack1[2047]);

        DosSemWait(sem_hdl1,no_to);              /* Wait thread */

        lmsg_p1 = strlen(msg_p1);                /* Length Message 1 */

        VioWrtTTy(msg_p1,lmsg_p1,vio_hdl);

        DosExit(action,result);                  /* End main thread */
        }

void FAR thread1(void)
        {
        USHORT  freq1 = 5000, duration = 400;


        DosBeep(freq1,duration);                 /* Beep speaker */

        lmsg_p2 = strlen(msg_p2);                /* Length Message 2 */

        VioWrtTTy(msg_p2,lmsg_p2,vio_hdl);       /* Message 2 */

        DosSemClear(sem_hdl1);                   /* Clear semaphore */
        }
```

Figure 4.15 The program ckthred.c, which creates and exercises a child thread.

4.4 OTHER PROGRAMS

We have seen several examples of how to employ the API services in the C environment. These were beginning examples. In this section it will be useful to develop several more examples illustrating the lower-level nature of the API calls in this C environment. A great deal about the OS/2 implementation can be learned from these examples since frequently implementation features are obscured by the general syntax considerations.

4.4.1 A Rotating Tetrahedron

As a starting point, consider two-dimensional space represented by the usual Cartesian axes: x and y. Now consider the rotation of the point (x_1, y_1) to (x_2, y_2). Here, if r is the radius of the points from the origin

$$x_1 = r\cos(\alpha_1) \tag{4.1}$$

$$y_1 = r\sin(\alpha_1) \tag{4.2}$$

and

$$x_2 = r\cos(\alpha_2) \tag{4.3}$$

$$y_2 = r\sin(\alpha_1) \tag{4.4}$$

Writing

$$\alpha_2 = \alpha + \alpha_2 \tag{4.5}$$

we have

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} r\cos(\alpha + \alpha_1) \\ r\sin(\alpha + \alpha_1) \end{pmatrix} \tag{4.6}$$

which becomes

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1\cos(\alpha) - y_1\sin(\alpha) \\ x_1\sin(\alpha) + y_1\cos(\alpha) \end{pmatrix} \tag{4.7}$$

when the trigonometric identities are used for sine and cosine of the addition of two angles [4]. In matrix form

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_2\cos(\alpha) - \sin(\alpha) \\ y_2\sin(\alpha) + \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{4.8}$$

This rotation can be extended to three dimensions, where

α : rotation angle about the x-axis
β : rotation angle about the y-axis
γ : rotation angle about the z-axis

and we obtain the rotation matrices (A, B, and C, respectively) appearing in Table 4.1. Choosing an order to the rotation, we generate an overall three-dimensional rotation given by the matrix

$$R = CBA \tag{4.9}$$

This matrix is indicated in Table 4.1 and will serve as the basis for rotation of the tetrahedron. Note that the three rotations in Equation (4.9) are not orthogonal. We would need to select a different set of rotation angles to ensure orthogonality.

TABLE 4.1 ROTATION MATRICES FOR THREE-DIMENSIONAL MOVEMENT

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \quad B = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \quad C = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R = CBA = \begin{bmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta \end{bmatrix}$$

If we have a point on the tetrahedron given by (x, y, z), it is possible to define a rotated point based on R using

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = R \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.10}$$

In this example the tetrahedron appearing in Figure 4.16 will be used and rotated using R.

Figure 4.17 is the MAKE file for the rotating tetrahedron program. Note that pprtscr.c is not compiled in this MAKE file and we assume that an object module is available at link time. Figure 4.18a is the Structure Chart for this program. Figure 4.18b illustrates a flowchart for tetra.c, the main program module. Figure 4.19 contains the code for tetra.c. This module reads the three angular rates of rotation: α_0, β_0, and γ_0. Also read in is a scale factor for the size of the display image and the number of iterations to be rotated. Each iteration assumes an effective time increment of $dt = 0.05$ unit.

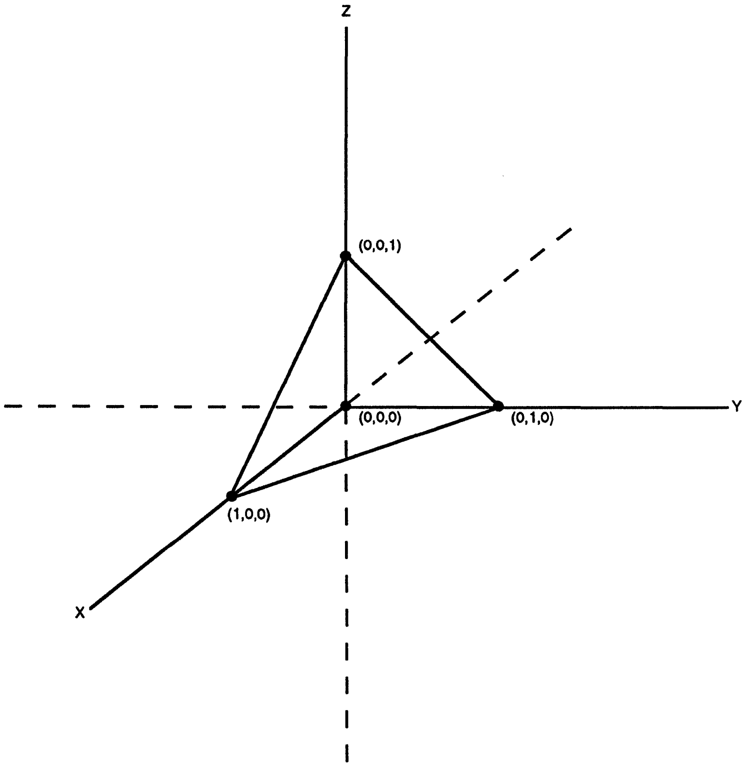


Figure 4.16 Three-dimensional starting representation for the tetrahedron.

```
tetra.obj: tetra.c
        cl -c -Zi -Gs -FPc -F C00 -Lp tetra.c

rotetra.obj: rotetra.c
        cl -c -Zi -Gs -FPc -F C00 -Lp rotetra.c

rotmat.obj: rotmat.c
        cl -c -Zi -Gs -FPc -F C00 -Lp rotmat.c

rotpt.obj: rotpt.c
        cl -c -Zi -Gs -FPc -F C00 -Lp rotpt.c

dmapoint.obj: dmapoint.c
        cl -c -Zi -Gs -FPc -F C00 -Lp dmapoint.c

udmapoin.obj: udmapoin.c
        cl -c -Zi -Gs -FPc -F C00 -Lp udmapoin.c

tetra.exe: tetra.obj rotetra.obj rotmat.obj rotpt.obj \
                dmapoint.obj udmapoin.obj pprtscr.obj cgraph.lib
        link tetra+rotetra+rotmat+rotpt+dmapoint+udmapoin+pprtscr,,,\
                slibce.lib/NOE os2.lib/NOE cgraph.lib/NOE,,
```

Figure 4.17 The MAKE file, tetra.mak, for the rotating tetrahedron.

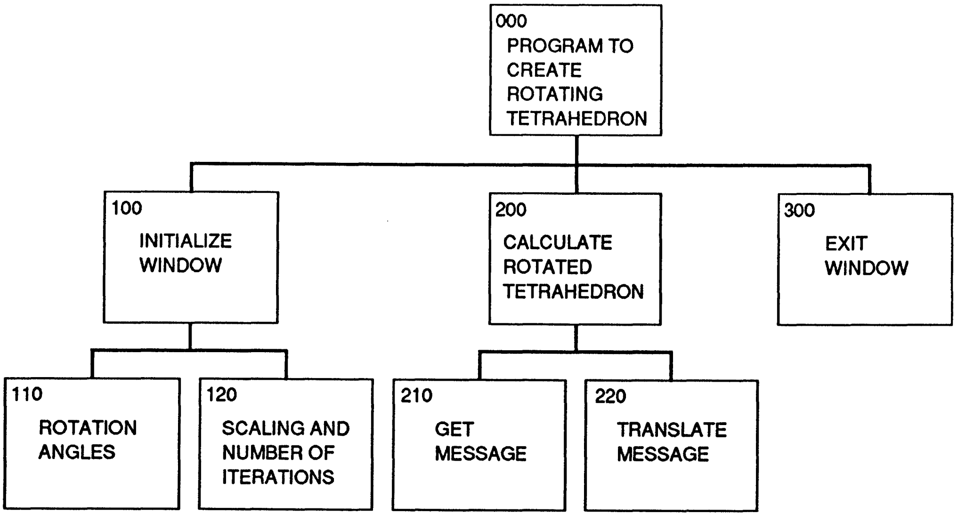


Figure 4.18a Structure Chart for the rotating tetrahedron program.

The module tetra.c contains the usual structures for VioSetMode(). After the display and physical buffer selectors are obtained, a call is made to r_tetra(), which sets up the new dynamic angle variables and calls rot_tetra(), which appears in Figure 4.20. Note that the arrays XX[], YY[], and ZZ[] are used to transfer the vertexes of the tetrahedron to rot_point(), where these points undergo the appropriate rotations. A call rot_mat() sets up the rotation matrix constants. The two functions DMA point() and uDMApoin() generate and remove the connecting lines, respectively. Figure 4.21 illustrates the calculation of the rotation matrix elements and should be compared with R in Table 4.1. Figure 4.22 merely completes Equation (4.10).

Figures 4.23 and 4.24 show the routines for accessing the physical screen buffer (DMApoint.c) and removing an existing dot on the screen (uDMApoin.c). They call wdot() and uwdot(), respectively, which are obtained from the library, cgraph.lib (see Figure 4.6).

Figures 4.25a and 4.25b illustrate the rotating tetrahedron after 100 iterations and 50 iterations, respectively. The input values for rates of rotation are $\alpha_0 = \beta_0 = \gamma_0 = 1$ and the scale is 60 units. These figures were obtained using prtscr().

4.4.2 Plotting Dow Jones Activity

The major purpose of this example is to illustrate disk access under OS/2. We need to recognize that some activities that are performed using the API calls can also be performed using the default C library (run time). Disk access is one of these activities.

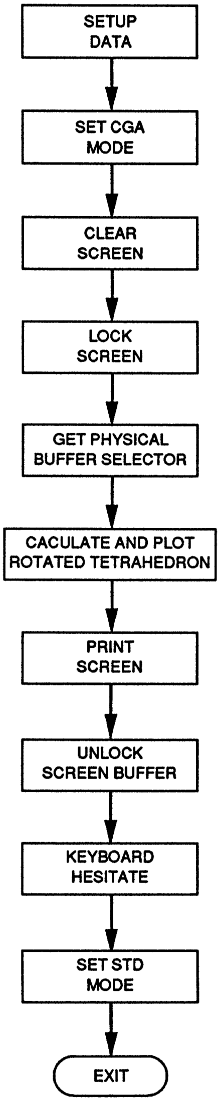


Figure 4.18b Flowchart for tetra.c, the rotating tetrahedron program.

```
                                                    /* This program gener
 *  The routine calls gphrout.c graphics functions. */

#define INCL_BASE                                  /* Conditional load */
#include <os2.h>
#include <stdio.h>

struct _STRINGINBUF lkbd_buf;                      /* keyboard buf len */
CHAR kbd_buf[80];                                  /* keyboard buffer */

UINT action = 0;                                   /* end thread */
UINT error_code = 0;                               /* result code */
UINT wait = 1;                                     /* reserved word */

CHAR dstat[1];                                     /* lock status */
CHAR dstat1[1];                                    /* lock status */

float XX[5] = {0.,1.,0.,0.,0.};                    /* Coords tetra */
float YY[5] = {0.,0.,1.,0.,0.};
float ZZ[5] = {0.,0.,0.,0.,1.};

float x,y,z;
float scale;
float a[10];
float xx1[5],yy1[5];
float xxx1[5],yyy1[5];
float dt = 0.05;
int NTOTAL;
float alpha,beta,gamma,alpha0,beta0,gamma0;

/*      ------------------------------------------------
 *                  Print Screen Parameters
 *      ------------------------------------------------        */
BYTE    col1[320];
BYTE    MM[4] = {0x40,0x10,0x04,0x01};
BYTE    w[8] = {128,64,32,16,8,4,2,1};
BYTE    s[4];
BYTE    shift1[4] = {6,4,2,0};
BYTE    in_buffer1[4] = {0x1B,0x4B,64,1};
BYTE    in_buffer2[2] = {0x0D,0x0A};
BYTE    in_buffer3[3] = {0x1B,0x41,8};
BYTE    in_buffer4[2] = {0x1B,0x32};
BYTE    dev_name[5] = {'L','P','T','1',0};
/*      ------------------------------------------------        */


main()
        {
        extern prtscr();                           /* PrtSc routine */

        SHANDLE vio_hdl = 0;                       /* video handle */
        SHANDLE kbd_hdl = 0;                       /* keyboard handle */
        UINT wait2 = 1;                            /* reserved */
        SEL MM1;                                   /* dummy selector */

        struct _VIOPHYSBUF PVBPrt2;                /* physical buffer */
        struct _VIOMODEINFO CGAm;                  /* CGA structure */
        struct _VIOMODEINFO STDm;                  /* 80 x 25 struct */

        PVBPrt2.pBuf = (BYTE far *)(0xB8000);      /* buffer start */
        PVBPrt2.cb = 0x4000;                       /* buffer size */

        CGAm.cb = 12;                              /* struct length */
        CGAm.fbType = 7;                           /* CGA mode */
        CGAm.color = 2;                            /* CGA color */
```

Figure 4.19 Program code for tetra.c, the main calling program for the rotating tetrahedron.

```
CGAm.col = 40;                                  /* text columns */
CGAm.row = 25;                                  /* text rows */
CGAm.hres = 320;                                /* CGA hor res */
CGAm.vres = 200;                                /* CGA vert res */

STDm.cb = 12;                                   /* struct length */
STDm.fbType = 1;                                /* 80 x 25 mode */
STDm.color = 4;                                 /* STD color */
STDm.col = 80;                                  /* text columns */
STDm.row = 25;                                  /* text rows */
STDm.hres = 720;                                /* STD hor res */
STDm.vres = 400;                                /* STD vert res */

lkbd_buf.cb = 80;                               /* buffer size */

printf("Input x-rotation rad/sec \n");
scanf("%f",&alpha0);
printf("Input y-rotation rad/sec \n");
scanf("%f",&beta0);
printf("Input z-rotation rad/sec \n");
scanf("%f",&gamma0);
printf("Input scale \n");
scanf("%f",&scale);
printf("Input total no. iterations \n");
scanf("%d",&NTOTAL);

                                                /* set CGA mode */
VioSetMode(((struct _VIOMODEINFO far *)&CGAm),vio_hdl);

cclsCGA(vio_hdl);                               /* clear CGA screen */

VioScrLock(wait2,(char far *)dstat1,vio_hdl);   /* lock screen */
                                                /* physical buffer */
VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt2,vio_hdl);

MM1 = PVBPrt2.asel[0];                          /* selector */

r_tetra(MM1);                                   /* tetrahedron */

prtscr(MM1);                                    /* print screen */

VioScrUnLock(vio_hdl);                          /* unlock screen */
                                                /* hesitate screen */
KbdStringIn((char far *)kbd_buf,
           ((struct _STRINGINBUF far *)&lkbd_buf),
           wait,kbd_hdl);
                                                /* set STD mode */
VioSetMode(((struct _VIOMODEINFO far *)&STDm),vio_hdl);
        DosExit(action,error_code);
        }

r_tetra(MM2)
        SEL MM2;
        {
        int n;

        alpha = 0.;                                     /* x-angle */
        beta = 0.,                                      /* y-angle */
        gamma = 0;                                      /* z-angle */

        for(n = 1;n <= NTOTAL;n++)
           {                                            /* dynamic angles */
           alpha = alpha + alpha0*dt;
           beta = beta + beta0*dt;
           gamma = gamma + gamma0*dt;

           rot_tetra(alpha,beta,gamma,n-1,MM2);         /* rotation */
```

Figure 4.19 (*Continued*)

```
            }
        }

cclsCGA(vio_hdl1)
        SHANDLE vio_hdl1;
        {
        SEL MM;
        UINT wait1 = 1;
        struct _VIOPHYSBUF PVBPrt1;                    /* physical buffer */

        PVBPrt1.pBuf = (BYTE far *)(0xB8000);          /* phys buf start */
        PVBPrt1.cb = 0x4000;                           /* buffer length */

        VioScrLock(wait1,(char far *)dstat,vio_hdl1);  /* lock screen */
                                                       /* physical buffer */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt1,vio_hdl1);

        MM = PVBPrt1.asel[0];                          /* selector */

        clrCGA(MM);                                    /* CGA clear */

        VioScrUnLock(vio_hdl1);                        /* unlock screen */
        }

clrCGA(MM)
        SEL MM;
        {
        INT n;
        INT N1 = 0x1F3F;                               /* end odd buffer */
        INT DM = 0x2000;                               /* even offset */
        PCHAR ptr;                                     /* pointer scr buf */

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,n);                          /* odd far pointer */
           *ptr = 0;                                   /* clear odd buffer */
           }
        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,DM+n);                       /* even far pointer */
           *ptr = 0;                                   /* clear even buffer */
           }
        }
```

Figure 4.19 *(Concluded)*

Figure 4.26 presents a routine timhist.c that runs in Protected Mode and functions identically to its Real Mode counterpart. The program generates a disk file that serves as a time-ordered database. The program accepts three values per record: a month, a year, and a tabulated value. The library functions, called in this code correspond to the standard C library functions, which are reentrant, hence can be used for Protected Mode calls. For the example to be illustrated in subsequent figures, this routine was used to create a database of monthly Dow Jones values between January 1988 and December 1988.

```
/* Function to rotate tetrahedron */

#define INCL_BASE
#include <os2.h>

rot_tetra(alpha1,beta1,gamma1,N,MM1)
        float alpha1,beta1,gamma1;                      /* angles */
        int N;
        SEL MM1;
        {
        extern float XX[],YY[],ZZ[];                    /* Tetra points */
        extern float x,y,z;                             /* point */
        extern float scale;                             /* scaling */
        extern float xx1[],yy1[];                       /* tetrahedron */
        int n;

        if(N > 0)
           {                                    /* Clear tetrahedron */
           uDMApoin(xx1[1],xx1[2],yy1[1],yy1[2],MM1);
           uDMApoin(xx1[1],xx1[3],yy1[1],yy1[3],MM1);
           uDMApoin(xx1[1],xx1[4],yy1[1],yy1[4],MM1);
           uDMApoin(xx1[2],xx1[3],yy1[2],yy1[3],MM1);
           uDMApoin(xx1[2],xx1[4],yy1[2],yy1[4],MM1);
           uDMApoin(xx1[3],xx1[4],yy1[3],yy1[4],MM1);
           }

        rot_mat(alpha1,beta1,gamma1);                   /* load rotate */

        for(n = 1;n <= 4;n++)
           {
           x = XX[n];
           y = YY[n];
           z = ZZ[n];
           rot_point();
           xx1[n] = x*scale + 150.;                     /* x-projection */
           yy1[n] = y*scale + 100;                      /* y-projection */
           }
                                                /* Rotate tetrahedron */
        DMApoint(xx1[1],xx1[2],yy1[1],yy1[2],MM1);
        DMApoint(xx1[1],xx1[3],yy1[1],yy1[3],MM1);
        DMApoint(xx1[1],xx1[4],yy1[1],yy1[4],MM1);
        DMApoint(xx1[2],xx1[3],yy1[2],yy1[3],MM1);
        DMApoint(xx1[2],xx1[4],yy1[2],yy1[4],MM1);
        DMApoint(xx1[3],xx1[4],yy1[3],yy1[4],MM1);
        }
```

Figure 4.20 The routine rotetra.c, which sets up the tetrahedron and calls the rotation matrices.

Figure 4.27 contains the MAKE file for dja.c, which reads the database created and generates a plot of the activity. In this case, this activity corresponds to the Dow Jones performance. Figure 4.28 presents the actual program code for dja.c. Note the needed API calls to clear the screen, plot the graphics, and print the screen. Figure 4.29 is the prtscr() output for this Dow Jones time history. The curve consists of monthly values rounded to the nearest five points.

The use of fprintf() constitutes the standard I/O call for the disk write and works as well in Protected Mode as in Real Mode. IBM and Microsoft have maintained this standard I/O interface within the constraints of OS/2.

```
/* Function to calculate rotation matrix */

#include <math.h>

rot_mat(alpha,beta,gamma)
        float alpha,beta,gamma;                     /* angles */
        {
        extern float a[];                           /* rotation matrix */
        double a1,CA,CB,CG,SA,SB,SG;

        a1 = (double)(alpha);                       /* Sines & cosines */
        CA = cos(a1);
        SA = sin(a1);
        a1 = (double)(beta);
        CB = cos(a1);
        SB = sin(a1);
        a1 = (double)(gamma);
        CG = cos(a1);
        SG = sin(a1);

        a[1] = (float)(CB*CG);                      /* Matrix elements */
        a[2] = (float)(SA*SB*CG - CA*SG);
        a[3] = (float)(CA*SB*CG + SA*SG);
        a[4] = (float)(CB*SG);
        a[5] = (float)(SA*SB*SG + CA*CG);
        a[6] = (float)(CA*SB*SG - SA*CG);
        a[7] = (float)(-SB);
        a[8] = (float)(SA*CB);
        a[9] = (float)(CA*CB);
        }
```

Figure 4.21 The program rotmat.c, which calculates the rotation matrix.

```
/* Function to generate rotated point */

rot_point()
        {
        extern float x,y,z;                         /* point */
        extern float a[];                           /* rotation matrix */
        float x1,y1,z1;                             /* intermediate */

        x1 = a[1]*x + a[2]*y + a[3]*z;
        y1 = a[4]*x + a[5]*y + a[6]*z;
        z1 = a[7]*x + a[8]*y + a[9]*z;

        x = x1;
        y = y1;
        z = z1;
        }
```

Figure 4.22 The program rotpt.c, which rotates a point.

```
/* This routine plots a connecting line using DMA */

#define INCL_BASE
#include <os2.h>

DMApoint (x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;
        int row;
        int col;

        if (x1 == x2)
                m = 1000;                       /*Upper limit on slope*/
        else
                m = (y2 - y1)/(x2 - x1);        /* normal slope */
        if(x2 > x1)
           {
           for (col =(int)(x1)+1; col <= (int)(x2); col++)
                {                               /* y-axis behavior */
                row = (int)(y1 + m*(col - x1));
                wdot(col,row,MM1);              /* write dot */
                }
           }
        else
           {
           if(x2 < x1)
              {
              for(col =(int)(x2)+1;col <= (int)(x1); col++)
                 {                              /* y-axis behavior */
                 row = (int)(y2 + m*(col - x2));
                 wdot(col,row,MM1);             /* write dot */
                 }
              }
           else
              {
              col = (int)(x1);                          /* Vertical line */
              if(y1 > y2)
                 {
                 for(row = (int)(y2)+1;row <= (int)(y1);row++)
                    wdot(col,row,MM1);                  /* write dot */
                 }
              else
                 {
                 for(row = (int)(y1)+1;row <= (int)(y2);row++)
                    wdot(col,row,MM1);                  /* write dot */
                 }
              }
           }
        }
```

Figure 4.23 The program DMApoint.c, which writes a point on the display using DMA.

```
/* This routine removes a connecting line using DMA */

#define INC_BASE
#include <os2.h>

uDMApoin(x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;
        int row;
        int col;

        if (x1 == x2)
                m = 1000;                       /*Upper limit on slope*/
        else
                m = (y2 - y1)/(x2 - x1);        /* normal slope */
        if(x2 > x1)
           {
           for (col =(int)(x1)+1; col <= (int)(x2); col++)
                {
                row = (int)(y1 + m*(col - x1));
                uwdot(col,row,MM1);             /* erase dot */
                }
           }
        else
           {
           if(x2 < x1)
              {
              for(col =(int)(x2)+1;col <= (int)(x1); col++)
                 {
                 row = (int)(y2 + m*(col - x2));
                 uwdot(col,row,MM1);            /* erase dot */
                 }
              }
           else
              {
              col = (int)(x1);                          /* Vertical line */
              if(y1 > y2)
                 {
                 for(row = (int)(y2)+1;row <= (int)(y1);row++)
                    uwdot(col,row,MM1);                 /* erase dot */
                 }
              else
                 {
                 for(row = (int)(y1)+1;row <= (int)(y2);row++)
                    uwdot(col,row,MM1);                 /* erase dot */
                 }
              }
           }
        }
```

Figure 4.24 The program uDMApoin.c, which removes a point from the display using DMA.

(a) (b)

Figure 4.25 (a) The tetrahedron after 100 iterations with and scale = 60. (b) The tetrahedron after 50 iterations with and scale = 60.

```
/* Routine to create time-history/value database -- timhist.c*/

#include <stdio.h>                              /* I/O file */

int month[288],year[288];                       /* time arrays */
float value[288];                               /* quant. interest */
char FN1[81];                                   /* filename array */

main()
        {
        int n,counter,check;                    /* integer var. */
        FILE *outfile;                          /* stream pointer */

        printf("Input database filename \n");
        gets(FN1);                              /* library routine */
        n = 1;                                  /* initialize index */
        month[0] = 1;                           /* init month not 0 */
        while(month[n-1] != 0)
           {
           printf("Input month as int (0 terminates)\n");
           scanf("%d",&month[n]);
           if(month[n] != 0)
              {
              printf("Input year as 2-digit int\n");
              scanf("%d",&year[n]);
              printf("Input value - floating point\n");
              scanf("%f",&value[n]);
              }
           n++;                                 /* increment index */
           if(n > 288)
              exit(1);                          /* overflow mem */
           }
        counter = n - 2;                        /* fix count */

        if((outfile = fopen(FN1,"w")) == NULL)  /* open out file */
           {
           printf("Output file failure: %s",FN1);
           exit(1);
           }
        fprintf(outfile,"%d ",counter);         /* output count */
        for(n = 1;n <= counter;n++)
           fprintf(outfile,"%d %d %f ",month[n],year[n],value[n]);

        if((check = fclose(outfile)) != 0)      /* close file */
           {
           printf("Error in output file close");
           exit(1);
           }
        }
```

Figure 4.26 The routine timhist.c, which creates a data file consisting of dates and values.

```
dja.obj: dja.c
        cl -c -Zi -Gs -FPc -F C00 -Lp dja.c

dja.exe: dja.obj pprtscr.obj cgraph.lib
        link dja+pprtscr,,,\
                slibce.lib/NOE os2.lib/NOE cgraph.lib/NOE,,
```

Figure 4.27 The MAKE file for dja.c, which plots the Dow Jones activity generated by timhist.c.

```
                                                  /* Routine to read Do

#define INCL_BASE
#include <stdio.h>
#include <os2.h>
#include <math.h>
#include <stdlib.h>

int month[288],year[288];                         /* time arrays */
float value[288];                                 /* quantity */
char FN1[81];                                     /* filename array */
float xx[1024];                                   /* Scratch buffers */
char buffer[90],buffer1[90];

struct _STRINGINBUF lkbd_buf;                     /* keyboard buf len */
CHAR kbd_buf[80];                                 /* keyboard buffer */

UINT action = 0;                                  /* end thread */
UINT error_code = 0;                              /* result code */
UINT wait = 1;                                    /* reserved word */

CHAR dstat[1];                                    /* lock status */
CHAR dstat1[1];                                   /* lock status */

/*      --------------------------------------------------
 *                  Print Screen Parameters
 *      --------------------------------------------------  */

BYTE    col1[320];
BYTE    MM[4] = {0x40,0x10,0x04,0x01};
BYTE    w[8] = {128,64,32,16,8,4,2,1};
BYTE    s[4];
BYTE    shift1[4] = {6,4,2,0};
BYTE    in_buffer1[4] = {0x1B,0x4B,64,1};
BYTE    in_buffer2[2] = {0x0D,0x0A};
BYTE    in_buffer3[3] = {0x1B,0x41,8};
BYTE    in_buffer4[2] = {0x1B,0x32};
BYTE    dev_name[5] = {'L','P','T','1',0};

/*      --------------------------------------------------   */

main()
        {
        int n,counter,check,N,i,delta,nmaxs,nmins;    /* integer var. */
        FILE *infile;                                 /* stream pointer */
        double x,y,z;
        float maxt,mint,b,b1;

        extern prtscr();                              /* print screen */

        SHANDLE vio_hdl = 0;                          /* video handle */
        SHANDLE kbd_hdl = 0;                          /* keyboard handle */
        UINT wait2 = 1;                               /* reserved */
        SEL MM1;                                      /* dummy selector */

        struct _VIOPHYSBUF PVBPrt2;                   /* physical buffer */
        struct _VIOMODEINFO CGAm;                     /* CGA structure */
        struct _VIOMODEINFO STDm;                     /* 80 x 25 struct */

        PVBPrt2.pBuf = (BYTE far *)(0xB8000);         /* buffer start */
        PVBPrt2.cb = 0x4000;                          /* buffer size */

        CGAm.cb = 12;                                 /* struct length */
        CGAm.fbType = 7;                              /* CGA mode */
        CGAm.color = 2;                               /* CGA color */
        CGAm.col = 40;                                /* text columns */
        CGAm.row = 25;                                /* text rows */
```

Figure 4.28 The program dja.c, which plots the Dow Jones activity.

```
CGAm.hres = 320;                              /* CGA hor res */
CGAm.vres = 200;                              /* CGA vert res */

STDm.cb = 12;                                 /* struct length */
STDm.fbType = 1;                              /* 80 x 25 mode */
STDm.color = 4;                               /* STD color */
STDm.col = 80;                                /* text columns */
STDm.row = 25;                                /* text rows */
STDm.hres = 720;                              /* STD hor res */
STDm.vres = 400;                              /* STD vert res */

lkbd_buf.cb = 80;                             /* buffer size */


printf("Input database filename\n");
gets(FN1);                                    /* library routine */
if((infile = fopen(FN1,"r")) == NULL)
   {
   printf("Input file failure: %s",FN1);
   exit(1);
   }
fscanf(infile,"%d ",&counter);                /* no. records */
for(n = 1;n <= counter;n++)
   {
   fscanf(infile,"%d %d %f ",&month[n],&year[n],&value[n]);
   printf("%5d %5d %6.0f \n",month[n],year[n],value[n]);
   }
if((check = fclose(infile)) != 0)
   {
   printf("Error in input file close");
   exit(1);
   }

mint = 1.e4;                                  /* reverse limit */
maxt = 0.0;                                   /* reverse limit */

N = counter;
for(i = 1;i <= N;i++)                         /* max/min */
   {
   if(maxt < value[i])
      maxt = value[i];
   if(mint > value[i])
      mint = value[i];
   }
                                              /* Set scale */
delta = maxt - mint;
delta = delta/10;
if(delta < 1)
   delta = 1;
if(delta > 1 && delta < 5)
   delta = 5;
if(delta > 5 && delta < 10)
   delta = 10;
if(delta > 10 && delta < 50)
   delta = 50;
if(delta > 50 && delta < 100)
   delta = 100;
if(delta > 100 && delta < 500)
   delta = 500;
if(delta > 500)
   {
   printf(" delta > 500");
   exit(1);
   }
nmaxs = maxt/delta + 1;
nmins = mint/delta;
```

Figure 4.28 (*Continued*)

```
        maxt = delta*nmaxs;
        mint = delta*nmins;
        if(mint <= 0)
           mint = mint - delta;                       /* scaled min */

        x = mint;                                     /* scaled max */
        y = fabs(x);
        z = (float)(y);
        bl = (z)/((float)(x));
        if(maxt > z && bl < 0)
           mint = -maxt;
        else
           {
           if(maxt < z && bl < 0)
              maxt = z;
           }

        b = 150./(maxt - mint);                       /* plot coords */
        for(i = 1;i <= N;i++)
           {
           value[i] = 25. + (150. - b*(value[i] - mint));
           xx[i] = 25. + (i - 1)*(256./(float)(N));
           }

                                                      /* CGA mode */
        VioSetMode(((struct _VIOMODEINFO far*)&CGAm),vio_hdl);

        cclsCGA(vio_hdl);                             /* clear screen */

        VioScrLock(wait2,(char far *)dstat1,vio_hdl);  /* lock buffer */

                                                      /* get physical buf */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt2,vio_hdl);

        MM1 = PVBPrt2.asel[0];                        /* selector */

        box_norm(MM1);                                /* plot box */

                                                      /* plot points */
        for (n=1;n<=(N-1);n++)
           pltpt(xx[n],xx[n+1],value[n],value[n+1],MM1);

        prtscr(MM1);                                  /* print screen */

        VioScrUnLock(vio_hdl);                        /* unlock buffer */

        KbdStringIn((char far *)kbd_buf,              /* hesitate */
                ((struct _STRINGINBUF far *)&lkbd_buf),
                wait,kbd_hdl);

                                                      /* STD mode */
        VioSetMode(((struct _VIOMODEINFO far *)&STDm),vio_hdl);

        DosExit(action,error_code);
        }

box_norm(SEL MM)
        {
        int xxbeg,xxend,yybeg,yyend;

        xxbeg = 25;                                   /* box parameters */
        xxend = 281;
        yybeg = 25;
        yyend = 175;

        bboxx(xxbeg,xxend,yybeg,yyend,MM);            /* draw box */
        }
```

Figure 4.28 (*Continued*)

```
cclsCGA(SHANDLE vio_hdl1)
        {
        SEL MM;
        UINT wait1 = 1;
        struct _VIOPHYSBUF PVBPrt1;                        /* physical buffer */

        PVBPrt1.pBuf = (BYTE far *)(0xB8000);              /* phys buf start */
        PVBPrt1.cb = 0x4000;                               /* buffer length */

        VioScrLock(wait1,(char far *)dstat,vio_hdl1);      /* lock screen */

        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt1,vio_hdl1);

        MM = PVBPrt1.asel[0];                              /* selector */

        clrCGA(MM);                                        /* CGA clear */

        VioScrUnLock(vio_hdl1);                            /* unlock buffer */
        }

clrCGA(SEL MM)
        {
        int n;
        int N1 = 0x1F3F;                                   /* end odd buffer */
        int DM = 0x2000;                                   /* even offset */
        PCHAR ptr;                                         /* pointer scr buf */

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,n);                              /* odd far pointer */
           *ptr = 0;                                       /* clear odd buffer */
           }

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MM,DM+n);                           /* even far pointer */
           *ptr = 0;                                       /* clear even buffer */
           }
        }
```

Figure 4.28 (*Concluded*)

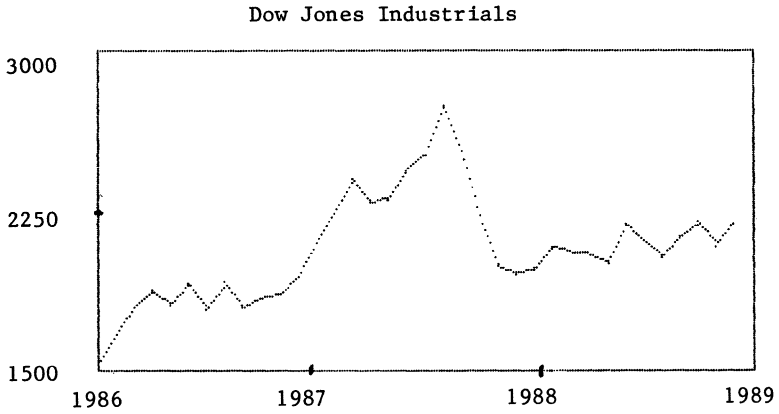


Figure 4.29 Annotated plot of Dow Jones activity through the October 1987 crash.

4.5 SUMMARY

This chapter has served to illustrate the OS/2 and C programming language interface. C represents a higher degree of abstraction than does assembler. The IBM Toolkit provides a complete set of types and API function prototypes to permit high-level access to the API services. The low-level nature of these services is illustrated in the C context.

Graphics under C are developed and a new print screen routine in C is presented that allows a screen dump in graphics mode. This screen dump program is designed for operation with CGA mode. The standard C syntax is presented and a knowledge of this syntax is assumed as a prerequisite to understanding the chapter.

The creation of multitasking routines is developed using multiple threads and processes. Finally, the issues associated with programming multitasked routines in C are developed through program examples. This chapter is necessarily introductory and establishes a basis for programming OS/2 applications using the C language.

REFERENCES

1. *Microsoft Optimizing C Compiler Version 5.1*, Microsoft Corporation, Redmond, WA, 1988.
2. *IBM C/2*, Language Reference and Fundamentals, International Business Machines Corporation, Boca Raton, FL, 1987.
3. *Operating System/2 Programmer's Toolkit Version 1.0 and 1.1*, Programmer's Guide, International Business Machines Corporation, Boca Raton, FL, 1987.
4. Godfrey, J. T., *Applied C: The IBM Microcomputers*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1990.
5. Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1988.
6. *Operating System/2 Technical Reference*, Vols. 1 and 2, International Business Machines Corporation, Boca Raton, FL, 1988.
7. Schildt, H., *OS/2 Programming: An Introduction*, Osborne McGraw-Hill, Berkeley, CA, 1988, p. 174.

PROBLEMS

4.1 When using the IBM Toolkit definitions, what precaution must be used between Versions 1.0 and 1.1?

4.2 What are the implications for the standard C I/O library running under the Protected Mode of OS/2?

4.3 The Version 5.1 of the Microsoft C Optimizing Compiler passes formal parameters with type specifications for these parameters appearing in the function definition. Typically, a function definition such as

```
INT box_norm(waitt, float x, float y)
...
```

replaces

```
INT box_norm(waitt, x, y)
INT waitt;
float x,y;
...
```

The type specification moves within the formal parameter list itself. What characteristic of this type definition remains consistent across formal parameter types?

4.4 In Figure 4.1, why must the input word integer be less than 32,768?

4.5 The OS/2 references dictate the manner in which the API services must be loaded on the stack. Using the normal C calling convention, what parameter access on the local stack exists? How does the Toolkit modify the API calls to load the local stack correctly for access?

4.6 What does the syntax

```
CHAR FAR *ptr;
```

mean? Why is the following acceptable?

```
CHAR FAR *shrname = "\\SHAREMEM\\SDAT1.DAT";
```

4.7 What does the function MAKEP (sel,off) accomplish?

4.8 In Figure 4.5b, what is the significance of the following?

```
PVBPrt2.pBuf = (BYTE far *)(0xB8000);
```

4.9 What is wrong with the code

```
...
float y;
double t;
...
y = sin(2.* PI *t);
```

Assume that PI is defined as

```
#define PI 3.141592654
```

4.10 What differentiates uwdot() from wdot()? (See Figure 4.6.)

4.11 In pltpt() and upltpt() (Figure 4.6) the x-values are associated with column values and the y-values are associated with row values. Explain.

4.12 In setting up the printer for a dump of the display, the following command is used (Figure 4.9b):

```
...
DosWrite(dev_hand,in_buffer,bytesin1,(PUSHORT)&bytesort);
...
```

Here bytesin1 = 4 and

```
BYTE in_buffer1[4] = {0x1B,0x4B,64,1};
```

What is the significance of this output value?

4.13 What is the meaning of the DosWrite() execution in Figure 4.9b with in_buffer3[] defined as

```
BYTE in_buffer3[3] = {0x1B,0x41,8};
```

4.14 In Figure 4.11b, are the semaphores used RAM semaphores or system semaphores?

4.15 What is characteristic about all the API function calls and their associated formal parameters?

4.16 What is the generic mechanism used by the semaphore calling sequence to ensure synchronization between two processes? Between two threads?

4.17 In Figure 4.12, what is the purpose of accessing the shared segment?

4.18 When creating a thread within a process, how is the thread's stack handled?

4.19 Throughout the examples of this chapter, error checking for such actions as creating a thread have been deleted from the basic code. Please comment on this lack of error checking intrinsic to the programs.

4.20 In the program tetra.c (Figure 4.19), what are the basic vertex points used for the tetrahedron?

4.21 In Figure 4.28, where does the routine dja.c acquire the function bboxx()?

5 Additional OS/2 Considerations

In Chapter 4, C programming for the OS/2 full-screen mode was presented. We saw that the C code required low-level access for many operations that call API functions. C has the advantages afforded high-level languages as well as low-level access to the operating system via these API services. Hence C is an ideal candidate language for programming OS/2. Occasionally, a need will exist for generating a special-purpose routine in assembly language and interfacing it to C program code. The methodology for accomplishing this is discussed in Section 5.1.

The Microsoft linker, which comes with the C compiler utilities, serves to load and link the object modules required by a program. When an .exe program is generated by the linker, all needed object modules must be input to the link process. This can be cumbersome, especially when large library files are called by the linker. OS/2 possesses the capability for two types of dynamic linking using dynamic linked libraries (DLL), where external references can be resolved by means other than the static linking normally employed. DLL satisfy external references either during loading, load-time linking, or at run-time, run-time linking. In the former case all external references are satisfied by the linker through knowledge of where the DLL routines reside. In this case a priori knowledge about the location of the needed DLL routines exists even though these routines are not located within the .exe module at linking. When a program begins execution the DLL routines are loaded, when called, based on this knowledge of where these routines reside. In run-time linking, the DLL routines are located at the time they are called and then loaded. The latter approach takes slightly longer than load-time linking but leaves the basic executable module unencumbered.

Why would OS/2 implement dynamic linked libraries? Primarily because of the multitasking feature. Since multitasking and memory management require movement into and out of memory, it is desirable to keep executable modules as small as possible. DLL management is one technique for achieving small executable modules. In Section 5.2 we address DLL implementation. In the remainder of the chapter we consider programming conventions, take a brief additional look at the API, and study a representative C example.

5.1 MIXED-LANGUAGE PROGRAMMING AND OS/2

We have seen how both assembly language and C code can be used for programming under OS/2. C code is preferred as the level of abstraction or task complexity increases. C, however, yields less optimized object code than does assembly language programming. Hence for critical applications, C routines must be interfaced to assembly language modules. This is particularly true when hardware is to be accessed directly or execution speed is important. In this section, where we address the integration of C and assembler, we will assume that the C modules call assembler modules when appropriate.

Assembler has a basic template for setup to interface to Microsoft C:

```
TITLE...
;
; Description...
;
_DATA1          SEGMENT  BYTE  PUBLIC  'DATA'
                PUBLIC  _VAR1,...
_VAR1           ...
_DATA1          ENDS
;
```

```
_TEXT           SEGMENT BYTE PUBLIC 'CODE'
                ASSUME CS:_TEXT,DS:_DATA1
                PUBLIC _Function*
_Function       PROC NEAR
                PUSH BP
                MOV BP,SP
                SUB SP,10
                PUSH BX
                PUSH CX
                PUSH DX
                PUSH SI
                PUSH DI
                                        ;
                PUSH DS
                MOV AX,SEG _DATA1
                MOV DS,AX
                ...
                (main body)
                ...
                MOV AX,...
                POP DS
                POP DI
                POP SI
                POP DX
                POP CX
                POP BX
                MOV SP,BP
                POP BP
                RET
_Function       ENDP
_TEXT           ENDS
                EMD
```

To interpret this template, consider first the segment definitions. Two segments are defined: _TEXT and _DATA1, the code segment and the data segment, respectively. The data segment is not _DATA because all parameters from the calling data segment will be passed using the stack. Hence there is no need to keep the "old" data segment during execution of the assembler code. The new segment, _DATA1, is optional as needed. Following definition of the segment registers using ASSUME, a procedure, _Function, is defined. This function must be PUBLIC so that it can be called externally. Upon entry to the procedure, a return address will be pushed on the stack. This address is an offset (2 bytes) for NEAR calls. After the call the calling routine has its frame pointer in the BP register. This pointer serves as the basis for moving from frame to frame. The template requires pushing this address on the stack. Four bytes now reside on the stack for a NEAR call. The stack pointer now contains the new frame pointer, which is loaded into BP, and space is allocated on the stack by advancing the stack pointer 10 bytes (as an example). These steps are accomplished with the code

```
PUSH  BP
MOV   BP,SP
SUB   SP,10
```

Next, the general-purpose registers (except AX) and the index registers are pushed on the stack. Finally, the old data segment address (appearing in DS) is saved on the stack and a new data segment address for _DATA1 is loaded into DS.

The parameters passed to the assembler reside starting at [BP+4] because a return address and a frame pointer have been loaded. Assuming that all parameters are of type int, they will reference as [BP+4], [BP+6], [BP+8], and so on. Clearly, other data types will occupy space accordingly. The return values from the assembler routine will occupy AX or AX and DX. At this point all pushed registers are popped, the caller's frame pointer restored, and the return address accessed. This, then, briefly describes the template for interfacing assembler to C.

Figure 5.1 illustrates a C program that reads an upper and lower frequency, a number of iterations, and an individual tone duration (in milliseconds). The program generates a musical or tone scale at intervals of 100 Hz for the range of frequencies spanned by the upper and lower frequencies. This C program calls an assembly language routine, scales1(), which accesses the tone generator. This assembly language routine appears in Figure 5.2 and follows the normal assembly language template for the C interface [1]. Note that the main portion of this routine simply passes the formal parameters to @DosBeep. In this case the frequency, freq, is passed at [BP+4] and the duration, dduration, is passed at [BP+6]. The inclusion

```
...
IF1
          include  sysmac.inc
ENDIF
...
```

loads the API services as required.

The routine scales1.asm, which appears in Figure 5.2, was assembled with the instruction

```
masm  scales1
```

The C program appearing in Figure 5.1 was compiled using

```
cl -c -Zi; scales.c
```

The linking was accomplished as

```
link  scales+scales1,  scales,,doscalls,,,/CO
```

```
/* This program generates scales and
 * calls an assembler routine */

#define INCL_BASE
#include <os2.h>
#include <stdio.h>

UINT low_freq,high_freq,no_iterations,dduration;

UINT action = 0;
UINT error_code = 0;

main()
        {

        printf("Input lower frequency (Hz) - integer\n");
        scanf("%d",&low_freq);
        printf("Input higher frequency (Hz) -- integer\n");
        scanf("%d",&high_freq);
        printf("Input number iterations \n");
        scanf("%d",&no_iterations);
        printf("Input component duration \n");
        scanf("%d",&dduration);

        sscale();                                       /* tone generator */

        DosExit(action,error_code);
        }

sscale()
        {
        extern scales1();                               /* assembler module */
        int freq,n,m,N;

        low_freq = low_freq/100;                        /* normalize */
        low_freq = low_freq * 100;

        if(low_freq <= 100)
           low_freq = 200;                              /* minimum set */

        high_freq = high_freq/100;                      /* normalize */
        high_freq = high_freq * 100;

        m = 0;                                          /* initialize loop */
        N = (high_freq - low_freq)/100;                 /* no. tone points */

        while(m <= no_iterations)                       /* check limit */
           {
           for(n = 1;n <= N;n++)                        /* up-scale */
              {
              freq = low_freq + n *100;                 /* set frequency */
              scales1(freq,dduration);                  /* tone */
              }
           for(n = 1;n <= N;n++)                        /* down-scale */
              {
              freq = high_freq - n * 100;               /* set frequency */
              scales1(freq,dduration);                  /* tone */
              }                                         /* increment loop */
           m++;
           }
        }
```

Figure 5.1 C program to generate musical scale based on input frequencies and time duration.

```
PAGE 40,132
TITLE scales1 - Routine to generate scales (sacles1.asm)
;
;       DESCRIPTION: This routine generates various tones.
;       [BP+4] contains the frequency and [BP+6] contains
;       the duration in millisec.
;
IF1
        include sysmac.inc
ENDIF
;
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        ASSUME CS:_TEXT
        PUBLIC _scales1
_scales1        PROC NEAR
                                            ;
        push BP
        mov BP,SP
        sub SP,8
        push DI
        push SI
        push AX
        push BX
        push CX
        push DX
                                            ;Begin beep
        @DosBeep [BP+4],[BP+6]
                                            ;
        pop DX
        pop CX
        pop BX
        pop AX
        pop SI
        pop DI
        mov SP,BP
        pop BP
                                            ;
        ret
_scales1        ENDP
_TEXT   ENDS
        END
```

Figure 5.2 C-callable assembly language routine to generate tonals.

5.2 DYNAMIC LINKING AND RESOURCE MANAGEMENT

Dynamic linking is a method of generating an executable program where not all modules are loaded into the execute file at link time but are loaded on demand during execution [2]. Under OS/2 a single code segment can be accessed by multiple programs, and such reentrant code facilitates dynamic linked library (DLL) usage, where simultaneous access of a library routine is possible. This is in keeping with the goal of minimizing code in a multitasking environment.

The two types of dynamic linking, load-time linking and run-time linking, serve distinctly different needs. Load-time linking involves complete knowledge of where a needed external routine resides prior to execution and is appropriate for frequently used routines. Run-time linking requires locating and installing external routines upon their call from an executing program. This form of linking is used primarily for accessing routines on an infrequent basis.

5.2.1 Using Dynamic Linked Libraries

We have seen that dynamic linked libraries (DLL) are useful when it is desirable to minimize the amount of code linked with multiple executable routines or tasks in a multitasking environment. It is in support of multitasking that DLL can contribute significantly. The run-time dynamic linking circumvents this situation where a DLL can be released from an executable module.

There are actually three types of linked modules possible when dynamic linking is employed:

1. Load-time dynamic Linking
   a. Preloaded DLL
   b. Load on Call DLL
2. Run-time dynamic Linking
   a. Explicit Load and Call

To fully appreciate the nuances among these options, we must examine the concept of a definition file, where one of these options is determined for each DLL implementation. Briefly, the preloaded DLL requires that these DLL routines be loaded at the start of execution. The load on call DLL implies that the code be loaded as each DLL routine is called by the executing program using guidance in the definition file. Finally, the explicit load and call situation for run-time dynamic linking requires that the DLL be accessed using API services. We will consider each type of DLL access in Section 5.2.4.

The LINK utility is used to join object routines into executable modules that have all their external references accounted for. The linker is used to create either a DLL or an executable, .exe, file. How does this work? Basically, the module definition file (.def) specifies whether or not a particular output (from the linker) is to be a DLL or an .exe file. This definition file also includes a number of statements that can be used to tailor executable code to accomplish various optimizations. It includes information that distinguishes between a DLL or application, a list of imported and exported functions (see below), the size of the stack and heap, and a number of options for the code and data segments. The latter option allows specification of whether or not segments are to be preloaded or loaded on demand. When using the linker we have noticed that a prompt for a definition file always occurs as the last entry in the linker prompt sequence. So far we have left this entry blank, which is an appropriate default for applications. We will now use this prompt to supply a .def file where appropriate.

5.2.2 The Definition File

Table 5.1 illustrates the allowed (and in some cases the mutually exclusive) statements that can appear in the module definition file. The first two statements are either NAME or LIBRARY. The former specifies the name of an application (.exe) and the latter specifies the name of a DLL (.dll). The description (DESCRIPTION) merely states in prose the module purpose. The statement PROTMODE specifies that a module is to run under Protected Mode. The statement

```
CODE  [load][shared][execute][privilege]
```

is used to define the default attributes for all the module's code segments. Subsequent statements using the SEGMENTS key word can override this statement (CODE) to tailor segment usage. In the CODE statement above, [load] is used to specify whether or not the code segments are physically loaded at the start of execution or on demand. This option has two possible values: PRELOAD (for loading at start of execution) and LOADONCALL (for demand loading).

The next option, [shared], specifies whether code segments in a DLL are to be accessed by all tasks needing these segments as a single instance or as multiple instances (where duplicate copies of the DLL routine are generated). This option has two possible values: SHARED (where only one copy of the code segment exists) or NONSHARED (where a unique copy of the code segments is loaded for each reference). The option [execute] allows code segments to remain distinct through the value EXECUTEONLY. In this case the code segment selector cannot be loaded into DS. The alternative value, EXECUTEREAD, permits the segment selector to be loaded into DS. Finally, [privilege] is used to give code segments I/O privilege at level 2 by having IOPL specified.

TABLE 5.1 MODULE DEFINITION FILE STATEMENTS

Statement	Comments
NAME	Declares a module as an application
LIBRARY	Declares a module as a DLL module
DESCRIPTION	Defines module descriptively
PROTMODE	Declares a module as a Protected Mode routine
CODE [load][shared][execute][privilege]	
	[load]: specified whether code loaded at the start of execution (PRELOAD) or on demand (LOADONCALL)
	[shared]: one copy of code loaded (SHARED) or multiple copies loaded with tasks (NONSHARED)
	[execute]: (EXECUTEONLY)—code segments can only be executed; or (EXECUTEREAD)—they can be read as well
	[privilege]: allows code segment I/O capability (IOPL)
DATA [load][instance][shared][write][privilege]	
	[load]: specifies whether code loaded at the start of execution (PRELOAD) or on demand (LOADONCALL)
	[instance]: no automatic data segment created (NONE), all instances share the same automatic data segment (SINGLE),and multiple copies for each instance (MULTIPLE)
	[shared]: same copy of a segment shared (SHARED) and new copies loaded for each instance (NONSHARED)

TABLE 5.1 *(Concluded)*

Statement	Comments
(READWRITE)	[write]: specifies that a memory segment can be written to or only read (READONLY)
SEGMENTS	[segname][CLASS('classname')][minalloc][segflags] [segname]: name of segment whose attributes are to be changed [classname]: 'CODE' or 'DATA' [minalloc]: minimum number of bytes reserved for segment [segflags]: attributes assigned to segment
IMPORTS	[intname]modulename.[entryname or entryordinal] [intname]: name to be used within importing module modulename: application library that contains functions [entryname]: entry point to DLL routine [entryordinal]: DLL routine ordinal position
EXPORTS	entryname[=intname][@ordinal][RESIDENTNAME][NODATA]argnum entryname: name to be used by accessing routines [intname]: real name of routine [@ordinal]: defines the routine's ordinal value within export module [RESIDENTNAME]: used with @ordinal argument to specify resident always [NODATA]: if present, specifies no stack or automatic data segment argnum: number of parameters to be received or IOPL
STACKSIZE	Number of bytes an application or DLL needs for its own stack
HEAPSIZE	Number of bytes in application or DLL heap
STUB	Name of a DOS 3.x program to replace an application or library invoked in Real Mode instead of correctly specifying Protected Mode

The statement

```
DATA [load][instance][shared][write][privilege]
```

is used to specify the default attributes for all the module's data segments. The first, [load], is the same as for the CODE statement except that LOADONCALL is the default option when no load argument is specified. The option [instance] describes the automatic data segment, which is the physical segments(s) represented by the name dgroup. This segment(s) contains the local heap and stack area for an application. It can take one of three values: NONE (no automatic segment), SINGLE (all application instances share the same automatic data segment), and MULTIPLE (default value where each instance has its own automatic data segment).

The argument [shared] parallels the [instance] value. It has two values: SHARED (same copy of a segment is shared by multiple instances of an application) and NONSHARED (new copies of data segments are loaded for each instance of an application; this is the default value). The [shared] argument and the [instance] argument must match. If a conflict arises, all segments in dgroup are shared, and all others are nonshared.

The argument [write] specifies whether the data segments can be written to or not: READONLY (cannot be written) and READWRITE (default option; the segment can be written to as well as read). The argument [privilege] is the same as for CODE. The statement

```
SEGMENTS
  [segname][CLASS('classname')][minalloc][segflags]
```

is used to assign attributes individually to code or data segments. The parameter [segname] denotes the segment label and can be declared as 'CODE' or 'DATA' via the classname. The default is 'CODE'. Each segment is allocated a minimum number of bytes: [minalloc]. The argument [segflags] can be any combination of arguments specified above with CODE or DATA segments.

The form of the IMPORTS statement is

```
IMPORTS
  [intname] modulename. [entryname|entryordinal]
```

Here intname specifies the internally used name of the importing module as it calls an external entry point (the ASCII string, entryname, within the DLL). The parameter modulename is the name of the application or DLL containing the needed functions, and entryordinal merely identifies entryname by its ordinal position with the DLL.

The form of the EXPORTS statement is

```
EXPORTS
  entryname[=intname][@ordinal][RESIDENTNAME][NODATA]argnum
```

This statement defines the routines within a DLL or application that are to be available for other programs. Alternatively, it can be used to specify routines that are to have level 2 I/O privileges. The argument entryname defines the name that calling modules will use when accessing the exported routine. The parameter [=intname] is the real name appearing in the exporting routine. The [@ordinal] parameter defines the routine's ordinal value within the module.

The argument [RESIDENTNAME] is used only when [@ordinal] is specified and it indicates that the function's name must be resident at all times and, consequently, the name and ordinal value will be stored in the DLL export table. The [NODATA] argument means that the export routines will have neither a stack nor an automatic data segment. Finally, argnum takes on the value IOPL when the export routine is to have level 2 privilege.

STACKSIZE specifies the number of bytes an application or DLL needs for the local stack. Similarly, HEAPSIZE specifies the number of bytes an application or DLL needs for its local heap. STUB specifies the name of a DOS executable file to be run in place of a Protected Mode application or library when such applications or libraries are invoked under Real Mode.

5.2.3 Creating a DLL

In the preceding section we saw examples of the use of IMPORT and EXPORT in the definition file. IMPORT specifies the routines that will be used by an executable file and incorporated at load or run time, as indicated in the application definition file. EXPORTS specifies the routines that will be transferred to the executable file at load or run time from the DLL, as indicated in the DLL definition file.

To see this work, consider the creation of a DLL and its incorporation with other modules. The following sequence of steps corresponds to Example 1 in Section 5.2.4, where the program code will be specified:

```
link dyninit.obj dlink1.obj, dyn11.dll, doscalls.lib, dyn11,def
```

This link statement links dyninit and dlink1 (both object modules) to create dyn11. We assume that dyn11.def has the LIBRARY option with dyn11 specified. Then the created routine has the .dll extension, denoting it as a dynamic linked library. The single library, doscalls.lib, imports the API service routines. Hence, from the foregoing process comes the DLL, dyn11.dll.

Dynamic linked libraries must be linked with applications as libraries, not .dll files. Hence the import library utility, implib, can be used to create this library based on the definition file. The routines above, for example, are in dyn11.dll, but the entry points can be specified in dyn11.lib, which is created as follows:

```
implib dyn11.lib dyn11.def
```

Here implib creates dyn11.lib, which has the entry points specified by the EXPORT table in dyn11.def.

The last step is to create the application run file and satisfy all external references through library access (as an example). Assume that the application exists as an object module named dyn1.obj. The link procedures will be

```
link dyn1.obj, dyn1.exe,,doscalls.lib dyn11.lib,,
```

Here dyn1.exe is the output for dyn1.obj and uses both doscalls.lib (the API service) and dyn11.lib (the library file for the DLL). During execution of dyn1.exe the DLL routines will be accessed via the dyn11.lib table (these DLL routines reside in dyn11.dll).

The importance of specifying EXPORT and IMPORT files should now be clear. It is the only way of identifying DLL routines to the implib utilities that create the DLL needed library file. This file points to the available DLL routines (this file is output from implib with extension .lib).

5.2.4 DLL Examples

In the preceding section we outlined how to create a DLL using an appropriately constructed module definition file. Both the link and implib utilities were used in this process. File names were specified without actually presenting the files themselves, in order that the mechanics of the linking process could be illustrated. By way of introducing an example, the code is now presented for the files in question.

Example 1. Figure 5.3 shows the main application program, dyn1.asm, which

```
PAGE 40,132
TITLE DYN1 - Main calling program example #1 (dyn1.asm)
;
;       DESCRIPTION: This program calls dyn11.dll to demonstrate
;       preloaded assembler dynamic link libraries.
;
        .286c
        .sall                                   ;Suppresses macro lists
;
        .xlist
           INCL_BASE equ 1
           include OS2.INC
        .list
;
        extrn dlink1:FAR
;
dgroup  GROUP   data                            ;defines automatic data seg
;
STACK   SEGMENT PARA    STACK 'STACK'
        db      64 dup('STACK   ')
STACK   ENDS
;
data    SEGMENT PARA    PUBLIC 'DATA'
;
msg1    db      'Dynamic Link Pre-loaded Routine',0DH,0AH
msg1_l  equ     &-msg1
msg2    db      'Error on access DLL',0DH,0AH
msg2_l  equ     $-msg2
;
action  equ     0                               ;code to end thread
result  dw      0                               ;return code for error
vio_hdl equ     0                               ;video handle
;
data    ENDS
;
CSEG    SEGMENT PARA    PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:dgroup
DYN1    PROC    FAR
                                                ;
        push ds                                 ;push message segment
        lea bx,msg                              ;offset of message
        push bx                                 ;save offset
        push msg1_l                             ;message length
                                                ;
        call dlink1                             ;DLL routine to print msg
                                                ;
        cmp ax,0                                ;check for error
        je EXIT                                 ;jump if OK
                                                ;
        @VioWrtTTY      msg2,msg2_l,vio_hdl
                                                ;
EXIT:
        @DosExit        action,result
DYN1    ENDP
CSEG    ENDS
        END DYN1
```

Figure 5.3 Main dynamic link library calling program, illustrating preloaded DLL routines.

imports routines from the DLL. It is this file that must be linked with doscalls.lib and dyn11.lib to create the application executable module. The data segment for this module is named data and the code segment is CSEG. In the figure three values are pushed to the stack: first, the data segment address for data; second, the offset address for msg1; and finally, the length of msg1 is pushed. These values will eventually be accessed using a structure (template) and the stack starting address. During a push operation a variable is placed on the stack and then the stack pointer is decreased. Hence the stack loads as

msg1_l	(lowest address)
offset of msg1	(next lower address)
segment address of msg1	(highest address)

Following this loading of the stack a call to the DLL routine (dlink1) is made. The operating system automatically places a return address (4 bytes for a FAR call) on the stack in response to the call instruction. Figure 5.4 illustrates the called procedure, dlink1, contained in the module dlink1.asm. The first instruction of the called procedure saves the old (current) value of bp and sp now points to this saved copy of bp. Recognizing that the stack pointer is decreased following each push, the following stack values appear on the local stack:

...	
caller's bp	(lowest address)
caller's ip	
caller's cs	
msg1_l	
offset of msg1	
segment address of msg1	(highest address)
...	

In the routine dlink1 the next instruction transfers the value in sp (which is the saved copy of the old bp's address) to bp. The routine pushes ds (which points to the dyn1.asm data segment) and loads the DLL routines's data segment address into ds. The program can now access structures of the form specified by st1, which appears in the new data segment. This access takes the form

```
variable.field
```

where the fields specified for st1 are m_len, m_offs, and m_seg. There are also three unnamed fields. Choosing variable equal to the address of the old bp, we can access the stack using the structure template. Hence the following values of interest can be retrieved by the DLL routine:

```
PAGE 40,132
TITLE DLINK1 -- DLL routine for example #1 (dlink1.asm)
;
;       DESCRIPTION: This is the DLL routine for example #1.
;       It is pre-loaded.
;
        .286c
        .sall                                   ;Suppresses macro listings
;
extrn   VioWrtTTY:FAR                           ;API routine
;
dgroup GROUP data_dll1
;
data_dll1       SEGMENT PARA PUBLIC 'DATA'
;
vio_hdl equ 0                                   ;video handle
;
st1     struc                                   ;parameter structure
        dw      ?                               ;caller's bp
        dw      ?                               ;caller's ip
        dw      ?                               ;caller's cs
m_len   dw      ?                               ;message length
m_offs  dw      ?                               ;message ptr offset
m_seg   dw      ?                               ;message ptr seg
st1     ENDS
data_dll1       ENDS
;
CODE1   SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE1,DS:dgroup
        PUBLIC  dlink1
dlink1  PROC    FAR
                                                ;
        push bp                                 ;caller's frame ptr
        mov bp,sp                               ;local stack
        push ds                                 ;caller's ds
        mov ax,data_dll1                        ;load new ds
        mov ds,ax
                                                ;use explicit parameter
                                                ;values because locations
                                                ;come from local stack
                                                ;
        push [bp].m_seg                         ;message seg address
        push [bp].m_offs                        ;message offset address
        push [bp].m_len                         ;length message
        push vio_hdl                            ;video handle
                                                ;
        call FAR ptr VioWrtTTY                  ;direct API call
                                                ;
        pop ds                                  ;restore caller's data seg
        pop bp                                  ;restore frame pointer
        ret
dlink1  ENDP
CODE1   ENDS
        END
```

Figure 5.4 The called procedure, dlinkl, for the example of Figure 5.3.

```
[bp].m_len – length of msg1
[bp].m_offs – the offset of msg1
[bp].m_seg – the segment address of msg1
```

In dlink1 these values are pushed on the stack as well as the video handle and a FAR call to

```
VioWrtTTY
```

is made. Note that macro version is not used and the API service directly accessed.

Execution of these instructions results in the message "Dynamic Link Preloaded Routine" appearing on the screen with a line feed and carriage return.

The routine dlink1 is a member of the DLL, dyn11.dll module created by the first link discussed above. This routine, dlink1.obj, was linked with a routine dyninit.obj. What is this routine? Each DLL must have an initialization routine. For dyn11.dll the initialization routine is dyninit.obj and this routine appears in Figure 5.5. This initialization routine simply writes the message "DLL Initialized" to the screen and forces the operating system to initialize the DLL. To ensure that the initialization routine executes first the DLL entry point to the module's procedure, init is specified as a parameter in the END pseudo-op: END init. Following initialization the routine must return a value of 1, not 0, to the calling program.

```
PAGE 40,132
TITLE DYNINIT -- Initialization Routine for DLL-1 (dyninit.asm)
;
;       DESCRIPTION: This routine is the initialization routine
;       for the DLL dyn11.dll.  The routine merely prints a message.
;
        .286c
        .sall                                   ;Suppresses macro listings
;
        .xlist
           include sysmac.inc                   ;include API services
        .list
;
dgroup  GROUP init_data                         ;defines automatic data seg
;
init_data       SEGMENT PARA PUBLIC 'DATA'
;
initOK  equ     1                               ;OK return code
msg     db      'DLL Initialized',0DH,0AH       ;Initialization message
msg_l   equ     $-msg                           ;message length
vio_hdl equ     0                               ;video handle
;
init_data       ENDS
;
CINIT   SEGMENT PARA PUBLIC 'CODE'
ASSUME  CS:CINIT,DS:dgroup
init    PROC    FAR
                                                ;
        push bp                                 ;save frame pointer
                                                ;
        @VioWrtTTY msg,msg_l,vio_hdl            ;Write initialization msg
                                                ;
        mov ax,initOK                           ;OK return value
                                                ;
        pop bp                                  ;restore frame pointer
        ret
init    ENDP
CINIT   ENDS
        END init                                ;DLL entry point
```

Figure 5.5 The initialization routine for the DLL called by the program in Figure 5.3.

Finally, Figure 5.6 presents the module definition file. This file is associated with the library dyn11.lib. Hence all references to segments and routines must come from dlink1.asm and dyninit.asm. In this example these segments are preloaded, as specified in the definition file. Only dlink1 is exported because this is the only routine used by dyn1.exe. Note that the presence of LIBRARY specifies that dyn11 will be a DLL.

```
LIBRARY         DYN11

PROTMODE

DESCRIPTION     'Example #1 DLL'

SEGMENTS
        init_data       CLASS   'DATA'  PRELOAD
        CINIT           PRELOAD
        data_dll1       CLASS   'DATA'  PRELOAD
        CODE1           PRELOAD

EXPORTS
        dlink1
```

Figure 5.6 The module definition file, dynll.def.

Example 2. For this example, a load-on-call execution was prescribed for the DLL routines exported to the application. All routines remain the same as in Example 1 except the definition file, named dyn11.def. This file appears in Figure 5.7.

```
LIBRARY         DYN22

PROTMODE

DESCRIPTION     'Example #2 DLL'

SEGMENTS
        init_data       CLASS   'DATA'  LOADONCALL
        CINIT           LOADONCALL
        data_dll1       CLASS   'DATA'  LOADONCALL
        CODE1           LOADONCALL

EXPORTS
        dlink1
```

Figure 5.7 The module definition file, dyn22.def.

It is important to note that the data segments (init_data, for the initialization routine, and data_dll1, for the DLL routine) in the DLL as well as the Code segments in the DLL are loaded as called based on the parameter LOADONCALL.

The link and definition utilities are executed as follows:

```
link dyninit dlinkl, dyn22,,doscalls.lib,dyn22.def

implib dyn22.lib dyn22.def

link dynl.obj, dynl.exe,,doscalls.lib dyn22.lib,,
```

These commands produced an executable module, dynl.exe, which accessed the dynamic link library, dyn22.dll, and imported the routine dlinkl only when actually called in the program dyn1.exe. In the earlier example access was granted immediately because of the preload condition. Execution of dynl.exe produces the same result as it did in Example 1, but the DLL input occurred at load time rather than when linking takes place. Note that the message to the screen still reads

```
"DLL Initialized
Dynamic Link Preloaded Routine"
```

even though the load-on-call status exists. This is because we have changed only the definition file to check this DLL implementation.

Example 3. In this example we illustrate an example of run-time dynamic linking based on the OS/2 API services. Following the lead set by the first two examples, the routines dyninit.asm and dlink1.asm were used to make up the DLL. Figure 5.8 illustrates the main calling module, dyn2.asm.

The routine dyn2.asm uses the API service @DosLoadModule to load the DLL (DYN33.DLL) and it returns a handle to the DLL. To access entry points within DYN33.DLL the API services macro @DosGetProcAddr is called and an address returned for the entry DLINK1. This address is returned in addr_proc. Next the message parameters are pushed on the stack so that they can be accessed by DLINK1. A call is made via the CALL instruction and this writes msgl to the screen. Following release of the DLL using @DosFreeModule, the program exit takes place. Note that the two ASCIIZ strings corresponding to the DLL and the entry point name have uppercase letters. This is because the assembler always uses uppercase and both the API functions in dyn2.asm are case sensitive. The CALL instruction is not case sensitive; hence the references to dlinkl in dynl.asm are unaffected by whether the reference is to dlinkl or DLLNK1. In dyn2.asm the references must be upper case.

Figure 5.9 illustrates the DLL definition file for Example 3. There are few changes in this file (DYN33.DEF) from earlier .def files. These three examples constitute the three ways in which dynamic link libraries can be implemented under OS/2. The examples all employed assembly language for both the calling module and the routines appearing in the DLL. The link sequence for Example 3 is as follows:

```
link dyninit + dlinkl,dyn33,,doscalls,dyn33.def

implib dyn33.lib dyn33.def

link dyn2,dyn2,,doscalls+dyn33,,
```

```
PAGE 40,132
TITLE DYN2 - Main calling program example #3 (dyn2.asm)
;
;       DESCRIPTION: This program calls dyn33.dll to demonstrate
;       explicit load by application assembler dynamic link libraries.
;       The main calling routine is DYN2.
;
        .286c
        .sall                                   ;Suppresses macro lists
;
        .xlist
           include sysmac.inc                   ;API services
        .list
;
dgroup  GROUP   data                            ;defines automatic data seg
;
STACK   SEGMENT PARA    STACK 'STACK'           ;stack defined
        db      128 dup('STACK   ')
STACK   ENDS
;
data    SEGMENT PARA    PUBLIC 'DATA'
;
msg1    db      'Dynamic Link Run-Time Routine',0DH,0AH
msg1_l  equ     $-msg1
msg2    db      'Error on access DLL',0DH,0AH
msg2_l  equ     $-msg2
;
action  equ     0                               ;code to end thread
result  dw      0                               ;return code for error
vio_hdl equ     0                               ;video handle
;
obj_buf         dd      64 dup(?)               ;error returm buffer
obj_buf_len     dw      $-obj_buf               ;length error buffer
obj_buf_add     dd      obj_buf                 ;address buffer
name_mod        db      'DYN33',0               ;module name
hhandle         dw      0                       ;handle to DLL module
;
name_proc       db      'DLINK1',0              ;DLL procedure name
addr_proc       dd      0
;
data    ENDS
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:dgroup
DYN2    PROC    FAR
                                                ;

        @DosLoadModule obj_buf_add,obj_buf_len,name_mod,hhandle
        cmp ax,0                                ;check for error
        jne ERROR
                                                ;
        @DosBeep 4000,200
                                                ;
        @DosGetProcAddr hhandle,name_proc,addr_proc
        cmp ax,0                                ;check for error
        jne ERROR
                                                ;
        @DosBeep 1000,500
                                                ;
        push ds                                 ;push message segment
        lea bx,msg1                             ;offset of message
        push bx                                 ;save offset
        push msg1_l                             ;message length
                                                ;
        call addr_proc                          ;DLL routine to print msg
                                                ;
```

Figure 5.8 Run-time dynamic linked library calling module.

```
        @DosFreeModule hhandle
        cmp ax,0                                ;check for error
        je EXIT
ERROR:
        @VioWrtTTY      msg2,msg2_l,vio_hdl     ;error message
EXIT:
        @DosExit        action,result           ;terminate all threads
DYN2    ENDP
CSEG    ENDS
        END DYN2
```

Figure 5.8 (*Concluded*)

```
LIBRARY         dyn33   INITINSTANCE

PROTMODE

DATA            NONSHARED

DESCRIPTION     'Example #3 DLL'

SEGMENTS
        init_data       CLASS   'DATA'  LOADONCALL
        CINIT   LOADONCALL
        data_dll1       CLASS   'DATA'  LOADONCALL
        CODE1           LOADONCALL

HEAPSIZE        1024

EXPORTS
        dlink1
```

Figure 5.9 The module definition file, dyn33.def.

It is possible to access a DLL in an additional fashion: through specification of IMPORTS entry points in a module definition file. By listing the library entry points to be imported in a definition file for the application, the application can import DLL routines.

5.3 OPTIMIZING THE C DESIGN PROCESS

Most modern textbooks address the topics of structured programming, modular code, and top-down design. These techniques have come to embody an organized approach to program development which is repeatable in an optimal sense. This approach is predictable and meaningful in that programmers of differing backgrounds will approach algorithm development in the same fashion when these tools are used. In the following discussion we explore each of these topics, starting with top-down design because it represents the start of the design process.

5.3.1 Top-Down Design, Structured Programming, and Modular Code

Top-down design is an informal strategy for starting with a global problem statement and then subdividing the development into smaller and smaller modules until each module accomplishes a singular task. Such a systematic approach to design leads to modular techniques that develop and link program elements together to solve the overall task.

A convenient starting point for the top-down approach is to define the functional structure for the program under consideration. This functional structure has been reflected in the Structure Charts of earlier programming examples. These charts illustrate a hierarchy of importance for the components of the program. Structure Charts are established by associating with level 0 an overall functional statement of the programming problem. This occupies a single box at the top of the hierarchy. Next, the level 1 position categories associated with variable I/O and algorithm computation are indicated at a reasonably high level. Below this level, successive reduction of the problem into multiple smaller pieces occurs, with the relationships clearly defined. Through this process the program architecture is defined in terms of hierarchy. The Structure Chart does not, however, illustrate the dynamic interrelationship among modular components. Also, it does not illustrate at the module level the flow of execution for the program. To achieve this, the top-down design process needs an additional mechanism for describing program activity. This mechanism can take one of two forms: the flowchart or pseudo-code, which describes the program activity in natural-language syntax. In this book we employ flowcharts for describing programs dynamically. (The reader can just as conveniently approach program design using pseudo-code, but it is generally less compact than flowcharts, hence our use of the latter technique.)

In general, the procedure discussed here is a reasonable approach to program design. The structure chart reflects the high-level functionalism and the flowcharts illustrate the more detailed dynamics at the module level. Top-down design is informal and thus most useful for small-scale design tasks. When faced with larger design problems, the programmer must resort to additional techniques to supplement the guidance obtained from the top-down methodology. We will consider two additional tools, as discussed earlier: modular programming and structured code.

The reader will note by now that in programs appearing in this book, there is a tendency to relegate many of the tasks to smaller functions and modules. This suggests the notion of modular programming. Modular programming concepts have been established over a long period of time during which theoretical methods evolved for designing programs [3,4]. The principal requirement on smaller program units or modules is that they be independently testable and can be integrated to accomplish the overall program objective.

Modules are generally defined, in the context of C programming, in terms of one or more related functions. Each function should perform a single independent task and be self-contained, with one exit and entry point. This suggests a single "thread" to program execution, which is the manner in which most modern computers execute code. Module execution in the world of the CPU is sequential. With this architecture in mind, it becomes straightforward to accept the one entry and one exit feature associated with functions, at least theoretically.

We discussed global variables as a mechanism for returning more than one value from a function. When is this likely to become most necessary? One situation is the generation of an array of similar values. Here a single function might be used to compute a time series, for example, and each computation would generate an array element in recursive fashion. Obviously, the function should be self-contained and all array values generated at once. Consider the following function:

```
filter (N)
  int N;
  {
  extern float y[]. x[];        /*x=time element*/
  float b, c;
  int n;
  y[0]=0;                       /*Initialize*/
  b=.01;
  c=.001;

  for (n=1; n<=N; n++)
        y[n]=c*y[n-1]+b*x[n];

  }
```

This low-pass filter generates a smoothing of the time series x[n]. The array y[n] is generated in its entirety with the simple one-statement for loop. It would be highly undesirable to fail to return the complete array from this function; hence the use of global variables is appropriate. This use does not detract from the modular nature of the function. Thus even though a module (or function) has one entry and one exit point, multiple values can be returned.

Size is handled by accepting a general guideline that modules contain between 10 and 100 lines of code. This is suggested as a rule of thumb and if, for example, the code were written in APL, 100 lines would be very tiresome to debug. For C, however, the guideline seems appropriate, as evidenced by the modules in this book. A more rigorous enforcement of size must resort to quantitative measures such as complexity and complexity metrics. Consider the metric [5]

$$N = N1 + N2 \tag{5.1}$$

where

N1 = total number of operators in a module
N2 = total number of operands in a module

Returning to the function filter, the following counts apply:

OPERATOR	COUNT	OPERAND	COUNT	
=	6	y[0]	2	
for	1	b	3	
<	1	c	3	
++	1	0	1	
*	2	y[]	3	
		x[]	2	
		n	7	
		N	3	(variable)
	11		24	

Here N is 35 [from Equation (5.1)]. What does this mean? To interpret N, a body of statistical data must evolve based on complex interactions between programmers and code. It is clear from this example that N = 35 is a reasonably simple program. In reference 6, a similar example with N = 28 is illustrated. Again, this metric value indicates a relatively small and understandable module.

The programmer is unlikely to apply a complexity metric during program development. It is, nonetheless, useful to allow complexity to guide program development, particularly when modules begin to approach a high level of difficulty (for the programmer developing the code). Let us summarize some general guidelines on module development *in C*:

1. Restrict modules to between 10 and 100 lines of code.
2. Within the module concept, allow one entry and one exit (exception handling can call for multiple exits, but this is an abortive condition and the error state should be flagged before the exit).
3. Arrays should be treated globally.
4. Modules returning a single value should do so formally with return().
5. Modules returning multiple but small numbers of variables should do so with pointers (pointers are complex, so the trade-off here is how many pointers are involved).
6. Modules returning large numbers of variables, other than arrays, should be rewritten.
7. Modules returning an intermediate number of variables can do so with global declarations.
8. A module should perform one self-contained task.
9. Allow for exceptions to these guidelines when the code can be made easier to understand and it is not time-critical.

TABLE A.2 MACRO ASSEMBLER INSTRUCTIONS (8086 CONVENTION)

Instruction	Purpose	Comments
Arithmetic		
ADC dest,src	Add with carry	Performs an addition of the two operands and adds one if CF is set.
ADD dest,src	Addition	Adds the two operands.
DIV src	Unsigned divide	Divides the numerand (AL and AH for byte division and AX and DX for word division) by src. The result is returned in AL (byte) or AX (word).
IDIV src	Signed integer division	Signed division using the registers of DIV.
IMUL src	Signed integer multiply	Multiplies AL or AX times src.
MUL src	Unsigned multiply	Same as IMUL.
SBB dest,src	Subtract with borrow	Subtracts the two operands and subtracts one if CF is set.
SUB dest,src	Subtract	Subtracts the two operands.
Logical		
AND dest,src	Logical AND	Performs the bit conjunction of the two operands: the result is zero except when both bits are set.
NEG dest	Two's complement	Forms the two's complement of dest.
NOT dest	Logical NOT	Inverts dest bit by bit.
OR dest,src	Logical inclusive OR	Performs the bit logical inclusive disjunction of the two operands: returns a one except when both bits are zero.
TEST dest,src	Logical compare	Performs the bit conjunction of the two operands with only the flags affected.
XOR dest,src	Exclusive OR	Performs the bit logical exclusive disjunction of the two operands: returns a one when one operand is zero.
Move		
MOV dest,src	Move	Moves: 1. To memory from AX (AL) 2. To AX (AL) from memory 3. To seg-reg from memory/reg 4. To reg from seg-reg 5. To reg from reg To reg from memory To memory from reg 6. To reg from immediate 7. To memory from immediate
MOVS dest-str, src-str	Move byte or word string	Transfers a byte or word string from src, addressed by SI, to dest, addressed by DI.
Load		
LODS src-str	Load byte or word string	Transfers a byte (word) from src, addressed by SI, to AL (AX) and adjusts SI.

easier to implement in assembler since the API in OS/2 is presented as a generic assembler interface. C provides such an interface but is sophisticated and relies on the use of special macro libraries (see Appendix C) to set up the API function calls.

With these thoughts in mind, let us briefly summarize the IBM Macro Assembler/2. We will not attempt to illustrate examples of the language; the early chapters accomplish that. In this appendix we merely present the assembler constructs in tabular form based on reference 3. Table A.1 presents the addressing modes for the language. There are seven forms of addressing within the Macro Assembler. Table A.2 contains brief descriptions for the basic instruction set common to the Intel 8086

TABLE A.1 MACRO ASSEMBLER ADDDRESING MODES

Mode	Comment
Immediate	A byte or word constant in the source operand is loaded into a register operand. Example: mov ax,18.
Register	Register destination operands are loaded from register source operands. Example: mov ds,ax.
Direct	A register destination operand is loaded with the *value* of a location specified by its offset added to DS. Example: mov ax,dddw, where dddw is a variable in the data segment (addressed by DS).

Register indirect	The effective address (segment offset) is contained in BX, BP, SI, or DI, and this is used to load a register. Example:
	`mov bx, OFFSET dddw`
	`mov ax,[bx]`
	Here the brackets indicate bx contains an address.

Base relative	The effective address for the source is obtained by adding a displacement to BX or BP, which are assumed to contain an offset, Example:
	`mov bp,OFFSET dddw`
	`mov ax,[bp+4]`

Direct indexed	Here the effective source address is the sum of an index register (SI or DI) and an offset. Example:
	`mov si,4`
	`mov ax,dddw [si]`
	This loads ax with the same value as loaded in the base relative example.

Base indexed	Typically, the effective source address is the sum of a base register (BX or BP), an index register (SI or DI), and a displacement. Example:
	`mov bx,OFFSET dddw`
	`mov si,4`
	`mov ax,[bx][si+2]`

Appendices

A IBM Macro Assembler/2

In this appendix we present the IBM Macro Assembler/2, which can be used to program the assembler under both DOS and OS/2 [1, 2]. In the early chapters all programming was accomplished in assembly language. A major consideration, however, is how desirable is this choice of language for the IBM microcomputer environment? The answer lies in how the programmer intends to use the assembly language. Basically, assembler is very programming intensive and serves best when access of the system hardware is important. For the OS/2 Kernel service functions this is particularly important when acquiring or writing to the display, the video services (Vio), or the DOS functions (Dos) within the API. In addition to access of the system hardware, another associated requirement should be mentioned: *rapid* access of the hardware. The assembler is ideally suited for this requirement but has the added stipulation that the programmer must be prepared to devote considerable effort to writing very low-level code.

In this book an alternative language, the C programming language, is considered. C is more ideally suited for programming applications that require higher-level functions to accomplish tasks. Typical examples of these functions are sine and cosine, other hyperbolic and trigonometric functions, and a multitude of mathematical and statistical special-purpose functions. These can all be built up from assembly language programs by the user, but it is usually more desirable to access these functions through a high-level language, using the assembler-constructed libraries within the language. The general experience is, however, that the API calls are

5.12 Why are all lines appearing in Figures 5.19 through 5.23 not removed using the criteria that a negative direction to the facet normal constitutes a hidden line?

5.13 Show that the limit of Equation (5.13) is A^2 when $(x, y) = (0, 0)$.

5.14 In the surface plotting program appearing in Figure 5.14, the disk read function, xarray_diskrd(), is called prior to setting the CGA screen mode. Why?

5.15 In Figure 5.17 ,how would the function threeD_facets() change (aside from removal of the define and include statement associated with OS/2) if this routine were to execute in a normal Real Mode program?

2. Nguyen, T., and Moska, R., *Advanced Programmer's Guide to OS/2,* Brady Communications Company, Inc., New York, 1989, p. 316.
3. LaBudde, K., *Structured Programming Concepts,* McGraw-Hill Book Company, New York, 1987, p. 26.
4. Parnas, D., Information Distribution Aspects of Design Methodology, Carnegie-Mellon University Technical Report, Carnegie-Mellon University, Pittsburgh, PA, 1971.
5. Fitzsimmons, A., and Love, T., A Review and Evaluation of Software Science, *ACM Computing Surveys,* Vol. 10, No. 1, pp. 3–18, 1978.
6. Martin, J., and McClure, C., *Structured Techniques for Computing,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985, p. 70.
7. Dijkstra, E., Structured Programming, Software Engineering 1969, NATO Scientific Affairs Division, Brussels, Belgium, 1969.
8. Sedgewick, R., *Algorithms,* Addison-Wesley Publishing Company, Reading, MA, 1984.
9. Bentley, J., *Programming Pearls,* Addison-Wesley Publishing Company, Reading, MA, 1986.
10. Folk, M. J. ,and Zoellick, B., *File Structures,* Addison-Wesley Publishing Company, Reading, MA, 1987.
11. Petzold, C., *Programming the OS/2 Presentation Manager,* Microsoft Corporation, Redmond, WA 1989.

PROBLEMS

5.1 If a routine is self-contained within a task and called very infrequently, what OS/2 technique could be used to conserve storage?

5.2 What sequence of steps differentiates a DLL from another executable module?

5.3 In Figure 5.2, what is the parameter on the stack at BP+2?

5.4 How does one differentiate a DLL module from another source code module?

5.5 Define the difference between IMPORTS and EXPORTS as appearing in the module definition file.

5.6 What key API services are required for run-time DLL loading which are not required for load-time linking? Structurally, why are these API services needed?

5.7 If you wished to achieve an understanding of the relationships between program components, would you choose a Structure Chart or a flowchart? Explain.

5.8 What is the dominant characteristic of a C function? What are the implications of this characteristic for program structure?

5.9 In C program code, when is it appropriate to use globally defined variables? Why are these variables generally considered undesirable to program understanding?

5.10 Why is the include file "pmwin.h" not available as part of OS/2 Version 1.0?

5.11 In defining the three-dimensional surface, the grid of points is defined in the x-y plane and facets established by connecting cyclically the projected points defined by the surface itself. In plotting the resulting surface, the points are collapsed along the x-axis. Why is this necessary?

5.6 SUMMARY

This chapter has provided an examination of additional basic OS/2 features within the core API and has reinforced programming techniques within the C language, the primary language of choice for programming the Presentation Manager as well as more complex activities under OS/2. We began with a look at mixed-language programming in the context of C and assembler. Next, dynamic linked libraries were investigated from the viewpoint of assembly language, where they are more clearly understood. The use of load-time DLLs is recommended when memory allocation is to be optimized and a set of routines is to be called on a frequent basis. Load-time dynamic linking has the advantage that the calling routine does not need to ascertain the location of the module at loading. Run-time dynamic linking is recommended for those applications where memory allocation is dynamic and at a premium and the routines to be accessed are done so infrequently. This technique requires access through API service calls.

The C design process was examined from the viewpoint of top-down design, structured programming, and modular code. Recommendations were provided regarding module size and implementation. A typical template for C design was presented and the difficult issues of style and form touched upon. Style is such a crucial factor in the development of maintainable code that it is very surprising how often it is overlooked. Similarly, form can mean the difference between code that runs, and code that purports to accomplish the job but is so cumbersome and slow that it fails to achieve its objectives within a reasonable length of time.

All the API services return values that depend on the outcome of the call. The question of whether or not a program should monitor those outcomes, once the original debugging is complete, was left to the reader. Frequently, in the interest of brevity, the full API return checking has been suppressed in this book unless a hardware failure can result, such as the inability to access a disk.

A reexamination of the basic API core services was referenced: the Kbd, Mou, Vio, and Dos services. These are the Version 1.0 services and have since been added to by the presence of the PM functions, although these PM services are not treated in this book. Finally, a C application was developed. This application consisted of the plotting of a three-dimensional surface in the CGA mode of the OS/2 command screen mode. The purpose of the presentation was to illustrate the manner in which moderately large-scale programs would be interfaced to OS/2 in the normal operational environment, with a particular emphasis on the techniques outlined earlier in the chapter, such as modular code development.

REFERENCES

1. Godfrey, J. T., *Applied C: The IBM Microcomputers,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990, p. 237.

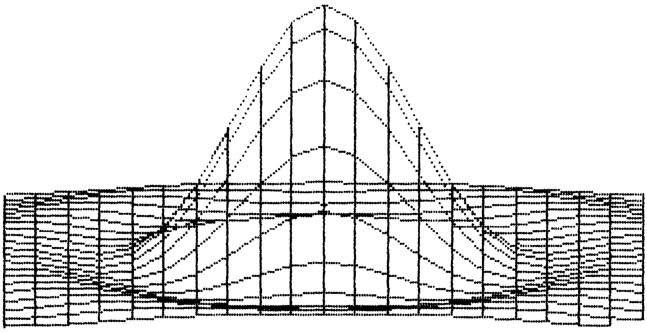


Figure 5.21 Surface plot with $N = 7$, $\alpha = \gamma = 0.0$ and $\beta = 1.2$.

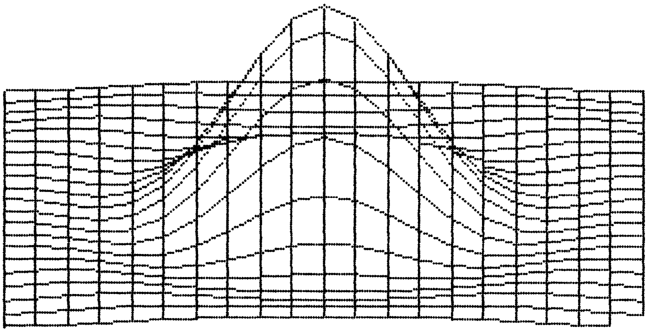


Figure 5.22 Surface plot with $N = 7$, $\alpha = \gamma = 0.0$ and $\beta = 1.4$.

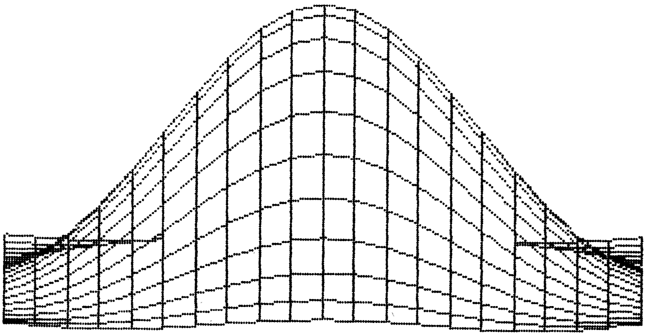


Figure 5.23 Surface plot with $N = 12$, $\alpha = \gamma = 0.0$ and $\beta = 1.0$.

```
        uwdot(col,row,MM1);                          /* erase dot */
        }
      }
    else
      {
      col = (int)(x1);
      if(y1 > y2)
        {
        for(row=(int)(y2)+1;row <= (int)(y1);row++)
          uwdot(col,row,MM1);                        /* erase dot */
        }
      else
        {
        for(row=(int)(y1)+1;row <= (int)(y2);row++)
          uwdot(col,row,MM1);                        /* erase dot */
        }
      }
    }
  }
```

Figure 5.18 (*Concluded*)

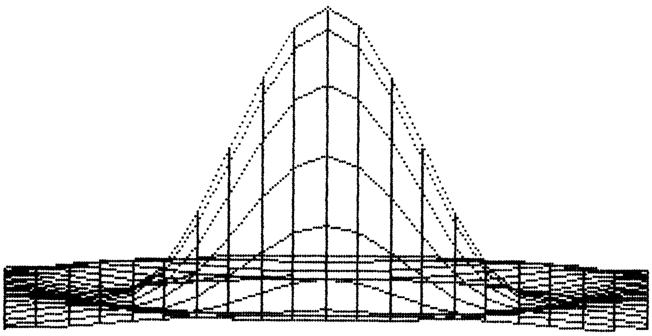


Figure 5.19 Surface plot with $N = 7$, $\alpha = \gamma = 0.0$ and $\beta = 0.8$.

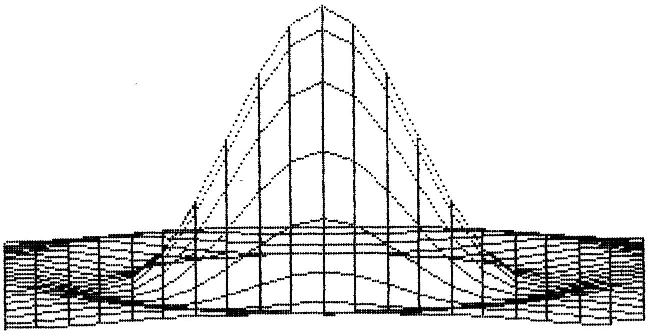


Figure 5.20 Surface plot with $N = 7$, $\alpha = \gamma = 0.0$ and $\beta = 1.0$.

```
        if(x1 == x2)
           m = 1000.;                                 /* zero divide */
        else
           m = (y2-y1)/(x2-x1);                       /* normal slope */
        if( x2 > x1)
           {
           for(col = (int)(x1)+1;col <= (int)(x2);col++)
              {
              row =(int)(y1 + m*(col - x1));          /* line equation */
              wdot(col,row,MM1);                      /* write dot */
              }
           }
        else
           {
           if(x2 < x1)
              {
              for(col =(int)(x2)+1;col <= (int)(x1);col++)
                 {
                 row=(int)(y2 + m*(col - x2));
                 wdot(col,row,MM1);                   /* write dot */
                 }
              }
           else
              {
              col = (int)(x1);                        /* verticle line */
              if(y1 > y2)
                 {
                 for(row=(int)(y2)+1;row <= (int)(y1);row++)
                   wdot(col,row,MM1);
                 }
              else
                 {
                 for(row=(int)(y1)+1;row <= (int)(y2);row++)
                    wdot(col,row,MM1);
                 }
              }
           }
        }

upltpt(x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;                                      /* slope */
        int row;
        int col;

        if(x1 == x2)                                  /* zero divide */
           m = 1000.;
        else
           m = (y2-y1)/(x2-x1);                       /* normal slope */
        if(x2 > x1)
           {
           for(col = (int)(x1);col <= (int)(x2);col++)
              {
              row = (int)(y1 + m*(col - x1));         /* line segment */
              uwdot(col,row,MM1);                     /* erase dot */
              }
           }
        else
           {
           if(x2 < x1)
              {
              for(col = (int)(x2)+1;col <= (int)(x1);col++)
                 {
                 row=(int)(y2 + m*(col -x2));
```

Figure 5.18 *(Continued)*

```
                /* Graph routines Protected Mode--gphrout.c */

#define INCL_BASE
#include <os2.h>

bboxx(xb,xe,yb,ye,MM1)
        UINT xb,xe,yb,ye;
        SEL MM1;
        {
        lineh(yb,xb,xe,MM1);                      /* top line */
        lineh(ye,xb,xe,MM1);                      /* bottom line */
        linev(xb,yb,ye,MM1);                      /* right line */
        linev(xe,yb,ye,MM1);                      /* left line */
        }
lineh(y,x1,x2,MM1)
        UINT y,x1,x2;
        SEL MM1;
        {
        UINT n;
        for(n = x1;n <= x2;n++)
           wdot(n,y,MM1);                         /* hor line */
        }
linev(x,y1,y2,MM1)
        UINT x,y1,y2;
        SEL MM1;
        {
        UINT n;
        for(n = y1;n <= y2;n++)
           wdot(x,n,MM1);                         /* vertical line */
        }
wdot(x,y,MM1)
        UINT x,y;
        SEL MM1;                                  /* x=col,y=row */
        {
        PCHAR ptr;
        UINT DM = 0x0000;
        CHAR MASK1 = 0x01;                        /* set dot */

        if(y & 0x01)
           DM = 0x2000;                           /* even buffer */
        ptr = MAKEP(MM1,DM+(80*(y >> 1) + (x >> 2)));   /* dot location */
        *ptr =(*ptr | (MASK1 << (2*(3 - x % 4))));      /* "OR" dot */
        }
uwdot(x,y,MM1)
        UINT x,y;
        SEL MM1;
        {
        PCHAR ptr;
        UINT DM = 0x0000;
        CHAR MASK1 = 0x00;                        /* clear dot */

        if(y & 0x01)
           DM = 0x2000;                           /* even buffer */
        ptr = MAKEP(MM1,DM+(80*(y >> 1) + (x >> 2)));   /* dot location */
        *ptr = (MASK1 << (2*(3 - x % 4)));        /* write undot */
        }
pltpt(x1,x2,y1,y2,MM1)
        float x1,x2,y1,y2;
        SEL MM1;
        {
        float m;                                  /* slope */
        int row;
        int col;
```

Figure 5.18 The file gphrout.c, containing the expanded line plot routines (and the line unplot routines).

```
/* Function to plot 3D facets: coordinates for CGA -- facet3d.c*/

#define INCL_BASE
#include <os2.h>
#include <stdio.h>

threeD_facets(m1,MM1)
        int m1;                                         /* index */
        SEL MM1;
        {
        extern int count;
        extern float xarray[],x,y,z,scaley = 125.,scalez = 75.;
        float z1,z2,y1,y2;
        int n,nc;
        float z_start = 25.,y_start = 25.,z_mid = 75.,y_mid = 125.;
        float dy1,dy2,dz1,dz2,xa[5],ya[5],za[5];

        nc = 3*count;                                   /* next y-value */
        xa[1] = xarray[m1];                             /* 1st y-value grid */
        ya[1] = xarray[m1+1];
        za[1] = xarray[m1+2];
        xa[2] = xarray[m1+3];
        ya[2] = xarray[m1+4];
        za[2] = xarray[m1+5];

        xa[3] = xarray[m1+nc+3];                        /* 2nd y-value grid */
        ya[3] = xarray[m1+nc+4];
        za[3] = xarray[m1+nc+5];
        xa[4] = xarray[m1+nc];
        ya[4] = xarray[m1+nc+1];
        za[4] = xarray[m1+nc+2];

        dy1 = ya[2] - ya[1];                            /* ck rotation */
        dy2 = ya[3] - ya[2];
        dz1 = za[2] - za[1];
        dz2 = za[3] - za[2];
        if((dy1*dz2 - dz1*dy2) > 0)
           {
           for(n = 1;n <= 4;n++)                        /* scale facet */
              {
              za[n] = z_start + (z_mid - za[n]*scalez);
              ya[n] = y_start + (y_mid + ya[n]*scaley);
              }
           for(n = 1;n <= 3;n++)                        /* plot 3 of 4 */
              {
              y1 = ya[n];
              y2 = ya[n+1];
              z1 = za[n];
              z2 = za[n+1];

              pltpt(y1,y2,z1,z2,MM1);                   /* collapsed y-z */
              }
           y1 = ya[4];
           y2 = ya[1];
           z1 = za[4];
           z2 = za[1];
              pltpt(y1,y2,z1,z2,MM1);                      /* 4th segment */
              }
           }
```

Figure 5.17 The file facet3d.c, used to generate the individual surface facets.

```
/* Function to scale xarray data */

#include <stdio.h>

scale()
        {
        extern int ncount,mcount;
        extern float xarray[],scalex,scaley,scalez;
        int n,m,m1;
        float max_x = -1.e14,max_y = -1.e14,max_z = -1.e14;
        float min_x = 1.e14,min_y = 1.e14,min_z = 1.e14;

        m1 = 1;
        for(n = 1;n <= ncount;n++)
           {
           for(m = 1;m <= mcount;m++)
              {                                       /* max/min determine */
              if(max_x < xarray[m1])
                 max_x = xarray[m1];
              if(min_x > xarray[m1])
                 min_x = xarray[m1];
              if(max_y < xarray[m1+1])
                 max_y = xarray[m1+1];
              if(min_y > xarray[m1+1])
                 min_y = xarray[m1+1];
              if(max_z < xarray[m1+2])
                 max_z = xarray[m1+2];
              if(min_z > xarray[m1+2]);
                 min_z = xarray[m1+2];
              m1 = m1 + 3;                            /* next point set */
              }
           }
                                                      /* scale [-1,1] */
        scalex = 2./(max_x - min_x);
        scaley = 2./(max_y - min_y);
        scalez = 2./(max_z - min_z);
        m1 = 1;
        for(n = 1;n <= ncount;n++)
           {
           for(m = 1;m <= mcount;m++)
              {                                       /* normalize [1,-1] */
              xarray[m1] = -1. + scalex*(xarray[m1] - min_x);
              xarray[m1+1] = -1. + scaley*(xarray[m1+1] - min_y);
              xarray[m1+2] = -1. + scalez*(xarray[m1+2] - min_z);

             m1 = m1 + 3;
              }
           }
        }
```

Figure 5.16 The file xscale.c, used for scaling the array between (-1,+1) along all three axes.

tively, this tilts the viewing angle upward so that the observer is looking down at an angle toward the image. Figures 5.20 through 5.22 maintain α and γ at zero but change β to span 1.0, 1.2, and 1.4 radians, respectively. This progressively tilts the image toward the reader, as illustrated. Figure 5.23 presents the figure for $\alpha = \gamma = 0$ and $\beta = 1.0$ with $N = 12$ instead of $N = 7$ [see Equation (5.13)].

```
VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrtl,vio_hdll);
MM = PVBPrtl.asel[0];                           /* selector */
clrCGA(MM);                                     /* CGA clear */
VioScrUnLock(vio_hdll);                         /* unlock screen */
```

Figure 5.14 (*Concluded*)

```
/* Function to read xarray from disk */

#include <stdio.h>

xarray_diskrd()
        {
        extern float xarray[];
        int n,check,counter;
        FILE *infile;
        char FN2[81];

        printf("Input read database filename \n");
        gets(FN2);
        gets(FN2);

        if((infile = fopen(FN2,"r")) == NULL)
           {
           printf("Input file failure");
           exit(1);
           }
        fscanf(infile,"%d \n",&counter);
        for(n = 1;n <= counter;n++)
           fscanf(infile,"%f \n",&xarray[n]);
        if((check = fclose(infile)) != 0)
           {
           printf("Error on input file close");
           exit(1);
           }
        return(counter);
        }
```

Figure 5.15 The file xadiskr.c, for reading the disk file input to the surface plotting program.

and if the facet is to be plotted, further scaling from [–1.,1.] to screen coordinates is implemented. Here only the *y-z* plane is considered (the *x*-axis is collapsed).

Figure 5.18 contains the file gphrout.c used as a basis for the library cgraph.lib. These routines are called to access the screen buffer. Figure 5.19 illustrates the output with $\alpha = \gamma = 0$ (no rotation about the x and z axes). In this figure $\beta = 0.8$, which is a counterclockwise rotation of 0.8 radian about the y-axis. Effec-

```
        }

clrCGA(MMM)
        SEL MMM;
        {
        INT n;
        INT N1 = 0x1F3F;                            /* end odd buffer */
        INT DM = 0x2000;                            /* even offset */
        PCHAR ptr;                                  /* pointer scr buf */

        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MMM,n);                      /* odd far pointer */
           *ptr = 0;                                /* clear odd buffer */
           }
        for(n = 0;n <= N1;n++)
           {
           ptr = MAKEP(MMM,DM+n);                   /* even far pointer */
           *ptr = 0;                                /* clear even buffer */
           }
        }



/* Function to read xarray from disk */

#include <stdio.h>

xarray_diskrd()
        {
        extern float xarray[];
        int n,check,counter;
        FILE *infile;
        char FN2[81];

        printf("Input read database filename \n");
        gets(FN2);
        gets(FN2);

        if((infile = fopen(FN2,"r")) == NULL)
           {
           printf("Input file failure");
           exit(1);
           }
        fscanf(infile,"%d \n",&counter);
        for(n = 1;n <= counter;n++)
           fscanf(infile,"%f \n",&xarray[n]);
        if((check = fclose(infile)) != 0)
           {
           printf("Error on input file close");
           exit(1);
           }
        return(counter);
        }
                                                    /* set STD mode */
        VioSetMode(((struct _VIOMODEINFO far *)&STDm),vio_hdl);

        DosExit(action,error_code);
        }

cclsCGA(vio_hdl1)
        SHANDLE vio_hdl1;
        {
        SEL MM;
        UINT wait1 = 1;
        struct _VIOPHYSBUF PVBPrt1;                 /* physical buffer */

        PVBPrt1.pBuf = (BYTE far *)(0xB8000);       /* phys buf start */
        PVBPrt1.cb = 0x4000;                        /* buffer length */

        VioScrLock(wait1,(char far *)dstat,vio_hdl1); /* lock screen */
                                                    /* physical buffer */
```

Figure 5.14 (*Continued*)

```
        lkbd_buf.cb = 80;                              /* buffer size */

/*           -------------------------------------------
 *                     Setup Rotated Figure
 *           -------------------------------------------    */

        printf("Input ncount\n");                      /* x-axis count */
        scanf("%d",&ncount);
        Printf("\n Square array: ncount=mcount\n");
        mcount=ncount;                                 /* y-xais count */
        count=ncount;                                  /* grid shift */
                                                       /* Input rotations */
        printf("Input x-rotation (rad)\n");            /* x-rotation */
        scanf("%f",&alpha0);
        printf("Input y-rotation (rad)\n");            /* y-rotation */
        scanf("%f",&beta0);
        printf("Input z-rotation (rad)\n");            /* z-rotation */
        scanf("%f",&gamma0);

        rot_mat(alpha0,beta0,gamma0);                  /* loads global a[] */
        N = xarray_diskrd();                           /* disk values */

        m1 = 1;                                        /* 1st facet group */
        for(n = 1;n <= ncount;n++)
           {
           for(m = 1;m <= mcount;m++)
              {
              x = xarray[m1];                          /* x-input rotation */
              y = xarray[m1+1];                        /* y-input rotation */
              z = xarray[m1+2];                        /* z-input rotation */
              rot_point();                             /* rotate (x,y,z) */
              xarray[m1] = x;                          /* reload x */
              xarray[m1+1] = y;                        /* reload y */
              xarray[m1+2] = z;                        /* reload z */
              m1 = m1 + 3;                             /* inc index 3 */
              }
           }

/*           -------------------------------------------
 *                  Graphics Screen Access
 *           -------------------------------------------    */
        scale();                                       /* x,y,z-> [1,-1] */

                                                       /* set CGA mode */
        VioSetMode(((struct _VIOMODEINFO far *)&CGAm),vio_hdl);

        cclsCGA(vio_hdl);                              /* clear CGA screen */

        VioScrLock(wait2,(char far *)dstat1,vio_hdl); /* lock screen */
                                                       /* physical buffer */
        VioGetPhysBuf((struct _VIOPHYSBUF far *)&PVBPrt2,vio_hdl);

        MM1 = PVBPrt2.asel[0];                         /* selector */

        m1 = 1;
        nm_count = 3*ncount*mcount - (ncount*3 + 6);   /* adjust limit */
        for(n = 1;n <= ncount;n++)
           {
        for(m = 1;m <= mcount;m++)
           {
           if(m1 < nm_count)                        /* check facet count */
              threeD_facets(m1,MM1);                /* plot facets */
           m1 = m1 + 3;                             /* increment index */
           }
        }

     prtscr(MM1);                                   /* PrtSc routine */

     VioScrUnLock(vio_hdl);                         /* unlock screen */
                                                    /* hesitate screen */
     KbdStringIn((char far *)kbd_buf,
               ((struct _STRINGINBUF far *)&lkbd_buf),
               wait,kbd_hdl);
```

Figure 5.14 (*Continued*)

```
/* This routine sets & clears CGA mode with screen clear--mmain3d.c
*  The generalized nomenclature is used.
*  A 3D (sin(u)/u)**2 is plotted.
*  The routine calls gphrout.c graphics functions. */

#define INCL_BASE                                        /* Conditional load */
#include <os2.h>                                         /* OS2 includes */
#include <stdio.h>

struct _STRINGINBUF lkbd_buf;                            /* keyboard buf len */
CHAR kbd_buf[80];                                        /* keyboard buffer */

UINT action = 0;                                         /* end thread */
UINT error_code = 0;                                     /* result code */
UINT wait = 1;                                           /* reserved word */

CHAR dstat[1];                                           /* lock status */
CHAR dstat1[1];                                          /* lock status */

float xarray[3072],scalex,scaley,scalez,x,y,z,a[10];     /* needed globals */
int ncount,mcount,count;                                 /* x-y count max */

/*      --------------------------------------------------
*                    Print Screen Parameters
*       -------------------------------------------------- */
BYTE    coll[320];                                       /* raster line array */
BYTE    MM[4] = {0x40,0x10,0x04,0x01};                   /* byte mask */
BYTE    w[8] = {128,64,32,16,8,4,2,1};                   /* gross weight */
BYTE    s[4];                                            /* dummy */
BYTE    shiftl[4] = {6,4,2,0};                           /* byte pos. shift */
BYTE    in_buffer1[4] = {0x1B,0x4B,64,1};                /* location byte */
BYTE    in_buffer2[2] = {0x0D,0x0A};                     /* c.r & l.f. */
BYTE    in_buffer3[3] = {0x1B,0x41,8};                   /* escape sequence */
BYTE    in_buffer4[2] = {0x1B,0x32};                     /* line spacing */
BYTE    dev_name[5] = {'L','P','T','1',0};
/*      --------------------------------------------------- */

main()
        {
        extern prtscr();                                 /* PrtScr routine */

        SHANDLE vio_hdl = 0;                             /* video handle */
        SHANDLE kbd_hdl = 0;                             /* keyboard handle */
        UINT wait2 = 1,nnn;                              /* reserved */
        UINT xb = 75,xe = 150,yb = 25,ye = 175;          /* box points */
        SEL MM1;                                         /* selector */
        int n,m,m1,N,nm_count;                           /* plot variables */
        float alpha0,beta0,gamma0;                       /* direction cosines */

        struct _VIOPHYSBUF PVBPrt2;                      /* physical buffer */
        struct _VIOMODEINFO CGAm;                        /* CGA structure */
        struct _VIOMODEINFO STDm;                        /* 80 x 25 struct */

        PVBPrt2.pBuf = (BYTE far *)(0xB8000);            /* buffer start */
        PVBPrt2.cb = 0x4000;                             /* buffer size */

        CGAm.cb = 12;                                    /* struct length */
        CGAm.fbType = 7;                                 /* CGA mode */
        CGAm.color = 2;                                  /* CGA color */
        CGAm.col = 40;                                   /* text columns */
        CGAm.row = 25;                                   /* text rows */
        CGAm.hres = 320;                                 /* CGA hor res */
        CGAm.vres = 200;                                 /* CGA vert res */

        STDm.cb = 12;                                    /* struct length */
        STDm.fbType = 1;                                 /* 80 x 25 mode */
        STDm.color = 4;                                  /* STD color */
        STDm.col = 80;                                   /* text columns */
        STDm.row = 25;                                   /* text rows */
        STDm.hres = 720;                                 /* STD hor res */
        STDm.vres = 400;                                 /* STD vert res */
```

Figure 5.14 The program mmain3d.c, which is the main calling program for plotting and printing the three-dimensional surface.

```
/* Function to write xarray to disk */

#include <stdio.h>

xarray_diskwt(NCOUNT)
        int NCOUNT;                                     /* Number points */
        {
        int n,check;
        FILE *outfile;
        char FN1[81];
        extern float xarray[];

        printf("Input database filename\n");
        gets(FN1);
        gets(FN1);

        if((outfile = fopen(FN1,"w")) == NULL)
           {
           printf("Output file failure ");
           exit(1);
           }
        fprintf(outfile,"%d \n",NCOUNT);
        for(n = 1;n <= NCOUNT;n++)
           fprintf(outfile,"%f \n",xarray[n]);
        if((check = fclose(outfile)) != 0)
           {
           printf("Error on output file close");
           exit(1);
           }
        }
```

Figure 5.12 The file xadiskw.c, which generates a Protected Mode disk write using reentrant library routines.

```
mmain3d.obj: mmain3d.c
        cl -c -Zi mmain3d.c

facet3d.obj: facet3d.c
        cl -c -Zi facet3d.c

xscale.obj: xscale.c
        cl -c -Zi xscale.c

rotmat.obj: rotmat.c
        cl -c -Zi rotmat.c

rotpt.obj: rotpt.c
        cl -c -Zi rotpt.c

xadiskr.obj: xadiskr.c
        cl -c -Zi xadiskr.c

pprtscr.obj: pprtscr.c
        cl -c -Zi pprtscr.c

mmain3d.exe: mmain3d.obj xscale.obj facet3d.obj pprtscr.obj\
                rotmat.obj rotpt.obj xadiskr.obj cgraph.lib
        link /CO mmain3d+xscale+facet3d+rotmat+rotpt+pprtscr+\
                xadiskr,,,cgraph,,
```

Figure 5.13 The MAKE file for the program that plots a three-dimensional surface based on an input data file.

```
/* generate 3d surface */

#include <math.h>

float xarray[2000],x[500],y[500],z[500];

main()
        {
        int n,m,ncount = 21,mcount = 21,m1,m2,N,NN;
        float A= 10., error = 1.e-5;
        double PI = 3.141592654,u,v;

        printf("Input interval divider\n");
        scanf("%d",&NN);

        m2 = 1;
        m1 = 1;
        for(n = 1;n <= ncount;n++)
           {
           for(m = 1;m <= mcount;m++)
              {
              x[m2] = (float)(m - mcount + 10);
              y[m2] = (float)(n - ncount + 10);
              u = (double)(x[m2]);
              v = (double)(y[m2]);
              u = (double)((PI/NN)*sqrt(u*u + v*v));

              if((u < error) && (u > -error))
                 z[m2] = A;
              else
                 z[m2] = A*(sin(u)/u);

              z[m2] = z[m2]*z[m2];
              xarray[m1] = x[m2];
              xarray[m1+1] = y[m2];
              xarray[m1+2] = z[m2];
              m2++;
              m1 = m1 + 3;
              }
           }
           N = m1-1;
           xarray_diskwt(N);
        }
```

Figure 5.11 The program gen3d.c, which generates the surface data file.

trates the disk write function. Figure 5.13 contains the MAKE file for the program that plots the three-dimensional surface.

Figure 5.14 presents a main calling function for the program that plots the three-dimensional surface input using xarray_diskrd(), which is contained in Figure 5.15. The function threeD_graph() reads rotation angles for locating the observer and rotates the input points. These functions, rotmat() and rotpt(), have been mentioned previously. Next, the data is scaled using scale(), which appears in Figure 5.16. The function scale() simply ensures that all x, y, and z values lie within the interval [–1.,1.].

After scaling the data, threeD_graph() clears the screen, sets CGA display mode, and plots the facets using threeD_facet(). This function appears in Figure 5.17. The routine threeD_facet() first loads the arrays xa[], ya[], and za[] with each of the four vertex points on the facet. A check for a hidden-line condition is made, where the first row consists of Cartesian unit vectors. Since it is the x-axis term we are interested in, we examine

$$n_x = m_{iy} m_{(i+1)z} - m_{iz} m_{(i+1)y} \tag{5.10}$$

Here

$$\mathbf{m}_i = m_{ix}\,\hat{\mathbf{i}} + m_{iu}\,\hat{\mathbf{j}} + m_{iz}\,\hat{\mathbf{k}} \tag{5.11}$$

If

$$n_x < 0 \tag{5.12}$$

the facet contains hidden lines.

A Simple Mathematical Example

It is useful to create a simple example to illustrate a three-dimensional surface. Consider the function

$$z = A^2 \frac{\sin^2[(\pi\,/N)\sqrt{x^2 + y^2}\,]}{[(\pi\,/N)\sqrt{x^2 + y^2}\,]^2} \tag{5.13}$$

This function has the familiar $(\sin x/x)^2$ behavior. We note that in the limit $(x, y) = (0, 0)$ the result is

$$z = A^2 \tag{5.14}$$

It is useful to generate values for x and y in the range [–10.,10.] with N = an input value that is a measure of the range of z.

Figure 5.10 illustrates the MAKE file for a program gen3d.c, which creates

```
gen3d.obj: gen3d.c
        cl -c -Zi gen3d.c

xadiskw.obj: xadiskw.c
        cl -c -Zi xadiskw.c

gen3d.exe: gen3d.obj xadiskw.obj
        link gen3d.obj+xadiskw.obj,,,,
```

Figure 5.10 The MAKE file for the program that writes the data file for a three-dimensional $(\sin x\,/x)^2$ surface.

this surface data file. Figure 5.11 illustrates a main calling program that generates these values and writes them to disk. Initially, the total number of values (x, y, z) for each point on an x-y grid spaced at unity intervals in the range above, is written to disk followed by the points themselves in x, y, z order. This is an array and is defined as specified above. It is the array xarray[] in Figure 5.11. Figure 5.12 illus-

Connecting these points cyclically yields the surface facet. When collapsed along the x-axis we have the final points

$$\begin{aligned}
&1: \ (0, y_m, f(x_n, y_m))\\
&2: \ (0, y_{m+1}, f(x_n, y_{m+1}))\\
&3: \ (0, y_{m+1}, f(x_{n+1}, y_{m+1}))\\
&4: \ (0, y_m, f(x_{n+1}, y_m))
\end{aligned}$$

If these points are plotted with the y-axis corresponding to column values and the z-axis corresponding to row values, a surface representation will be displayed with facets outlined.

It is important to recognize that the surface described above will display all lines appearing in the facets. This includes "hidden lines," which are those lines appearing in facets whose view would normally be obstructed. This obstruction results from the fact that other facets are located in front of the facet in question when viewed in the chosen direction.

To avoid illustrating hidden lines, it is useful to delete plotting of facets containing these lines. Although there are several ways to eliminate these hidden line facets, a very simple procedure is to create a vector normal to the facet and ignore the facet if this vector has a negative component pointing into the screen. Since the x-axis is normal to the screen, this implies that a negative, x-component of this normal vector would denote a facet with hidden lines.

We can create this normal vector from any three points in the surface. Suppose that we have the vertices defined by vectors from the origin:

$$\begin{aligned}
\mathbf{p}_1 &= (x_1, y_1, z_1)\\
\mathbf{p}_2 &= (x_2, y_2, z_2)\\
\mathbf{p}_3 &= (x_3, y_3, z_3)
\end{aligned} \tag{5.6}$$

and cyclically define line segments

$$\begin{aligned}
\mathbf{m}_1 &= (\mathbf{p}_1 - \mathbf{p}_3)\\
\mathbf{m}_2 &= (\mathbf{p}_2 - \mathbf{p}_1)\\
\mathbf{m}_3 &= (\mathbf{p}_3 - \mathbf{p}_2)
\end{aligned} \tag{5.7}$$

Then a normal to the surface subtended by these three line segments is given by

$$\mathbf{n} = \mathbf{m}_i \ \mathrm{x} \ \mathbf{m}_{i+1} \quad (\mathbf{i} = 1, 2, 3) \tag{5.8}$$

In this equation i is cyclic (modulo 3). The vector product is defined by the determinant

$$n = \begin{vmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ m_{ix} & m_{iy} & m_{iz} \\ m_{(i+1)x} & m_{(i+1)y} & m_{(i+1)z} \end{vmatrix} \tag{5.9}$$

Here the sets $\{x_n\}$ and $\{y_m\}$ have been chosen to span the space of interest. The three–dimensional surface is then determined relative to this grid using Equation (5.3). We assume further that an observer is located at the point (x_p, y_p, z_p) which is achieved by a rotation (α, β, γ) about the x, y, and z axes, respectively. (Note that this rotation is *not* composed of orthogonal components.) This rotation was treated in some detail in Chapter 4 and the reader is referred to the routine rotmat() and rotpt() for a complete discussion.

With this formulation, then, we can generate an abstract three-dimensional space with the observer located at any point in the space. Following the rotation a new set of coordinates is defined by

$$\begin{pmatrix} x'_n \\ y'_m \\ f(x'_n, y'_m) \end{pmatrix} = R(\alpha, \beta, \gamma) \begin{pmatrix} x_n \\ y_m \\ f(x_n, y_m) \end{pmatrix} \qquad (5.5)$$

$R(\alpha, \beta, \gamma)$ is given by Table 4.1. To display this space it will be useful to collapse the x-axis once a suitable rotation has been achieved. The points plotted on this display will then be members of the set

$$\{(0, y'_m, f(x'_n, y'_m)) \colon n=1,2,\ldots,N;\ m=1,2,\ldots,M\}$$

The order for the display will be to let $\{y'_m\}$ correspond to column positions and $\{f(x'_n, y'_m)\}$ correspond to row positions.

One final concept is needed: the notion of a facet. Basically, for plotting purposes it is useful to break the surface into facets (or small localized areas). The methodology for achieving this (used here) is to consider a grid structure on the x-y plane and assume a facet to be bounded by each set of grid lines projected onto the surface. For example, if we consider a surface grid, it is clear that the four x-y plane grid points

$$\begin{aligned} &1\colon\ (x_n, y_m, 0) \\ &2\colon\ (x_n, y_m, 0) \\ &3\colon\ (x_{n+1}, y_{m+1}, 0) \\ &4\colon\ (x_{n+1}, y_m, 0) \end{aligned}$$

define the locations of the vertices of the grid. Lines connecting 1 and 2, 2 and 3, 3 and 4, and 4 and 1, respectively, define the grid. Projecting these lines onto the surface yields the surface points

$$\begin{aligned} &1\colon\ (x_n, y_m, f(x_n, y_m)) \\ &2\colon\ (x_n, y_{m+1}, f(x_n, y_{m+1})) \\ &3\colon\ (x_{n+1}, y_{m+1}, f(x_{n+1}, y_{m+1})) \\ &4\colon\ (x_{n+1}, y_m, f(x_{n+1}, y_m)) \end{aligned}$$

5.4 REEXAMINING THE CORE VERSUS PRESENTATION MANAGER API SERVICES

The core or basic API services are largely derivative from the OS/2 Version 1.0, where only keyboard (Kbd), mouse (Mou), video (Vio), and DOS (Dos) calls are available. These are the only services treated in this book. For those readers familiar with Microsoft's Windows environment, the PM presents a similar graphics-like interface. It is programmed in a fashion similar, but not identical, to Windows. Programming the PM requires a great deal of concentration and patience. This effort will be simplified greatly when additional object-oriented tools are developed. Petzold [11] has written a lengthy book on how to accomplish this Presentation Manager programming. The reader is cautioned that some differences exist between the PM described in Petzold's book, which is based on the Microsoft version, and the IBM version of the Presentation Manager, which was released after the Microsoft version. When accessing the graphical interface, for example, in the full command screen mode the Vio calls must be used. Under PM the Gpi function calls are used.

5.5 ADVANCED C EXAMPLE: A THREE-DIMENSIONAL SURFACE

In this section we present an analytical approach for describing three-dimensional surfaces within the framework of simple vector arithmetic. A technique for removing "hidden lines" is illustrated based on consideration of the rotating characteristics of facets. Here a facet is a member of a logical subdivision of the three-dimensional surface. We begin with a brief discussion of surface characterization.

Functions of Two Variables

It is convenient to denote a function of one variable using the notation

$$y = f(x) \tag{5.2}$$

Graphically, such a relationship is represented with a two-dimensional plot using the independent variable, x, along the horizontal axis and the dependent variable, y, along the vertical axis. When a function depends on two variables it is representable in a three-dimensional space defined by

$$z = f(x, y) \tag{5.3}$$

In displaying such data a third axis must somehow be represented on a two-dimensional surface, the display screen. We have seen that it is useful to assume three perpendicular (orthogonal) axes: an x-axis, a y-axis, and a z-axis. Points in this space are denoted by

$$(x, y, z) = (x, y, f(x, y)) \tag{5.4}$$

The geometry for a three-dimensional surface consists of a *grid* of x-y points.

$$\{(x_n, y_m, 0)\}: n = 1, 2, ..., N; m = 1, 2, ..., M \}$$

TABLE 5.3 *(Concluded)*

Type	Comments
	accessible either sequentially, randomly, or directly via indexes or keys.
Hashing	This is a structure technique in which an algorithm or function is used to generate an address of a data element from a key. Typical associated structures would be a hash list.
Heaps	Heaps are most easily described as binary tree structures possessing *order* and *shape*. Order, for example, might specify that the value at any node is less than or equal to the value at the children. Shape suggests the tree architecture.
Linked lists	These data structures are used as indexes to other structures and have an associated pointer index that points to a relative record in the primary list.
Priority queues	This is a set of elements arranged according to priority. When an element is added or deleted, we do so in accordance with assigned priority or associated rules.
Sparse arrays	These are data structures with many zero elements. They can be reduced significantly to smaller storage by using additional indexing arrays with an appropriate indexing algorithm.

5.3.3 API Return Values and Error Checking

When an API routine is called, it contains a return value in the AX register which is passed back to the calling routine. In general, if this AX or return value is zero, the call has been successful. If not, one of several possible error conditions may exist, depending on the value returned. The user has an option as to how to treat these calls.

As an example of a typical API error return processing, consider

```
...
error = DosGetPID(process IDs);
if (error!=0)
   {
   printf("Error on acquiring process ID");
   exit(1);
   }
...
```

The reader should feel free to insert his or her own error processing as appropriate following API service calls.

ized algorithms for handling large data organizations that require speed of access of optimized storage. Similarly, sparse array techniques minimize the amount of storage needed for multidimensional data.

TABLE 5.2 SOME TYPICAL ALGORITHMS

Type	Comments
Mathematical	Arithmetic, random numbers, interpolation, simultaneous equations, integration
Sorting	Exchange, bubble, quicksort, radix, priority queues, selection/merging
Searching	Sequential, binary, tree, hashing
String processing	Pattern matching, parsing, file compression
Geometric	Polygons, line intersection, convex surfaces, grids, closest point
Graph	Connectivity, mazes, shortest path, topological sorting, networks
Advanced	Systolic arrays, FFT, dynamic programming, linear programming

All these techniques are used in developing the area of data structures and database design. The interested reader is referred to reference 9 for specific details of large-scale implementations. In this book we will confine most of the discussion to the primitive structures listed in the beginning of Table 5.3.

TABLE 5.3 SOME REPRESENTATIVE DATA STRUCTURES

Type	Comments
Arrays	These structures consist of concatenated variables stored in a block and accessible via one or more indexes.
Bit strings	These structures constitute the basic building blocks of any language and are accessible in C by using the bitwise operators.
Bit maps	This is a mapping of a set of variables and their associated parameters onto a set of bits, which constitutes a smaller set of storage. All attributes of the variables are not represented in this fashion, and the mapping must be attribute specific.
Databases	Databases are complex data structures consisting of data items or fields collected into records that are

```
            increment++;
        }
    }
    ...
```

Now consider the alternative

```
    ...
    for  (n=1;n<=N;n++)
      {
      v[n]=v[n]*v[n]
      for (m=1;m<=M;m++)
          {
          if((v[n]>q[m])  &  (v[n]<q[m+1]))
          increment++;
          }
      }
    }
```

The second form is admittedly more cumbersome, although easier to understand. What about time criticality? In the first fragment the expression

```
v[n]  =  v[n]*v[n]
```

is executed NM times, while in the second fragment it executes only N times. The latter program fragment ensures an optimum time-critical compiled result. Although this example may appear academic, it is representative of the decisions regarding form that must be made.

Algorithm development The topic of this section is algorithms. Algorithms are structured approaches to solving mathematically, particular problems amenable to solutions. A more general definition would, of course, encompass most programming efforts. We have iterative and recursive programming and it is true that a fundamental technique in designing efficient solutions is the recursive method, because this approach builds on an earlier solution. Table 5.2 illustrates some typical algorithms as discussed by Sedgewick [8]. We have already seen examples of some of these algorithm techniques. In general, we will not address the complete class of problems covered by the table; the interested reader is referred to reference 8 for a complete discussion. It is important, however, to recognize that algorithms are what computer programs are all about. Problems amenable to algorithmic solution can easily be tailored to computers.

Data Structures We briefly consider the subject of data structures [9] (as opposed to file structures [10]). Table 5.3 illustrates some well-known data structures used in small- and large-scale program development. We have already seen arrays used as a basic element of the C language. Using the bitwise operators, it is possible to enter or extract information from data elements at the bit level. More complex data structures, such as hashed lists, heaps, linked lists, and priority queues, are special-

As part of style we need to consider templates. This topic is meant to cover the overall program structure. A general template is as follows:

Module 1 (main())

```
Documentation (comment describing module)
Preprocessor (include files, define directions, globals)
  main() (function definition)
```

Module 2 (function1(),...functionN())

```
Documentation (comment describing functions)
  function1()
...
  functionN()
```

Module 3 (function(N+1)(),...functionM())

```
Documentation (comment describing functions)
  function(N+1)()
...
  functionM()
...
```

Module L (functionQ(),...functionR())

```
Documentation (comment describing functions)
  functionQ()
...
  functionR()
```

Here

```
1<N<N+1<...<M<...<Q<...<R
```

and

```
2<3<...<L
```

Finally, we consider form. Good form consists of defining the optimum methodology for implementing an algorithm. Unfortunately, most algorithms are sufficiently complex that it is difficult to decide what the best way to implement the algorithm might be. The issue of form must be addressed in a somewhat simplistic fashion. Consider the following code fragment:

```
...
for (n=1;n<=N;n++)
 {
 for (m=1;m<=M;m++)
   {
   v[n]=v[n]*v[n];
   if((v[n]>q[m]) & (v[n]<q[m+1]))
```

5.3.2 Templates, Style, and Form

Style is a somewhat elusive feature of programming which reflects individual thought patterns as much as any organized approach to program development. Consider, for example, the problem of variable definition, mentioned earlier. The following is easily understood:

```
mortgage_int = loan_principal * interest_rate.
```

What about the following?

```
  Instantaneous_amp = exp(-time/delay_factor)*
/        cos(2. * pi * frequency * time)
```

For technically inclined users, the following is much easier:

```
A = exp(-t/tau) * cos(2. * pi * f * t)
```

(Those who are not technically trained probably will not care about such details.)

Programmers with a background in FORTRAN, ALGOL, or original BASIC are familiar with restrictions on the length of a variable name. They tend to be more cryptic than programmers of more recent vintage, who are used to 32-character limits. This is decidedly a learned style feature. More important is the need to clarify variable meaning. If the programmer provides a design document, which is essential for a clear understanding of the program, each variable should be delineated in an unambiguous fashion. In cumbersome assignments, spelling out each variable name in a wordy fashion can often obscure the meaning of the underlying relationship. Similarly, by being too cryptic or obscure, the meaning of the equality can escape the reader.

An additional feature of style is the nature of actual code reduction. In other words, is the code compact, or can each relationship be followed in easily readable form? The code

```
if(((x1==c1) | (x2==+c2)) & ((x3==c3) | (x4==c4)))
    A1;
```

is compact, the following is slightly easier to follow:

```
if((x1==c1) | (x2==c2))
   {
   if((x3==c3) | (x4==c4))
      {
      A1;
      }
   }
```

(If the reader has doubts as to which is easier, try assigning values and working out the truth table.)

structures the use of exception handling must be clarified. Consider the if... statement in the following form:

```
...
if(check==0)
  {
  printf ("Denominator zero");
  exit(1);
  }
...
```

Here the if statement specifically looks for an error condition and prints the message explaining this condition prior to exit. The exception handling is part of the if purpose. A sequence of the form

```
...
for (n=1; n<=N; n++)
    {
    x=x+a;
    if (x==NTOTAL)
       {
       printf ("x max exceeded");
       exit(1);
       }
    a=a+b;
    }
```

is less desirable because the causative factors (a, b, and x too large) are unclear at time of the exit. In general, we use the unconditional exit only [exit (1)] and we only use this exit when the complete nature of the error condition is absolutely clear. Also, it is used subject to the constraints indicated above. A typical use is

```
...
if ((check=fopen (FN1,"r"))==NULL)
   {
   printf ("Error on read file open");
   exit(1);
   }
...
```

To some extent this discussion is semantic. Exits from within loops should not occur. Since C does not allow branching or jumps (as in assembler, FORTRAN, and other languages), there is really no way to exit a loop except with an exception handler or a goto. We have virtually ruled out goto statements, so only exception handling remains. The latter exit should be avoided within a loop structure. Flag the condition and upon exit from the loop report all relevant parameter data prior to exit. The latter exit should be accomplished using a conditional structure that specifically tests the exception status.

This artifact illustrates what functional activities are subordinate to other activities. At a glance, the user can glean overall functional program relationships from the program Structure Chart. This entity should be developed as the first component of program design. Next, the execution flow must be developed. The functional flowchart serves as a convenient vehicle for accomplishing this.

Briefly, there are a number of types of flowcharts. They vary from more functionally oriented descriptions of program behavior, in which generalized activity is interconnected, to very detailed descriptions where each line of code literally occupies a place in the overall flowchart. In this book we favor the functional approach because it is a good compromise: It gives the user a sense of the program execution and does not require pages of description.

As an alternative to the flowchart, pseudo-code can be used. Pseudo-code has the advantage that it is very close to the actual program mechanics and has the structure of natural language. With pseudo-code the uninitiated can develop a feeling for program execution while not fully comprehending the language syntax in which the program is written. A number of authors of high-level languages are developing program design languages (PDLs) which are essentially pseudo-code. The PDL approach to program design is particularly appropriate for very large-scale program development that may require significant development time. In this case a need exists to have an easily understood design document that can be made available to new programming team members. For most microcomputer applications, however, the flowchart is quite suitable.

Actual module implementation within the confines of structured code employs sequential and control statements as discussed above. The module should have one entry and one exit path (with exception control handled so as to delineate the error condition). Finally, each module should be documented to explain its function and what data structures exist, where needed. The relationship of one module to another can be delineated in special cases where it is not explained in an associated flowchart or Structure Chart.

The purpose of this short discussion is to reemphasize the importance of structured programming in the C language by briefly illustrating several features of the language. Also, we discuss some philosophical implications for structured code.We begin with the one entry and one exit precept applied to module definition. This control mechanism can be extended backward to the architectural structures mentioned earlier. Next we discuss the implications of style and form. (Also, a brief look at templates is treated as part of style.) Finally, a general look at algorithm development is used to round out the discussion. Also, data structures are treated.

The control and loop structures used in the C language are designed to provide one entry and one exit to modules. This ensures that control flow is linear through the structure (if, if...else, ..., while, do...while, etc.). Also, exit conditions are clearly made available to the user during execution. With regard to iterative structures such as loops, it is clear that multiple exits can be disastrous because the user may never learn about the state of the system that causes the exit condition. To jump outside a loop that is undergoing normal execution is highly undesirable. With conditional

An example of point 9 would be where very simple "bookkeeping" is involved in a module for clarity and maintenance purposes. Also, the use of globals does not appreciably increase the level of difficulty of a *small program* but can significantly reduce the size of variable handling code (particularly when pointers are used as an alternative).

The major difficulty with software development is not in determining how to make the computer function to execute a program but rather, in ensuring that a given program actually generates the output it was intended to generate. The emphasis here is on software integrity, with the presumption made that the programmer will learn the mechanics of programming within a particular language. To simplify program design and development, structured programming techniques evolved. Dijkstra [7] defined the initial concept, and structured programming is now a well-established discipline which has greatly affected the C architecture. The notion of structured programming in the broadest sense encompasses top-down design and modular programming. At a more localized level, structured programming focuses on coding techniques intended to simplify program understanding and facilitate program use (such as program modification and maintenance). In the following discussion we focus on the latter area: structured code.

The most desirable structure concept is *sequentially* defined code. In this instance instructions are executed as they are encountered. C provides for two deviations from this approach: conditional execution and iterative loops. We have seen examples of both of these conditions. Conditional execution included use of the forms

1. if...
2. if...else
3. the conditional operator
4. switch...case...break

Similarly, iterative loops utilize the forms

1. while...
2. do...while
3. for...

These two groups of statements are the most important control mechanisms in the C language. They form the basis of C structured coding techniques. Although C allows the goto... statement, it is discouraged and appropriate only in very extenuating circumstances. It should be argued forcefully that any code segment using a goto can be rewritten to avoid this statement. The major difficulty with goto statements, as pointed out by Dijkstra, is that unrestrained branching within a module can take place. This can lead to difficulty in understanding the intent of the code.

Structured code begins with a hierarchical description in the Structure Chart.

pendent thread has a particular piece of code that must execute prior to any other operation for the parent process. Then it would be desirable to monitor and ensure that this code executed, before continuing. Clearly, this could be dynamic and change with the active chronology of execution.

2.17. DosExit is used to terminate an application and return to OS/2. All other returns NEAR or F.

Chapter 3

3.1. The drivers mentioned operate from the kernel, level 0. They must originate here because they have to be protected ahead of all other code. We cannot have a disk-write preempted in the middle, nor can we tolerate "jerky" mouse cursor movement as the mouse position changes.

3.2. The macro calls admittedly remove a layer of detail from the program code. This layer would tend to expand the code by a factor of 4 to 7. All the pushes to the stack have been suppressed prior to each API call and the call takes on the form of a higher-level-language (HLL) function call. The data area tends to expand considerably with all the macro parameter definitions, but the actual executable code remains compact. This requires the programmer to develop a general familiarity with the macro calls at the level of the IBM Programmer's Toolkit or Appendix C of this book. Once this familiarity has developed it is a very easy matter to read the resulting "structured" code and follow the flow of execution. Hence maintenance becomes an easy task. Clarity (of how the code executes) is also paramount, and much more so under the macro call format. The macro calls do, however, inhibit debugging in that the in-line code is missing. If the user prints a copy of the list file with macros expanded, tracing the source code is still an easy matter. In general, these approaches tend to be a matter of preference based on the programmer's orientation. We favor the HLL appearance of the code. It makes functional performance of the code the primary mechanism to be emphasized. Expansion of the in-line code makes it more obscure from a functional viewpoint but easier (and essential) to debug.

3.3. For the segment to be sharable, bit 0, to be sharable through @DosGiveSeg, or bit 1, to be sharable through @DosGetSeg, must be set in the flags word (the third parameter in the calling list). Bit 2 of this same flags word must be set if the segment is to be discardable.

3.4. The write to the huge segment must use the proper selector. When crossing the 64K-byte boundary the program must access a new but contiguous selector.

3.5. There must be some common link between the two processes. Usually, this is a common element name such as

```
\SEM\SDAT.DAT
```

or

```
\QUEUES\QDAT.DAT
```

which appears in both processes and is the same. The system then provides the connection. Alternative to this is the passing of a selector or printer in a common

2.7. ...

```
kbd_buf          db       80
lkbd_buf         dw       $-kbd_buf
iowait           dw       0
kbdhdl           equ      0
freq             dw       1000
dur              dw       5000
...
@KbdStringIn     kbd_buf,lkbd_buf,iowait,kbdhdl
@DosBeep         freq,dur
...
```

2.8. Yes, all calls to the API can be made in full form, where each push and pop, as well as EXTRN declaration, is stated explicitly according to the rules of OS/2. The toolkit simply provided a set of assembler .inc files and C .h files that facilitated usage of the API services through very functional macros.

2.9. The key assumption is that segment selectors can be treated as segment addresses. Since the 80286 accesses segments using the selectors, the selector value must reside in a segment register. The address is then calculated in the usual Protected Mode fashion, where the segment selector acts as a segment address. The use of segment override addressing, such as

```
es:[bp]
```

simply permits specification of an address in the usual fashion, where the segment selector is made to correspond to the physical segment address when VioGetPhysBuf is exercised.

2.10. They represent FAR locations because the entry points are called from external API modules, hence a 32-bit address must be specified.

2.11. No hierarchy should have a single child subordinate to a parent. The box 310 should be absorbed in 300.

2.12. The command is

```
@DosWrite  dev_hand,in_buffer5,bytesin3,bytesout
```

where the undefined parameter is

```
in_buffer5  db  1BH,41H,0CH
```

2.13. It is intuitive that they cannot be preempted by an OS/2 task switch, or the possibility of losing data from the device would occur.

2.14. To access the screen buffer (physical) properly, the screen must be locked; hence if scr_ld is to load scr_buffer with the screen context, it must be locked. If prtscr is executed when the screen is locked, it could dominate access time for the physical display buffer. Hence the program should load a temporary buffer, release the screen context, and then begin the print operation.

2.15. Ten complete raster segments.

2.16. The DosExitCritSec corresponds to exit of a critical section of execution for a thread and returns control to a process. This could be used, for example, when an inde-

2.2.
```
@VioGetPhysBuf          macro       dstruc,rsrvd
                        @define     VIOGETPHYSBUF
                        @pushs      dstruc
                        @pushw      rsrvd
                        call        far ptr  VIOGETPHYSBUF
                        endm
```

2.3.
```
@VioScrLock             macro       wait,status,handle
                        @define     VIOSCRLOCK
                        @pushw      wait
                        @pushw      status
                        @pushw      handle
                        call        far ptr  VIOSCRLOCK
                        endm
```

2.4. The VioGetPhysBuf structure, PVBPtr1, needs to be specified as
```
...
PVBPtr1                 label       FAR
bufst1                  dd          0A0000H
buflen1                 dd          6D60H
physel1                 dw          0
...
```

2.5.
```
...
freq                    dw          5000
dur                     dw          1000
...
@DosBeep                            freq,dur
...
```

2.6.
```
...
waitf                   equ1
dstat                   db  ?
viohdl                  equ  0
PVBPtr1                 label       FAR
bufst1                  dd          0B8000H
buflen1                 dd          4000H
physel1                 dw          0
...

@VioScrLock             waitf,dstatk,viohdl
@VioGetPhysBuf          PVBPtr1,viohdl
push  physel1
pop  es
mov  dh,25
mov,  dl,154
mov  bx,75
mov  cx,235
call  conn12
@VioScrUnLock           viohdl
...
```

1.10. This occurs as a result of IBM's reservation of the last 384 KB of address space for special-purpose system memory, much of which is either screen buffer memory or ROM (read-only memory). OS/2 extended memory resides from 1 MB to 16 MB (in the physical address space).

1.11. Physical memory occupies the actual hardware locations accessed by the 24 pins from the address lines of the 80286 CPU chip. This can be a maximum of 16 MB (2^{24}). Virtual memory is memory allocated in an abstract sense by the system. Since there are 16,384 (2^{14}) possible selectors with 65,536 locations per selector, there are a maximum total of 1,073,741,824 possible locations that can be addressed uniquely in this virtual space. OS/2 manages this space by mapping each segment selector to a segment base address through manipulation of the translation registers. Hence, if the available physical memory is less than 16 MB, for example, and the required program and data memory exceed 16 MB or the actual physical memory, OS/2 will move code and data to and from disk as needed.

1.12. The Table Indicator (TI) bit in the segment selector is set for references to system memory. It is zero for references to local application program memory. There are 536,870,912 locations accessible by applications in virtual memory.

1.13. If the data communications service resided at level 0, it could preempt the CPU during long sessions which would mask out other, potentially more important interrupts. This could lead to catastrophic failure.

1.14. The boot record loads the Machine Status Word register with a MSW that has bit zero set for Protected Mode operation. Subsequent loads of this register using the LMSW instruction can modify this state.

1.15. **1.** Initialization Routine—Assembler
2. Strategy Routine—C
3. Interrupt Service Routine—Assembler

1.16. A pipe references a bulk memory area, whereas a queue allows access to individual members of the queue.

1.17. The API framework is more cumbersome for assembly language programs where software interrupts (using INT) provide immediate low-level access to the system hardware, for example. On the other hand, the API call orientation allows a rather elegant description of the services, which can enhance understanding and readability. In the C environment the API services are an asset, providing very readable function-like access to system services.

1.18. In both cases the threads must synchronize their access.

1.19. DosSleep. Unlike a process, the thread cannot terminate itself. Threads can be terminated only when the thread's parent process is terminated.

1.20. No, the Gpi services can be used only with the PM.

1.21. Modaless because the screen context is not preempted.

Chapter 2

2.1. (a) pins 8, 6, 4, 3, and 2
(b) pins 8, 7, 4, 2, and 1
(c) pins 7, 5, 3, and 1

```
                extrn         callname:far
                endif
                endm
```

1.5. @VioScrUnLock

```
@VioScrUnLock   macro         handle
                @define       VIOSCRUNLOCK
                @pushw        handle
                call          far  ptr  VIOSCRUNLOCK
                endum
```

where

```
@pushw          macro         parm
                mov           ax,parm
                push          ax
                endm
```

and

```
@define         macro         callname
                ifndef        callname
                extrn         callname:far
                endif
                endm
```

1.6. @VioScrLock

```
@VioScrLock     macro         wait,status,handle
                @define       VIOSCRLOCK
                @pushw        wait
                @pushs        status
                @pushw        handle
                call          far  ptr  VIOSCRLOCK
                endm
```

where @define and @pushw are defined as in the answers to Problems 1.4 and 1.5 and

```
@pushs          macro         parm
                mov           ax,SEG  parm
                push          ax
                lea           ax,parm
                push          ax
                endm
```

1.7. 1. A multitasking environment
2. A memory management facility
3. The PM user-friendly interface

1.8. Since we are talking about applications code, the non-system-oriented instruction set is applicable, and this is generally common to both CPUs with few exceptions. The major drawback to 80386 code is the use of references to the extended register set (32-bit registers): EAX, EBX, ECX, ... While the 80286 general-purpose registers (AX, BX, CX, ...) are a subset of these extended registers, the converse is not true.

1.9. 2

Answers to Problems

Chapter 1

1.1. 81CA H

1.2. No, while OS/2 employs time slicing to share access to a single CPU among multiple separate tasks or threads, it is not designed to service more than one CPU. Hence OS/2 does not provide for the parallel operation of multiple CPUs.

1.3. 4, 294, 967, 295 ($2^{32}-1$); +2, 147, 483, 647 ($2^{31}-1$)

1.4.
```
@DosExit        macro       action,result
                @define     DOSEXIT
                @pushw      action
                @pushw      result
                call        far ptr  DOSEXIT
                endm
```
where
```
@pushw          macro       parm
                mov         ax,parm
                push        ax
                endm
```
and
```
@define         macro       callname
                ifndef      callname
```

TABLE E.2 *(Concluded)*

Service	Description
MouGetDevStatus	Getting the state of the pointer and the event queue
MouReadEventQue	Reading a data record from the event queue
MouGetNumQueEl	Determining the number of records in the queue
MouFlushQue	Clearing the queue
MouRemovePtr	Defining a restricted screen area where the pointer is not allowed to appear
MouFlushQue	Clearing the queue
MouRemovePtr	Defining a restricted screen area where the pointer is not allowed to appear
MouDrawPtr	Redefining a restricted screen area, where the pointer is allowed to appear
MouGetPtrShape/ MouSetPtrShape	Getting or setting the shape of the pointer
MouGetPtrPos/ MouSetPtrPos	Getting or setting the vertical and horizontal positions of the pointer
MouRegister	Registering another mouse subsystem for the current session
MouDeRegister	Canceling the registration of a mouse subsystem
MouSynch	Synchronizing access for a mouse subsystem with the mouse device driver

REFERENCE

1. *IBM Operating System/2 Programmer's Toolkit,* International Business Machines Corporation, Boca Raton, FL, 1987.

TABLE E.1 *(Concluded)*

Service	Description
KbdFlush	Clears the keyboard input buffer of all queued keystrokes
KbdOpen/kbdClose	Opening and closing a handle to a secondary logical keyboard
KbdGetFocus/ KbdFreeFocus	Getting and releasing the input focus by the secondary keyboard
KbdGetStatus/ KbdSetStatus	Getting and setting the keyboard state
KbdGetCP/ KbdSetCP	Getting and setting the ID of the system code page used to translate scan codes into ASCII codes
KbdSetCustXt	Installing a customized keyboard translation table
KbdRegister	Registering a keyboard subsystem for the current session
KbdDeRegister	Canceling the registration of a keyboard subsystem
KbdSynch	Synchronizing the subsystem's access to the physical keyboard

E.3 THE MOUSE SERVICES

Table E.2 indicates the mouse API services. Again, these services are available through the macros and functions defined in the IBM Toolkit.

TABLE E.2 THE MOUSE FUNCTION CALLS

Service	Description
MouOpen	Initializes the mouse event queue and obtains a handle to access it
MouClose	Closes the mouse device for the current session and removes the mouse device driver handle from the list of valid open mouse device handles
MouGetNumButtons	Determining the number of buttons supported
MouGetEventMask/ MouSetEventMask	Getting or setting the types of events reported by data records
MouSetDevStatus	Setting mouse data to be returned in mickeys instead of coordinates (a mickey is a unit of measurement for physical mouse motion, whose value depends on the mouse device driver currently loaded)
MouGetNumMickeys	Determining the number of mouse motion units per centimeter
MouGetScaleFact/ MouSetScaleFact	Getting or setting the mickey-to-pel ratio for mouse motion

BSE.H
This file allows the user to set up defined symbols such as all the OS/2 API kernel functions, including those that begin with Dos, kbd, Vio, and Mou. It includes loading of BSEDOS.H, BSESUB.H, and BSEERR.H.

Level 3: OS/2
BSEDOS.H
This file sets up constants, structures, and function prototypes for the Dos services.
BSESUB.H
This file sets up constants, structures, and function prototypes for the Vio, kbd, and Mou services.
BSEERR.H
This file sets up error code constants for the OS/2 kernel API services.

The remaining .h files are Presentation Manager files and are only available under the OS/2 1.1 or higher. The .inc files are defined as follows:

Level 1: SYSMAC.INC
This file sets up all the API macros by calling DOSCALLS.INC and SUBCALLS.INC.

Level 2: OS/2 kernel
DOSCALLS.INC
This file sets up all macros for Dos calls.
SUBCALLS.INC
This file sets up all macros for kbd, Mou, and Vio calls.

E.2 THE KEYBOARD SERVICES

Table E.1 illustrates the keyboard services. These are available with the IBM OS/2 Toolkit referenced throughout the book.

TABLE E.1 KEYBOARD FUNCTION CALLS

Service	Description
KbdStringIn	Reads a string from the keyboard and loads a buffer
KbdCharIn	Reads a character and loads the associated internal structure
KbdPeek	Allows examination of a character data record without removing it from the buffer
KbdXlate	Translates a scancode and shift key state into an ASCII character code, using the code page set for the keyboard

E Keyboard and Mouse Kernel Functions

In this book we describe the OS/2 Kernel API services which are used to access the full-screen mode. In this appendix we discuss the keyboard and mouse services. These calls cannot be used in a Presentation Manager application.

E.1 ACCESSING THE TOOLKIT

To employ the Toolkit functions [1] the programmer must resort to defining and using various assembler and C structures and databases that contain parameter information. These structures and data types are contained in a set of files, with extension .inc for the assembler and a set of files with extension .h for the C compiler. The .h files are defined as follows (we use .h and .H interchangeably):

Level 1: OS/2 and Presentation Manager
OS2.H (includes OS2def.H, BSE.H, and PM.H)
This file sets up the compiler for access to all OS/2 definitions, base include files, and files needed to define Presentation Manager data types, functions, and structures.

Level 2: OS/2
OS2DEF.H
This file defines common constants, data types, error codes, and structures needed to OS/2 kernel access.

TABLE D.1 *(Concluded)*

Program	Description	Page
dyn22.def	Definition file for load on call	236
dyn2.asm	Assembler program for run-time DLL	238
dyn33.def	Definition file for dyn2.asm	239
gen3d.c	C program that generates a surface	255
xadiskw.c	Diskwrite	256
mmain3d.c	Main calling program for 3D surface	257
xadiskr.c	Diskread	260
xscale.c	Scales array for mmain3d.c	261
facet3d.c	Generates facets for mmain3d.c	262
gphrout.c	Plot routines	263

Table D.2 MAKE Files Used in Text

Program	Description	Page
ioprgm.mak	MAKE file for ioprgm.c	173
swave.mak	MAKE file for swave.c	178
prtwave.mak	MAKE file for prtwave.c	184
pipestc.mak	MAKE file for pipestc.c	197
pipeclc.mak	MAKE file for pipeclc.c	197
ckthred.mak	MAKE file for ckthred.c	198
dja.mak	MAKE file for dja.c	212
gen3d.mak	MAKE file for gen3d.c	254
mmain3d.mak	MAKE file for mmain3d.c	256

TABLE D.1 *(Continued)*

Program	Description	Page
prtscrm.asm	Modified prtscr.asm used with twolnm.asm	101
nos2512.asm	Creates shared segment, child process, and prints screen	106
nos261.asm	Child process that generates random numbers	111
nos252.asm	Supplemental routines needed by nos2512.asm	113
memseg.asm	Program that creates and reallocates memory	117
hugeseg.asm	Program that allocates a huge segment	122
suballo.asm	Program that suballocates memory	126
ckth1.asm	Program that sets up two threads using RAM semaphores	132
unos251.asm	Uses multiple threads to generate a box	136
nnos252.asm	Support routines for unos251.asm	141
ckpr1.asm	Program that sets up two processes using system semaphores	146
os2p2.asm	Child process using system semaphores	149
pipest.asm	Pipe main setup program	152
pipecl.asm	Child process for pipe communications	155
queuest.asm	Queue main setup program	158
queuecl.asm	Child process for queue example	161
ioprgm.c	C program to illustrate Protected Mode	172
swave.c	C program to plot dynamic sinewave	180
gphrout.c	C graphic routines used in cgraph.lib	183
prtwave.c	C program to print sinewave	186
pprtscr.c	C program to print screen	190
pipestc.c	C program counterpart to pipest.asm	194
pipeclc.c	C program counterpart to pipecl.asm	196
ckthred.c	C program that creates a child thread	199
tetra.c	C program for rotating tetrahedron	205
rotetra.c	C program that sets up tetrahedron	208
rotmat.c	C program that calculates rotation matrices	209
rotpt.c	C program that rotates a point	209
DMApoint.c	C program that removes a point from the display	211
timhist.c	C program that creates time-history/value database	212
dja.c	C program that plots Dow Jones activity	213
scales.c	C program to generate musical scales	225
scales1.asm	Assembler routine to generate scales	226
dyn1.asm	Assembler program for preloaded DLL routines	232
dlink1.asm	Assembler routine that is DLL for dyn1.asm	234
dyninit.asm	Initialization routine for DLL	235

D Programs Used in This Book

In this appendix we list the programs used in this book. Table D.1 presents each program, a brief description, and the page number corresponding to the program. Table D.2 contains the MAKE files that appear in the text.

TABLE D.1 PROGRAMS CONTAINED IN THE TEXT

Program	Description	Page
ptr2.asm	Assembler program to print "74" to generate line	46
boxprt1.asm	Assembler program to plot two lines to display	56
scrld.asm	Assembler procedure to load screen buffer	62
prtscr.asm	Assembler procedure to print the screen	65
twoln.asm	Assembler procedure to plot/print two lines	69
graph1.asm	Partial contents of GRAPHLIB.LIB	72
connl2.asm	Procedure to plot connected line	76
slopeln.asm	Program to plot connecting line	78
bbox1.asm	Procedure to generate a box	82
llinev.asm	Procedure to generate a vertical line	83
bbox.asm	Program to plot/print box	84
twolnm.asm	Modified twoln.asm that creates a screen buffer	97
scrldm.asm	Modified scrld.asm used with twolnm.asm	100

```
@VioScrLock      macro   m1, m2, m3
                 @def    VIOSCRLOCK
                 @pw     m1                        ;waitflag
                 @ps     m2                        ;status
                 @pw     m3                        ;handle
                 call    far ptr VIOSCRLOCK
                 endm
```

use: Figure 2.3, 2.7b, 2.8, 2.10, 2.15, 3.1, 3.6, 3.18b, 3.19

```
@VioScrUnLock    macro   m1
                 @def    VIOSCRUNLOCK
                 @pw     m1                        ;selector
                 call    far ptr VIOSCRUNLOCK
                 endm
```

use: Figure 2.3, 2.7b, 2.8, 2.10, 2.15, 3.1, 3.4, 3.6, 3.18b, 3.19

```
@VioScrollUp     macro   m1, m2, m3, m4, m5, m6, m7
                 @def    VIOSCROLLUP
                 @pw     m1                        ;top
                 @pw     m2                        ;left
                 @pw     m3                        ;bottom
                 @pw     m4                        ;right
                 @pw     m5                        ;number lines
                 @ps     m6                        ;attribute
                 @pw     m7                        ;handle
                 call    far ptr VIOSCROLLUP
                 endm
```

use: Figure 2.3, 2.8, 3.6, 3.17b, 3.18b, 3.20b, 3.22b

```
@VioSetMode      macro   m1, m2
                 @def    VIOSETMODE
                 @ps     m1                        ;modedata
                 @pw     m2                        ;handle
                 call    far ptr VIOSETMODE
                 endm
```

use: Figure 2.3, 2.7b, 2.10, 2.15, 3.1, 3.4, 3.18b

```
@VioWrtTTY       macro   m1, m2, m3
                 @def    VIOWRTTTY
                 @ps     m1                        ;charstr
                 @pw     m2                        ;length
                 @pw     m3                        ;handle
                 call    far ptr VIOWRTTTY
                 endm
```

use: Figure 3.17b, 3.20b, 3.21b, 3.23b, 3.24b

```
@DosSubSet       macro   m1, m2, m3
                 @def    DOSSUBSET
                 @pw     m1                          ;selector
                 @pw     m2                          ;flags
                 @pw     m3                          ;size
                 call    far ptr DOSSUBSET
```

use: Figure 3.15

```
@DosWrite        macro   m1, m2, m3, m4
                 @def    DOSWRITE
                 @pw     m1                          ;handle
                 @ps     m2                          ;buffer
                 @pw     m3                          ;length
                 @ps     m4                          ;byteswritten
                 call    far ptr DOSWRITE
                 endm
```

use: Figure 2.1, 2.6b, 3.3, 3.22b

```
@DosWriteQueue1  macro   m1, m2, m3, m4, m5
                 @def    DOSWRITEQUEUE1
                 @pw     m1                          ;handle
                 @pw     m2                          ;request
                 @pw     m3                          ;length
                 @pd     m4                          ;buffer
                 @pw     m5                          ;priority
                 call    far ptr DOSWRITEQUEUE1
                 endm
```

use: Figure 3.25b

```
@KbdStringIn     macro   m1, m2, m3, m4
                 @def    KBDSTRINGIN
                 @ps     m1                          ;buffer
                 @ps     m2                          ;length
                 @pw     m3                          ;iowait
                 @pw     m4                          ;handle
                 call    far ptr KBDSTRINGIN
                 endm
```

use: Figure 2.3, 2.2b, 2.10, 2.15, 3.1, 3.4, 3.18b

```
@VioGetPhysBuf   macro   m1, m2
                 @def    VIOGETPHYSBUF
                 @ps     m1                          ;structure
                 @pw     m2                          ;reserved
                 call    far ptr VIOGETPHYSBUF
                 endm
```

use: Figure 2.3, 2.7b, 2.8, 2.10, 2.15, 3.1, 3.4, 3.6, 3.18b, 3.19

```
                call    far  ptr  DOSREADQUEUE
                endm
```

use: Figure 3.24b

```
@DosReAllocSeg  macro   m1,  m2
                @def    DOSREALLOCSEG
                @pw     m1                        ;size
                @pw     m2                        ;selector
                call    far  ptr  DOSREALLOCSEG
                endm
```

use: Figure 3.8

```
@DosSemClear    macro   m1
                @def    DOSSEMCLEAR
                @pd     m1                        ;handle
                call    far  ptr  DOSSEMCLEAR
                endm
```

use: Figure 3.17b, 3.21b, 3.23b

```
@DosSemSet      macro   m1
                @def    DOSSEMSET
                @pd     m1                        ;handle
                call    far  ptr  DOSSEMSET
                endm
```

use: Figure 3.17b, 3.20b, 3.22b

```
@DosSemWait     macro   m1,  m2
                @def    DOSSEMWAIT
                @pd     m1                        ;handle
                @pd     m2                        ;timeout
                call    far  ptr  DOSSEMWAIT
                endm
```

use: Figure 3.17b, 3.20b, 3.22b

```
@DosSubAlloc    macro   m1,  m2,  m3
                @def    DOSSUBALLOC
                @pw     m1                        ;selector
                @ps     m2                        ;offset
                @pw     m3                        ;size
                call    far  ptr  DOSSUBALLOC
                endm
```

use: Figure 3.15

```
                @pw     m5                              ;attribute
                @pw     m6                              ;openflag
                @pw     m7                              ;openmode
                @pd     m8                              ;0
                call    far ptr DOSOPEN
                endm
```

use: Figure 2.1, 2.6b

```
@DosOpenQueue   macro   m1, m2, m3
                @def    DOSOPENQUEUE
                @ps     m1                              ;owner ID
                @ps     m2                              ;handle
                @ps     m3                              ;name
                call    far ptr DOSOPENQUEUE
                endm
```

use: Figure 3.25b

```
@DosOpenSem     macro   m1, m2
                @def    DOSOPENSEM
                @ps     m1                              ;handle
                @ps     m2                              ;name
                call    far ptr DOSOPENSEM
                endm
```

use: Figure 3.21b, 3.23b

```
@DosRead        macro   m1, m2, m3, m4
                @def    DOSREAD
                @pw     m1                              ;handle
                @ps     m2                              ;buffer
                @pw     m3                              ;length
                @ps     m4                              ;bytesread
                call    far ptr DOSREAD
                endm
```

use: Figure 3.23b

```
@DosReadQueue   macro   m1, m2, m3, m4, m5, m6, m7, m8
                @def    DOSREADQUEUE
                @pw     m1                              ;handle
                @ps     m2                              ;request
                @ps     m3                              ;length
                @ps     m4                              ;address
                @pw     m5                              ;code
                @pw     m6                              ;nowait
                @ps     m7                              ;priority
                @pd     m8                              ;semhandle
```

```
                call    far  ptr  DOSGETHUGESHIFT
                endm
```

use: Figure 3.12

```
@DosGetShrSeg   macro   m1, m2
                @def    DOSGETSHRSEG
                @ps     m1                          ;name
                @ps     m2                          ;selector
                call    far  ptr  DOSGETSHRSEG
                endm
```

use: Figure 3.5, 3.23b

```
@DosGiveSeg     macro   m1, m2, m3
                @def    DOSGIVESEG
                @pw     m1                          ;caller sel
                @pw     m2                          ;process ID
                @ps     m3                          ;recipient sel
                call    far  ptr  DOSGIVESEG
                endm
```

use: Figure 3.25b

```
@DosKillProcess macro   m1, m2
                @def    DOSKILLPROCESS
                @pw     m1                          ;action
                @pw     m2                          ;result
                call    far  ptr  DOSKILLPROCESS
                endm
```

use: Figure 3.4, 3.20b, 3.22b, 3.24b

```
@DosMakePipe    macro   m1, m2, m3
                @def    DOSMAKEPIPE
                @ps     m1                          ;read hdl
                @ps     m2                          ;write hdl
                @pw     m3                          ;size
                call    far  ptr  DOSMAKEPIPE
                endm
```

use: Figure 3.22b

```
@DosOpen        macro   m1, m2, m3, m4, m5, m6, m7, m8
                @def    DOSOPEN
                @ps     m1                          ;name
                @ps     m2                          ;handle
                @ps     m3                          ;action
                @pd     m4                          ;size
```

```
                    call    far  ptr  DOSCREATESEM
                    endm
```

use: Figure 3.20b, 3.22b

```
@DosCreateThread    macro   m1,  m2,  m3
                    @def    DOSCREATETHREAD
                    @pd     m1                         ;address
                    @ps     m2                         ;thread  ID
                    @pd     m3                         ;end  stack
                    call    far  ptr  DOSCREATETHREAD
                    endm
```

use: Figure 3.17b, 3.18b

```
@DosExecPgm         macro   m1,  m2,  m3,  m4,  m5,  m6,  m7
                    @def    DOSEXECPGM
                    @ps     m1                         ;name  buffer
                    @pw     m2                         ;length
                    @pw     m3                         ;flags
                    @ps     m4                         ;argpointer
                    @ps     m5                         ;envpointer
                    @ps     m6                         ;retrun
                    @ps     m7                         ;pgmpointer
                    call    far  ptr  DOSEXECPGM
                    endm
```

use: Figure 3.4, 3.20b, 3.22b, 3.24b

```
@DosExit            macro   m1,  m2
                    @def    DOSEXIT
                    @pw     m1                         ;action
                    @pw     m2                         ;result
                    call    far  ptr  DOSEXIT
                    endm
```

use: all processes

```
@DosFreeSeg         macro   m1
                    @def    DOSFREESEG
                    @pw     m1                         ;selector
                    call    far  ptr  DOSFREESEG
                    endm
```

use: Figure 3.1, 3.4, 3.5, 3.12, 3.15, 3.24b, 3.25b

```
@DosGetHugeShift    macro   m1
                    @def    DOSGETHUGESHIFT
                    @ps     m1                         ;shiftcount
```

```
@DosAllocShrSeg macro   m1, m2, m3
                @def    DOSALLOCSHRSEG
                @pw     m1                          ;no. bytes
                @ps     m2                          ;name
                @ps     m3                          ;selector
                call    far ptr DOSALLOCSHRSEG
                endm
```

use: Figure 3.4, 3.22b

```
@DosBeep        macro   m1, m2
                @def    DOSBEEP
                @pw     m1                          ;hertz
                @pw     m2                          ;duration
                call    far ptr DOSBEEP
                endm
```

use: Figure 3.17b, 3.18b, 3.20b, 3.21b, 3.22b, 3.23b, 3.24b, 3.25b

```
@DosClose       macro   m1
                @def    DOSCLOSE
                @pw     m1                          ;handle
                call    far ptr DOSCLOSE
                endm
```

use: Figure 2.1, 2.6b, 3.3, 3.4

```
@DosCloseQueue  macro   m1
                @def    DOSCLOSEQUEUE
                @pw     m1                          ;handle
                call    far ptr DOSCLOSEQUEUE
                endm
```

use: Figure 3.24b, 3.25b

```
@DosCreateQueue macro   m1, m2, m3
                @def    DOSCREATEQUEUE
                @ps     m1                          ;handle
                @pw     m2                          ;priority
                @ps     m3                          ;name
                call    far ptr DOSCREATEQUEUE
                endm
```

use: Figure 3.24b

```
@DosCreateSem   macro   m1, m2, m3
                @def    DOSCREATESEM
                @pw     m1                          ;no exclusive
                @ps     m2                          ;handle
                @ps     m3                          ;name
```

```
@ps             macro       m1              ;push address
                mov         ax, SEG m1
                push        ax
                mov         ax, OFFSET m1
                push        ax
                endm

@pd             macro       m1              ;push doubleword
                push        ds
                push        bx
                mov         ax, SEG m1
                mov         ds, ax
                mov         bx, OFFSET m1
                push        word ptr [bx]
                mov         ax, [bx+2]
                push        bp
                push        sp
                pop         bp
                xchg        [bp+6], ax
                pop         bp
                mov         ds, ax
                pop         ax
                pop         bx
                push        ax
                endm
```

With these preliminary macros defined it is now possible to define the macro calls used in the book.

```
@DosAllocHuge   macro   m1, m2, m3, m4, m5
                @def    DOSALLOCHUGE
                @pw     m1                      ;no. segments
                @pw     m2                      ;size last seg
                @ps     m3                      ;selector
                @pw     m4                      ;max seg
                @pw     m5                      ;flags
                call    far ptr DOSALLOCHUGE
                endm
```

use: Figure 3.12

```
@DosAllocSeg    macro   m1, m2, m3
                @def    DOSALLOCSEG
                @pw     m1                        ;no. bytes
                @ps     m2                        ;selector
                @pw     m3                        ;flags
                call    far ptr DOSALLOCSEG
                endm
```

use: Figure 3.1, 3.8, 3.15, 3.25b

points determined by symbols such as INCL_BASE. Typically, these include files are:

Level 1 (OS/2):	os/2.inc (includes os/2.def.inc.bse.inc, and pm.inc)
Level 2 (OS/2):	os/2def.inc (defines constants, types, error codes, and structures) bse.inc (includes bsedos.inc, bsesub.inc, and bseerr.inc)
Level 3 (OS/2):	bsedos.inc (defines constants, structures, and prototypes for the Dos API) bsesub.inc (sets up calls for Vio, Kbd, and Mou API) bseerr.inc. (sets up error code constants for all API calls)
Level 2 (PM):	pm.inc (includes pmwin.inc, pmgpi.inc, pmdef.inc, pmavio.inc, pmspl.inc, pmpic.inc, pmord.inc, pmbitmap.inc, pmfont.inc)
Level 3 (PM):	pmwin.inc (sets up windows, message manager, keyboard, mouse, and dialog manager API calls) purgpi.inc (sets up Gpi API calls) pmdev.inc (sets up device context API calls) pmavio.inc (sets up the PM Vio API calls) pmspl.inc [sets up the spool (Spl) API calls] pmpic.inc (sets up the picture API calls) pmord.inc (sets up the GOCA orders for the Gpi API calls) pmbitmap.inc (sets up the bitmap types) pmfont.inc (sets up the types for fonts)

These include files contain macros for loading API service routines and pushing the stack with appropriate parameter data. In this appendix it is desirable to present a similar set of macro-based or function calls used in the assembly language in this book. These macros bridge the gap between the macro calls used in the text and the actual assembler code required to lead a particular API service. They are very similar to the macros available through the Toolkit. We ignore the error-processing features of the Toolkit macros and leave the addition of these features to the reader. With these thoughts in mind, let us begin with several subordinate macro definitions:

```
@pw         macro     m1          ;push word
            mov       ax,m1
            push      ax
            endm

@def        macro     nm          ;define API entry
            ifndef    nm
            extrn     nm:far
            endif
            endm
```

C Function Declarations and Macros Used to Interface the API

The IBM Programmer's Toolkit Versions 1.0 and 1.1 [1.2] contain a set of assembler macros and C function declarations that provide interfaces to the API services. In addition, Version 1.1 contains macros and C function declarations for accessing the Presentation Manager (PM). In this appendix we address similar interfaces for C and assembler and provide the relevant code for the macro calls (assembler) and function declarations (C) used in this book. The reader is referred to the Toolkit for a complete discussion of similar macros and function declarations.

C.1 THE ASSEMBLER INTERFACE

The primary assembler include file in Version 1.0, and available under Version 1.1, is

```
sysmac.inc
```

This, in turn, calls

```
doscalls.inc
subcalls.inc
```

which loads Dos, Mou, Kbd, and Vio service macros.

Under Version 1.1, a new set of include files is provided with variable entry

We can use pointers to access this structure through the same approach; for example, the fifth element can be accessed using

```
pcar_type -> ...
```

These two statements have identical results.

In a structure, space is reserved for each element. In a union, space is reserved only for the largest element; all other elements must share this space. For example, in the following template:

```
union
  {
  int c;
  int d;
  float g;
  double h;
  } letter, *pletter;
```

the largest amount of space reserved for the union is 8 bytes with the h variable. All the remaining variables must share the space in storage with this variable amount. Since c and d each occupy 2 bytes and g occupies 4 bytes, this union can be used to store c, d, and g simultaneously, or, alternatively, h or other combinations less than or equal to 8 bytes.

This discussion completes our brief look at C. We have attempted to touch only on those syntax features that are used in programming the OS/2 Kernel.

REFERENCES

1. *Microsoft C 5.1 Optimizing Compiler: CodeView, and Utilities, Microsoft Editor, Mixed-Language Programming Guide,* Microsoft Corporation, Redmond, WA, 1987.
2. *Microsoft C 5.1 Optimizing Compiler: Run-Time Library Reference,* Microsoft Corporation, Redmond, WA, 1987.
3. *Microsoft C 5.1 Optimizing Compiler: User's Guide and Language Refer*ence, Microsoft Corporation, Redmond, WA, 1987.
4. Godfrey, J. T., *Applied C: The IBM Microcomputers*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
5. Petzold, C., *Programming the OS/2 Presentation Manager*, Microsoft Corporation, Redmond, WA, 1989.

In many respects C is the language of choice for programming the API once the programmer obtains a familiarity with its features. Certainly, C is the choice for programming the Presentation Manager [5].

The structure for a function in C has the following form:

```
function_name()
  ...
  formal parameter types
  ...
  {
  ...
  local parameter types
  ...
  statements
  ...
  return expression;
  }
```

Here the lines of code immediately following the function definition statement contain the typing for formal parameters. The function brackets are next, with the local parameter typing contained within the function brackets, together with all remaining function statements. If a value such as d is to be returned, this value is assigned and used as the argument of a return( ) statement.

A structure, for example, can look as follows:

```
struct tag_name
  {
  ...
  type declarations
  ...
  } struct_name;
```

This architecture allows a tag name identifier, tag_name, to be used with later definitions to define a structure that has similar characteristics. For example, the structure

```
struct car
  {
  char olds, chevy, pontiac;
  char accura, honda, mazda;
  char ford, lincoln, mercury;
  } car_type, *pcar_type;
```

has as tag, car, and as structure name, car_type. The structure elements can be accessed using a period to offset the element from the structure name. In the structure above the fifth element is accessed using

```
car_type.honda = ...
```

contains some additional operators of less utility. The basic C data types are as follows:

```
int      = integer (2 bytes); signed
long     = integer (4 bytes); signed
short    = integer (2 bytes); signed
unsigned = integer; zero or positive values

           unsigned int (2 bytes);   0 - 255
           unsigned long (4 bytes);  0 - 65535
           unsigned short (2 bytes); 0 - 255

char     = character (1 byte)    38        38
float    = floating (4 bytes); -10307 - +10307
double   = floating (8 bytes); -10 - +10
```

Within the OS/2 Kernel programming are a number of additional derived types used by Microsoft and IBM to expand the flexibility of OS/2. Some of these types are

`SEL`	segment selector
`PSEL`	pointer to selector
`SHANDLE`	handle
`BYTE`	byte (char)
`PCHAR`	pointer to character
`HFILE`	handle to file
`HSEM`	handle to semaphore
`PUINT`	pointer to unsigned integer
`HVIO`	handle to video context
`TID`	thread ID

The addition of the Presentation Manager files adds many more derived types to the OS/2 inventory.

The basic C storage classes are auto, external, static, and register:

`auto`	generated with temporary duration within a module as a local class
`external`	generated for all time as a global
`static`	generated for all time but local in scope
`register`	generated with temporary duration within a module as local and, if possible, associated with a CPU register

The remaining topics to be briefly examined are functions, structures, unions, and pointers. This, then, will complete our look at the C syntax. The examples in the text are intended to provide additional insight into the C language and its applicability in the OS/2 programming environment.

When *expression2* is evaluated TRUE, the associated statements are executed. Once the *continue* statement is executed, the processing jumps to the end of the loop without entering the second if structure regardless of whether *expression3* is TRUE or FALSE.

Table B.1 illustrates the major operators found in the C language. Table B.2

TABLE B.1 C OPERATORS

Operator	Discussion
( )	Grouping
{ }	Executes all contained syntax
++	Increment
--	Decrement
*	Multiply
/	Divide
+	Add
-	Subtract
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
&&	AND: logical
¦¦	OR: logical
=	Equal: assignment
+=	Adds right-hand quantity to left hand
-=	Subtracts right-hand quantity from left hand
*=	Multiplies left hand by right hand
/=	Divides left hand by right hand
==	Equal to: relational
!=	Not equal to: relational
%	Modulus
%=	Modulus after dividing left hand by right hand
*	Pointer: gives the value at the pointed address
&	Pointer: gives the address of the variable

TABLE B.2 ADDITIONAL C OPERATORS

Operator	Discussion
(type)	Changes the type of a variable
sizeof	Returns the size in bytes of the variable
->	Assigns a structure member
.	Assigns a structure member
!	NOT: bitwise
~	Takes one's complement: bitwise
&	AND: bitwise
^	EXCLUSIVE OR: bitwise
¦	OR: bitwise
?:	Conditional operator
<<	Left shift: bitwise
>>	Right shift: bitwise

Here the *alternate statements* are executed when expression is FALSE. The case, switch, and default statements work together. Consider the following structure:

```
switch (expression)
  {
  case A:
    statement #1;
    break;
  case B:
    statement #2;
    break;
  case C:
    statement #3;
    break;
  ...
  default:
    statement N;
    break;
  }
```

Here, if the value of *expression* takes on A, B, C, ... the corresponding *case* sequence is executed. For all values not specified with a subsequent *case* statement, the *default* statement sequence is executed.

The switch decision structure is used frequently in the Presentation Manager windows processing. We do not use this structure because the examples in this book did not involve multiple options. The break syntax was used to jump around subsequent statements once the preceding statement had been executed.

There are two statements that can be used to alter the sequence of processing. These are the jump statements, continue and goto. Consider the following loop:

```
for(expression1)
  {
  statements
  ...
  if{expression2)
    {
    alternate1 statements
    ...
    continue;
    }
  if(expression3)
    {
    alternate2 statements
    ...
    }
  }
```

```
for (k=1;k<=N;k++)
  {
  statements
  ...
  }
```

The while loop has the form

```
while(expression)
  {
  statements
  ...
  }
```

where *expression* is returned as a TRUE or FALSE value. When TRUE the *statements* in the brackets are executed. Otherwise, the processing passes to subsequent statements, outside the brackets.

The do while loop has an inverted structure with a test at the end of the loop:

```
do
  {
  statements
  ...
  } while(expression)
```

Here *statements* are executed the first time through the loop and each subsequent time that *expression* evaluates TRUE.

Decision structures are represented by the if, else, case, switch, and default statements. The if structure is of the form

```
if(expression)
  {
  statements
  ...
  }
```

where a TRUE value for *expression* causes the *statements* to be executed. The else statement appears as follows, and is used in conjunction with the if statement:

```
if(expression)
  {
  statements
  ...
  }
else
  {
  alternate statements
  ...
  }
```

B Microsoft C Compiler Version 5.1

In Part III of this book the OS/2 Kernel was programmed using the C language. The specific C implementaion used was the Version 5.1 C Optimizing Compiler developed by the Microsoft Corporation [1–3]. This compiler was one of the first that was made commercially available that would execute in the Protected Mode, so that it could be used with OS/2. Associated with the compiler is a Toolbox that is discussed in Appendix C. This Toolbox provides high-level C interfaces to the OS/2 API. These interfaces are suitable for use with the Microsoft C compilers and are provided as a set of .h include files.

In this appendix we briefly review some of the C language syntax used in this book. We assume that the reader has a familiarity with C, hence we only provide this appendix for reference. The following categories are mentioned:

1. Control structures
2. Operators
3. Data types and storage classes
4. Other syntax

The treatment of this appendix is similar to that given in *Applied C: The IBM Microcomputers* [4].

The basic control structures fall into three categories: loops, decision structures, and jumps. Loops consist of the for, while, and do while syntax. The for loop structure takes the form, for example:

TABLE A.11 *(Concluded)*

Instruction	Purpose	Comments
`FENI/FNENI`	Enable interrupts	This instruction is the reverse of FDISI and clears the interrupt mask in the control word.
`FLDCW source`	Load control word	This instruction replaces the current control word with the word defined by the source operand.
`FSTCW/FNSTCW destination`	Store control word	This instruction writes the current control word to the memory location defined by destination.
`FSTSW/FNSTSW destination`	Store status word	This instruction writes the current status word to the memory location defined by destination.
`FCLEX/FNCLEX`	Clear exceptions	Clears all exception flags, the interrupt request and busy flag.
`FSTENV/FNSTENV destination`	Store environment	Writes the basic status and exception pointers to the memory location defined by destination.
`FLDENV source`	Load environment	Reloads the 8087 environment from the memory area defined by the source.
`FSAVE/FNSAVE destination`	Save state	Writes the environment and register stack to the memory location specified by the destination operand.
`FRSTOR source`	Restore state	Reloads the 8087 from the source operand.
`FINCSTP`	Increment stack pointer	Adds 1 to the stack pointer.
`FFREE destination`	Free register	Changes the destination's tag to empty.
`FDECSTP`	Decrement stack pointer	Subtracts 1 from the stack pointer.
`FNOP`	No operation	Causes no operation.
`FWAIT`	Wait instruction	Causes the 8088 to wait until the current 8087 instruction is complete before the 8088 executes another instruction.

REFERENCES

1. *IBM Macro Assembler Version 2.00 Language Reference*, International Business Machines Corporation, Boca Raton, FL, 1984.
2. *IBM Macro Assembler Version 2.00 Fundamentals: Assemble, Link, and Run,* International Business Machines Corporation, Boca Raton, FL, 1984.
3. Godfrey, J. T., *IBM Microcomputer Assembly Language: Beginning to Advanced,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

TABLE A.11 *(Continued)*

Instruction	Purpose	Comments
FICOM source	Integer compare	This instruction compares ST(0) to the source operand, which is an integer-memory operand.
FICOMP source	Integer compare/ pop	This instruction is identical to FICOM except the stack top, ST(0), is popped following the compare.
FTST	Test	This instruction tests ST(0) relative to +0.0. The result of the test is returned in the condition code of the status word: (C3, C0) = (0, 0) for ST positive, (0, 1) for ST negative, (1, 0) for ST zero, and (1,1) if ST cannot be compared.
FXAM	Examine	The stack top, ST(0), is examined and the result returned in the condition code field as specified in the Version 2.0 *Macro Assembler Reference* manual.
Transcendental		
FPTAN	Partial tangent	This instruction calculates Y/X = TAN(z). The value z is contained in ST(0) prior to execution. Following execution, Y is contained in ST(1) and X contained in ST(0).
FPATAN	Partial arc tangent	This instruction calculates z = ARCTAN(Y/X), where X is ST(0) and Y is ST(1). The result, z, is returned to ST(0).
F2XM1	$2^x - 1$	This instruction calculates $2^x - 1$, where x is taken from ST(0) and must be in the range (0, 0.5). The result is replaced in ST(0).
FYL2X	$Y * \log_2(X)$	This instruction calculates $Y * \log_2(X)$, where X is ST(0) and Y is ST(1). The stack top is popped and the result returned to the new ST(0).
FYL2XP1	$Y * \log_2 (X + 1)$	This instruction is the same as FYL2X except 1 is added to X. X must be in the range $(0, 1 - \sqrt{2}/2)$.
Constant		
FLDZ	Load zero	This instruction loads +0.0 in ST(0).
FLD1	Load +1.0	This instruction loads +1.0 in ST(0).
FLDP1	Load pi	This instruction loads pi into ST(0).
FLDL2T	Load $\log_2(10)$	This instruction loads $\log_2(10)$ into the stack top, ST(0).
FLDL2E	Load $\log_2(e)$	This instruction loads $\log_2(e)$ into ST(0).
FLDLG2	Load $\log_{10}(2)$	This instruction loads $\log_{10}(2)$ into ST(0).
FLDLN2	Load $\log_e(2)$	This instruction loads $\log_e(2)$ into ST(0).
Control		
FINIT/FNINIT	Initialize processor	This instruction accomplishes a hardware reset of the 8087.
FDISI/FNDISI	Disable interrupts	This instruction prevents the 8087 from issuing an interrupt request.

TABLE A.11 *(Continued)*

Instruction	Purpose	Comments
FIDIV source	Integer divide	This instruction divides the destination by the source and returns the quotient to the destination. The destination is ST(0) and the source is an integer-memory operand.
FDIVR	Real reversed divide	This instruction is identical with FDIV except the source is divided by the destination. The quotient is still returned in the destination.
FDIVRP destination, source	Real reversed divide/pop	This instruction is identical to FDIVP except the source is divided by the destination. The quotient is still returned in the destination.
FIDIVR source	Integer divide reversed	This instruction is identical to FIDIV except the source is divided by the destination. The quotient is still returned in the destination.
Miscellaneous		
FSQRT	Square root	This instruction replaces the content of ST(0) with its square root.
FSCALE	Scale	This instruction interprets the value of the number contained in ST(1) as an integer. This value is added to the exponent of the number in ST(0), which is equivalent to multiplying ST(0) by 2 raised to this integer power.
FPREM	Partial remainder	This instruction takes the modulo of ST relative to the number contained in ST(1). The sign is the same as that of ST(0).
FRNDINT	Round to integer	This instruction rounds ST(0) to an integer. The rules for rounding are determined by setting the RC field of the control word. RC = 00 (round to nearest integer), 01 (round downward, 10 (round upward), and 11 (round toward 0).
FXTRACT	Extract exponent/significand	This instruction reduces the number in ST(0) to a significand and an exponent for 80-bit arithmetic.
FABS	Absolute value	This instruction yields the absolute value of ST(0).
FCHS	Change sign	This instruction reverses the sign of ST(0).
Comparison		
FCOM	Real compare	This instruction compares the source operand [which can be specified as a real-memory operand or implicit as ST(1)] and ST(0).
FCOMP	Real compare/pop	This instruction is identical with FCOM except the stack top, ST(0), is popped following the compare.
FCOMPP	Real compare/pop twice	This instruction is identical with FCOM except the stack top, ST(0), and ST(1) are popped following the compare.

TABLE A.11 *(Continued)*

Instruction	Purpose	Comments
FISUB source	Integer subtraction	The destination, ST(0), has the source operand, an integer-memory operand, subtracted from it and the result is stored in ST(0).
FSUBR	Real reversed subtract	The destination is subtracted from the source and the result left in the destination. The operand configuration is the same as for FSUB.
FSUBRP	Real reversed subtract/pop	This instruction is the same as FSUBP except the destination is subtracted from the source. ST(0) still serves as the source operand.
FISUBR source	Integer reversed subtract	This instruction is the same as FISUB except the destination is subtracted from the source. The source is still an integer-memory operand.
Multiplication		
FMUL	Real multiply	This instruction multiplies the destination operand by the source and returns the product in the destination. The instruction can be executed with no operands [ST(0) is the implied source and ST(1) the destination], with the source specified as a real-memory operand and ST(0) the destination, and with both destination register and source register [one of which is ST(0)] specified.
FMULP destination, source	Real multiply/pop	This instruction uses ST(0) as the source operand and another register as the destination. The product is returned in the destination register and the stack top popped.
FIMUL source	Integer multiply	This instruction multiplies the destination by the source and returns the product in the destination. The destination is ST(0) and source is an integer-memory operand.
Division		
FDIV	Real divide	This instruction divides the destination by the source and returns the quotient to the destination. The instruction can be executed with no operands [ST(0) is the implied source and ST(1) the implied destination], with a source specified and ST(0) the implied destination, and with a source [ST(0)] and destination (another register) specified.
FDIVP destination, source	Real divide/pop	This instruction divides the destination by the source and returns the quotient to the destination. It then pops the top of the 8087 stack. The source is the ST(0) register and the destination operand is another stack register.

TABLE A.11 COPROCESSOR INSTRUCTION SET

Instruction	Purpose	Comments
Data transfer		
`FLD source`	Load real	Pushes the source data onto the top of the register stack, ST(0).
`FST destination`	Store real	This instruction copies ST(0) into the indicated destination (real), which can be a memory operand or register.
`FSTP destination`	Store real/pop	This instruction copies ST(0) into the indicated destination and then pops ST(0) off the stack.
`FXCH destination`	Exchange ST	This instruction exchanges ST(0) with the indicated destination.
`FILD source`	Load integer	This instruction pushes the source data (integer) onto the top of the stack, ST(0).
`FIST destination`	Store integer	This instruction stores ST(0), the stack top, in the indicated destination, which must be an integer memory operand.
`FISTP destination`	Store integer/ pop	This instruction stores ST(0), the stack top, in the indicated destination, which must be an integer memory operand, and then pops ST(0) off the stack.
`FBLD source`	Load BCD	This instruction pushes the source, which must be a BCD number, onto the stack at ST(0).
`FBSTP destination`	Store BCD/pop	This instruction stores ST(0) as a BCD number at the destination and pops ST(0) off the stack.
Addition		
`FADD`	Real addition	This instruction can be used without operands [assumes ST(1) added to ST(0) with the result in ST(0)], with a real-memory operand added to ST(0), or with explicit reference to ST(0) added to another register.
`FADDP destination, source`	Real add/pop	The source is ST(0) and the destination must be another stack register. The result is left in the alternate stack register used as the destination.
`FIADD integer-memory`	Integer addition	The destination, ST(0), is added to the source, integer memory, and the sum returned in ST(0).
Subtraction		
`FSUB`	Real subtraction	This instruction can be used without operands [assumes ST(1) is the destination and ST(0) is subtracted from it with the result in ST(1)], with a real-memory operand subtracted from ST(0) and the result in ST(0), or with explicit reference to ST(0) and another register (the destination containing the result).
`FSUBP destination, source`	Real subtract/ pop	The source, ST(0), is subtracted from the destination, another stack register, and the result stored in the destination.

TABLE A.10 SPECIAL-PURPOSE MACRO OPERATORS

Operator	Description

&	Format: text&text. This operator concatenates text or symbols. An example is
	`TC1 MACRO X`
	`LEA DX, CHAR&X`
	`MOV AH, 9`
	`INT 21H`
	`ENDM`
	Here a call TC1 A would load DX with a character start position CHARA.

;;	Format: ;;text. A comment preceded by two semicolons is not produced as part of the expansion when a MACRO or REPT is defined in an assembly.
!	Format: !character. Causes the character to be interpreted as a literal value, not a symbol.

%	Format: %expression. Converts expression to a number. During expansion, the number is substituted for expression. Consider
	`MAC1 MACRO X`
	`L1 = X * 1000`
	`MAC2 %L1,X`
	`ENDM`
	`;`
	`MAC2 MACRO Y,X`
	`PROD&X DB 'Production No. &X = &Y'`
	`ENDM`
	This yields "PROD5 DB 'Production No. 5 = 5000,'" when called with MAC1 5.

TABLE A.9 *(Concluded)*

Pseudo-op	Description
.LFCOND	This pseudo-op causes the listing of conditional blocks that evaluate as false.
.LIST and .XLIST	.LIST causes a listing of source and object code in the output assembler list file. .XLIST turns this listing off. These pseudo-ops can be used to selectively list code during the assembly of programs, especially long sequences of instructions.
%OUT	Form: %OUT text. This pseudo-op is used to monitor progress through a long assembly. The argument "text" is displayed, when encountered, during the assembly process.
PAGE	Form: PAGE operand1, operand2. Controls the length (operand1) in lines and the width (operand2) in characters of the assembler list file.
.SFCOND	This pseudo-op suppresses the listing of conditional blocks that evaluate as false.
SUBTTL	Form: SUBTTL text. Generates a subtitle to be listed after each listing of title.
.TFCOND	This pseudo-op changes the listing setting (and default) for false conditionals to the opposite state.
TITLE	Form: TITLE text. This pseudo-op specifies a title to be listed on each page of the assembler listing. It may be used only once.
ENDM	ENDM is the terminator for MACRO, REPT, IRP, and IRPC.
EXITM	EXITM provides an exit to an expansion (REPT, IRP, IRPC, or MACRO) when a test proves that the remaining expansion is not needed.
IRP	Form: IRP dummy, <operandlist>. The number of operands (separated by commas) in operandlist determines the number of times the following code (terminated by ENDM) is repeated. At each repetition, the next item in operandlist is substituted for all occurrences of dummy.
IRPC	Form: IRPC dummy, string. This is the same as IRP except at each repetition the next character in string is substituted for all occurrences of dummy.
LOCAL	Form: LOCAL dummylist. LOCAL is used inside a MACRO structure. The assembler creates a unique symbol for each entry in dummylist during each expansion of the macro. This avoids the problem of a multiply defined label, for example, when multiple expansions of the same macro take place in a program.
MACRO	Form: name MACRO dummylist. The statements following the MACRO definition, before ENDM, are the macro. Dummylist contains the parameters to be replaced when calling the macro during assembly. The form of this call is name parmlist. Parmlist consists of the actual parameters (separated by commas) used in the expansion.
PURGE	Form: PURGE macro-name, ... PURGE deletes the definition of a specified MACRO and allows the space to be used. This is beneficial when including a macro library during assembly but desiring to remove those macros not used during the assembly.

TABLE A.9 *(Continued)*

Pseudo-op	Description
END	Form: END [expression]. END identifies the end of the source program, and the optional expression identifies the name of the entry point.
ENDP	Form: procedure-name ENDP. Designates the end of a procedure.
ENDS	Form: structure-name ENDS or seg-name ENDS. Designates the end of a structure or segment.
EQU	Form: name EQU expression. Assigns the value of expression to name. This value may not be reassigned.
=	Form: label = expression. Assigns the value of expression to label. May be reassigned.
EVEN	EVEN ensures that the code following starts on an even boundary.
EXTRN	Form: EXTRN name:type, ... EXTRN is used to indicate that symbols used in this assembly module are defined in another module.
GROUP	Form: name GROUP seg-name, ... GROUP collects all segments named and places them within a 64K physical segment.
INCLUDE	Form: INCLUDE [drive] [path] filename.ext. INCLUDE assembles source statements from an alternate source file into the current source file.
LABEL	Form: name LABEL type. LABEL defines the attributes of name to be type.
NAME	Form: NAME module-name. NAME gives a module a name. It may be used only once per assembly.
ORG	Form: ORG expression. The location counter is set to the value of expression.
PROC	Form: procedure-name PROC [attribute]. PROC identifies a block of code as a procedure and must end with RET/ENDP. The attribute is NEAR or FAR.
PUBLIC	Form: PUBLIC symbol, ... PUBLIC makes symbols externally available to other linked modules.
.RADIX	Form: .RADIX expression. .RADIX allows the default base (decimal) to be changed to a value between 2 and 16.
RECORD	Form: recordname RECORD fieldname:width [=exp], ... RECORD defines a bit pattern to format bytes and words for bit packing (see text).
SEGMENT	Form: segname SEGMENT [align-type] [combine-type] ['class'] (see Chapter 3 for a discussion of this pseudo-op).
STRUC	Form: structure-name STRUC. STRUC is used to allocate and initialize multibyte variables using DB, DD, DQ, DT, and DW. It must end with ENDS.
.CREF and .XCREF	This listing pseudo-op provides cross-reference information when a filespec is indicated in response to the assembler prompt (CREF). It is the normal default condition. .XCREF results in no output for cross reference when in force.
.LALL, .SALL, and .XALL	.LALL lists the complete macro text for all expansions. .SALL suppresses listing of all text and object code produced by macros. .XALL produces a source line listing only if object code results.

TABLE A.9 APPLICATION-ORIENTED PSEUDO-OPS

Pseudo-op	Description
ELSE	This pseudo-op must be used in conjunction with a conditional pseudo-op and serves to provide an alternate path.
ENDIF	This pseudo-op ends the corresponding IFxxx conditional.
IF	Form: IF expression. When the expression is true, the code following this pseudo-op is executed; otherwise it branches to an ELSE entry point or an ENDIF. IF pseudo-ops can be nested.
IFB	Form: IFB <operand>. This is the "if blank" pseudo-op and it is true if the operand has not been specified as in a MACRO call, for example. The code following the IFB is executed when operand is blank. Otherwise, the IP jumps to ENDIF.
IFDEF	Form: IFDEF symbol. If symbol has been defined via the EXTRN pseudo-op, this is true and the code following the pseudo-op is executed.
IFDIF	Form: IFDIF <operand1>, <operand2>. The code following this pseudo-op is executed if the string operand1 is different from the string operand2.
IFE	Form: IFE expression. The code following this pseudo-op is executed if expression = 0.
IFIDN	Form: IFIDN <operand1>, <operand2>. The code following this pseudo-op is executed if the string operand1 is identical to the string operand2.
IFNB	Form: IFNB <operand>. The code following this pseudo-op is executed if the operand is not blank.
IFNDEF	Form: IFNDEF symbol. The code following this pseudo-op is executed if the symbol has not been defined via the EXTRN pseudo-op.
IF1	This pseudo-op is true if the assembler is in pass 1, and it is used to load macros from a macro library (as an example).
IF2	This pseudo-op is true if the assembler is in pass 2, and it can be used to inform the programmer what version of the program is being used (when coupled with appropriate logic and a %OUT).
.286C	This pseudo-op tells the assembler to recognize and assemble 80286 instructions used by the IBM AT.
.8086	This pseudo-op tells the assembler not to recognize and assemble 80286 instructions.
.8087	This pseudo-op tells the assembler to recognize and assemble 8087 coprocessor instructions and data formats.
ASSUME	Form: ASSUME seg-reg:seg-name, ... This pseudo-op tells the assembler which segment register segments belong to.
COMMENT	Form: COMMENT delimiter text delimiter. COMMENT allows the programmer to enter comments without semicolons. It is not recognized by the SALUT program.
DB	Form: [variable] DB [expression]. It is used to initialize byte storage.
DD	DD has the same form as DB except it applies to doubleword quantities.
DQ	DQ has the same form as DB except it applies to four-word quantities.
DT	DT has the same form as DB except it applies to 10-byte packed decimal.
DW	DW has the same form as DB except it applies to word quantities.

TABLE A.8 *(Concluded)*

Operator	Type	Description
MASK	Record specific	The format of this operator is MASK recfield. It returns a bit mask for the field. The mask has bits set for positions included in the field and 0 for bits not included in the field.
WIDTH	Record specific	The format of this operator is WIDTH recfield. It evaluates to a constant in the range 1 to 16 and returns the width of a record or record field.
+	Arithmetic	Returns the sum of two terms. Form: term1 + term2.
-	Arithmetic	Returns the difference of two terms. Form: term1 − term2.
*	Arithmetic	Returns the product of two terms. Form: term1 * term2.
MOD	Arithmetic	Form: term1 MOD term2. It returns the remainder obtained by dividing term1 by term2.
SHL	Arithmetic	Form: term1 SHL term2. It shifts the bits of term1 left by the amount contained in term2. Zeros are filled in the new bits.
SHR	Arithmetic	Same as SHL except the shift is to the right.
EQ	Relational	Form: term1 EQ term2. Returns a value −1 (TRUE) if term1 equals term2, or 0 (FALSE) otherwise.
NE	Relational	Form: term1 NE term2. Returns a value −1 (TRUE) if term1 does not equal term2, or 0 (FALSE) otherwise.
LT	Relational	Form: term1 LT term2. Returns a value −1 (TRUE) if term1 is less than term2, or 0 (FALSE) otherwise.
LE	Relational	Form: term1 LE term2. Returns a value −1 (TRUE) if term1 is less than or equal to term2, or 0 (FALSE) otherwise.
GT	Relational	Form: term1 GT term2. Returns a value −1 (TRUE) if term1 is greater than term2, or 0 (FALSE) otherwise.
GE	Relational	Form: term1 GE term2. Returns a value −1 (TRUE) if term1 is greater than or equal to term2, or 0 (FALSE) otherwise.
AND, OR, and XOR	Logical	These operators have the form term1 (operator) term2 and return each bit position as follows:
NOT	Logical	Form: NOT term. This operator complements each bit of term.

term1 bit	term2 bit	AND	OR	XOR
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

TABLE A.8 IBM MACRO ASSEMBLER OPERATORS

Operator	Type	Description
PTR	Attribute	This operator has the form type PTR expression. It is used to override the type attribute (BYTE, WORD, DWORD, QWORD, or TBYTE) of a variable or the attribute of a label (NEAR or FAR). The expression field is the variable or label that is to be overridden.
Seg-reg, Seg-name	Attribute	The segment override operator changes the segment attribute of a label, variable, or address expression. It has three forms:
Group-name		seg-reg:addr-expression seg-name:addr-expression group-name:addr-expression
SHORT	Attribute	This operator is used when a label follows a JMP instruction and is within 127 bytes of the JMP. It has the form JMP SHORT label and changes the NEAR attribute. A pass 2 NOP instruction is avoided.
THIS	Attribute	The form of this operator is THIS type. The operator produces an operand whose segment attribute is equal to the defining segment, whose offset equals IP, and a type attribute defined by "type." For example, "AAA EQU THIS WORD" yields an AAA with attribute WORD instead of NEAR (if used in the same code segment).
HIGH	Attribute	This operator accepts a number/address argument and returns the high-order byte.
LOW	Attribute	This operator accepts a number/address argument and returns the low-order byte.
SEG	Value returning	This operator returns the segment value of the variable or label.
OFFSET	Value returning	This operator returns the offset value of the variable or label.
TYPE	Value returning	For operand arguments, this operator returns a value equal to the number of bytes of the operand. If a structure name, it returns the number of bytes declared by STRUC. If the operand is a label, it returns 65534 (FAR) and 65535 (NEAR).
SIZE	Value returning	This operator returns the value LENGTH × TYPE.
LENGTH	Value returning	For a DUP entry, LENGTH returns the number of units allocated for the variable. For all others it returns a 1.
SHIFT COUNT	Record specific	This operator is used with the RECORD pseudo-op and is the name of the record field. The format of RECORD is: recordname RECORD fieldname: width. The value of fieldname, when used in an expression, is the shift count to move the field to the far right within the byte or word.

TABLE A.7 SYSTEMS-ORIENTED 80286 AND 80386 INSTRUCTIONS

Instruction	Purpose	Comments
ARPL dest., source	Adjust RPL field of selector	If the RPL field of the selector (protection bits) in dest. is less than the RPL field of source, ZF = 1 and the RDL field of dest. is set to match source.
CLTS	Clear Task Switched Flag	The Task Switch Flag is in the Machine Status Word and is set each time a task change occurs. This instruction clears that flag.
LAR dest., source	Load access rights byte	Destination contains a selector. If the associated descriptor is visible at the called protection level, the access rights byte of the descriptor is loaded into the high byte of source (low byte = 0).
LGDT/LIDT m	Load Global/Interrupt Descriptor Table register	m points to 6 bytes of memory used to provide Descriptor Table values (Global and Interrupt). This instruction loads these tables into the appropriate 80286 registers.
LLDT source	Load Local Descriptor Table register	Source is a selector pointing to the Global Descriptor Table. The GDT should, in turn, be a Local Descriptor Table. The LDT register is then loaded with source.
LMSW source	Load Machine Status Word	The Machine Status Word is loaded from source.
LSL dest., source	Load segment limit	If the Descriptor Table value pointed to by the selector in destination is visible at the current protection level, a limit value specified by source is loaded into this descriptor.
LTR source	Load Task Register	The Task Register is loaded from source.
SGDT/SIDT m	Store Global/Interrupt Descriptor Table register	The contents of the specified Descriptor Table register are copied to 6 bytes of memory pointed to by m.
SLDT dest.	Store Local Descriptor Table register	The Local Descriptor Table register is stored in the word register or memory location specified by destination.
SMSW dest.	Store Machine Status Word	The Machine Status Word is stored in the word register or memory location specified by destination.
VERR/VERW source	Verify a segment for reading or writing	Source is a selector. These instructions determine whether the segment corresponding to this selector is reachable under the current protection level.
STR dest.	Store Task Register	The contents of the Task Register are stored in destination.

TABLE A.6 ADDITIONAL 80386 APPLICATION INSTRUCTIONS

Instruction	Purpose	Comments
BSF dest., source	Bit scan forward	The source word (doubleword) is scanned for a set bit and the index value of this bit loaded in destination. Scanning is from right to left.
BSR dest., source	Bit scan reverse	Scans as in BSF but reverse order.
BT base, offset	Bit test	This instruction loads the bit value from base at offset in the base, into the CF register.
BTC base, offset	Bit test and complement	This instruction loads the bit value from base at offset in the base, into the CF register, and complements the bit in base.
BTR base, offset	Bit test and reset	This instruction loads the bit value from base at offset in the base, into the CF register, and resets the bit to 0.
BTS base, offset	Bit test and set	This instruction is identical to BTR, but the resulting bit is set to 1.
CWDE, CWD	Convert word to doubleword	This instruction converts the signed word in AX to a doubleword in EAX.
CMPSD	Compare doublewords	This instruction compares ES: [EDI] with DS:[ESI].
CDQ	Convert doubleword to quadword	Converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the sign into EDX.
INSD	Input	Input from port DX to ES:[EDI] (doubleword).
LODSD	Load string operand	Load doubleword DS:[ESI] into EAX.
MOVSD	Move data from string to string	Move doubleword DS:[ESI] to ES:[EDI].
MOVSX	Move with sign-extend	Move byte to word, byte to dword, and word to dword with sign extend.
MOVZX	Move with zero-extend	Move byte to word, byte to dword, and word to dword with 0 extend.
OUTSD	Output	Output dword DS:[ESI] to port in DX.
POPAD	Pop all general registers	Pops the eight 32-bit general registers.
POPFD	Pop stack into EFLAGS	Pops the 32-bit stack top into EFLAGS.
PUSHAD	Push all general registers	Pushes the eight 32-bit general registers onto the stack.
PUSHFD	Push EFLAGS onto stack	Pushes the EFLAGS register onto the stack.
SCASD	Compare string data	Compares dwords EAX and ES:[EDI] and updates. EDI.
SETcc dest.	Byte set on condition	Stores a byte (equal to 1), if cc, the condition, is met (following a compare, for example). Otherwise, a value of 0 is stored at the destination.
SHLD dest., Count	Double-precision-shift left	The destination is shifted left by count.
SHRD dest., Count	Double-precision shift right	Same as SHLD but shift is to the right.
STOSD	Store string data	Store EAX in dword ES: [EDI] and update EDI.

TABLE A.5 ADDITIONAL 80286 APPLICATION INSTRUCTIONS

Instruction	Purpose	Comments
BOUND dest.,source	Check array index against bounds	This instruction ensures that an index (destination) is above or equal to the first word in the memory location defined by source. Similarly, it must be below or equal to "source + 2."
ENTER immediate-word, immediate-byte	Make stack frame for procedure parameters	"Immediate-word" specifies how many bytes of storage to be allocated on the stack for the routine being entered. "Immediate-byte" specifies the nesting level of the routine within the high-level source code being entered.
IMUL dest.,immediate	Integer immediate multiply	Does a signed multiplication of destination by an immediate value.
INS/INSB/INSW dest.-string,port	Input from port to string	Transfers a byte or word string from the port numbered by DX to ES:DI. The operand dest.-string determines the type of move: byte or word.
LEAVE	High-level procedure exit	Executes a procedure return for a high-level language.
OUTS/OUTSB/OUTSW port,source-string	Output string to port	Transfers a byte or word string from memory at DS:DI to the port numbered by DX.
POPA	Pop all general registers	Restores the eight general-purpose registers saved on the stack by PUSHA.
PUSH immediate	Push immediate onto stack	This instruction pushes the immediate data onto the stack.
RCL dest.,CL	Rotate left through carry	Same as RCL for 8088 except count can be 31.
RCR dest.,CL	Rotate right through carry	Same as RCR for 8088 except count can be 31.
ROL dest.,CL	Rotate left	Same as ROL for 8088 except count can be 31.
ROR dest.,CL	Rotate right	Same as ROR for 8088 except count can be 31.
SAL/SHL dest.,CL	Shift arithmetic left/ shift logical left	Same as 8088 instructions except count can be 31.
SAR dest.,CL	Shift arithmetic right	Same as 8088 instruction except count can be 31.
SHR dest.,CL	Shift logical right	Same as 8088 instruction except count can be 31.

TABLE A.4 THE COMPARE INSTRUCTION GROUP

Instruction	Purpose	Comments
CMP destination, source	Compare two operands	This instruction causes the source to be subtracted from the destination; however, only the flags are affected. The destination remains unchanged.

`CMPS destination-str source-str` (CMPSB) (CMPSW)	Compare byte or word string	The source string (with DI as an index for the extra segment) is subtracted from the destination string (which uses SI as index). Only the flags are affected and both DI and SI are incremented. A typical sequence of instructions could be
		`MOV SI, OFFSET AAA`
		`MOV DI, OFFSET BBB`
		`CMPS AAA, BBB`

Table A.5 contains additional instructions specific to the 80286 microprocessor. Table A.6 contains similar instructions for the 80386 microprocessor. Both of these microprocessors are designed to operate in Protected Mode. The computer used in writing this book was a PC AT with a 6-MHz throughput rate, as opposed to the 4-MHz clocks associated with the IBM PC. Expansion to the PS/2 systems should yield even faster performance than the 80286-based system used here. Table A.7 presents the system-oriented instructions available for the 80286 and 80386. These instructions are not normally accessible by the applications programmer.

Table A.8 contains the Macro Assembler operators available to the programmer, Table A.9 contains the pseudo-operations available to the Macro Assembler programmer, and Table A.10 contains a set of operators to be used with the macro pseudo-op.

Table A.11 illustrates the coprocessor instruction set. These instructions begin with the letter "F" and most rely on the use of the coprocessor stack registers, ST(0) through ST(7), for implementation. These stack registers serve as the general-purpose registers for the coprocessor. Usually, ST(0) serves as the source register and ST(1) as the destination, particularly in implicit instructions such as FADD when used without operands.

TABLE A.3 (*Concluded*)

Instruction	Purpose	Comments
JE short-label (JZ)	Jump if equal/ if zero	If the last operation to change ZF set this flag (gave a result of 0), JE will cause a jump to occur. This is a short-label jump.
JG short-label (JNLE)	Jump if greater/if not less or equal	If ZF = 0 and SF = OF, the JG instruction will cause a jump to short-label. This instruction is used with signed operands.
JGE short-label (JNL)	Jump if greater or equal/if not less	This instruction is the same as JG except ZF is not considered. If SF = OF, the jump occurs. This is a short-label instruction with signed operands.
JL short-label (JNGE)	Jump if less/if not greater or equal	If SF ≠ OF, the JL instruction will result in a jump. This instruction is short-label with signed operands.
JLE short-label (JNG)	Jump if less or equal/if not greater	If ZF = 1 or SF ≠ OF, the JLE instruction yields a short-label jump. The instruction is used with signed operands.
JMP target	Jump	This is a direct and unconditional jump.
JNC short-label	Jump if no carry	If CF = 0, this instruction yields a short-label jump.
JNE short-label (JNZ)	Jump if not equal/ if not zero	If ZF = 0, this short-label jump will occur.
JNO short-label	Jump if no overflow	If OF = 0, this short-label jump will occur.
JNB short-label (JPO)	Jump if no parity/ if parity odd	If PF = 0, this short-label jump will occur.
JNS short-label	Jump if no sign/if positive	If SF = 0, this short-label jump will occur.
JO short-label	Jump on overflow	If OF = 1, this short-label jump will occur.
JP short-label (JPE)	Jump on parity/ if parity even	If PF = 1, this short-label jump will occur.
JS short-label	Jump on sign	If SF = 1, this short-label jump will occur.

Table A.3 presents the jump instruction group. These instructions are used to achieve execution control within the langauge. They accomplish this by providing the capability to change the instruction execution sequence based on the outcome of various tests. These tests can be performed by various instructions that change the state of flags in the flags' register. Table A.4 illustrates the compare instructions, which serve as a basis for accomplishing such testing. These instructions change the flags without changing the source or destination.

TABLE A.3 JUMP INSTRUCTION GROUP

Instruction	Purpose	Comments
JA short-label (JNBE)	Jump if above/ if not below or equal	This jump is used in conjunction with the carry and zero flags. If either or both are set, no jump occurs. Suppose two operands are compared; then if the destination is greater than the source (above) CF = ZF = 0 and the jump occurs. The jump is within −128 to +127 bytes (short-label) and unsigned operands are used.
JAE short-label (JNB)	Jump if above or equal/if not below	This jump is similar to JA except only the carry flag is examined. If a previous compare, for example, is performed and the destination is greater or equal to the source (above or equal), CF = 0 and the jump occurs. This is a short-label instruction with unsigned operands.
JB short-label (JNAE) (JC)	Jump if below/if not above or equal/if carry	This jump is the opposite of JAE. If the carry flag is set, the jump will occur. Suppose a previous compare is performed and the destination is less than the source (below); CF = 1 and the jump occurs. This is a short label instruction with unsigned operands.
JBE short-label (JNA)	Jump if below or equal/if not above	This jump is the same as JB except it also takes place if the zero flag is set (below or equal). It is short-label with unsigned operands.
JCXZ short-label	Jump if CX is zero	Suppose an instruction sequence causes the count register (CX) to decrement. When CX reaches 0, control would transfer to the short-label after execution of JCXZ. This is a short-label jump.

TABLE A.2 *(Concluded)*

Instruction	Purpose	Comments
DAS	Decimal subtract adjust	Adjust for decimal subtraction.
I/O		
IN acc, port	Input byte/word	The byte/word contents of port are loaded into AL/AX.
OUT port, acc	Output byte/word	The contents of the accumulator are sent to port output.
Miscellaneous		
XCHG dest,src	Exchange	Exchanges the source (src) with dest.
XLAT src-table	Translate	BX is loaded with a table address. AL contains a location number (byte) in the table and this byte is replaced in AL.

family of microprocessors. These instructions are grouped by category:

1. Arithmetic
2. Logical
3. Move
4. Load
5. Loop
6. Stack
7. Count
8. Flags
9. Shift
10. Rotate
11. Store
12. String
13. Convert
14. Control
15. ASCII
16. Decimal
17. I/O
18. Miscellaneous

TABLE A.2 *(Continued)*

Instruction	Purpose	Comments
RCR dest,cnt	Rotate right through carry	Rotates dest right in wrap-around fashion cnt bits where cnt is in CL.
ROL dest,cnt	Rotate left	Same as RCL except the high-order bit rotates into CF as well as the low-order bit.
ROR dest,cnt	Rotate right	Same as ROL except to the right.
Store		
STOS dest-str	Store byte or word string	Transfers a byte (word) from AL (AX) to the location pointed to by DI.
SAHF	Store AH in flags	Transfers the value in AH to the flags register.
String		
REP	Repeat string operation	Causes the string operation that follows to repeat until CX = 0, ZF = 1.
REPNE	Repeat string operation	Same as REP except ZF = 0.
SCAS dest-str	Scan byte or word string	Subtracts the dest-str from AL (AX) one byte at a time and affects the flags.
Convert		
CWD	Convert word to doubleword	Sign extends AX into DX.
CBW	Convert byte to word	Sign extends AL into AX.
Control		
CALL target	Calls a procedure	Calls a procedure (target).
RET	Return from a procedure	Returns control to the calling routine.
ESC ext-opcode, src	Escape	Initiates the ext-opcode with operand src.
LOCK	Lock bus	Closes the bus to access.
NOP	No operation	A do-nothing operation.
WAIT	Wait	A bus cycle state used for synchronization.
ASCII		
AAA	ASCII adjust for addition	Adjusts the sum for an ASCII numerical value following addition.
AAD	ASCII adjust for division	Adjusts the quotient for ASCII numerical value following division.
AAH	ASCII adjust for multiply	Adjusts the product for ASCII numerical value following multiplication.
AAS	ASCII adjust for subtraction	Adjusts the difference for an ASCII numerical value following subtraction.
Decimal		
DAA	Decimal add adjust	Adjust for decimal addition.

TABLE A.2 *(Continued)*

Instruction	Purpose	Comments
LAHF	Load AH from flags	Transfers the flags to AH.
LDS dest,src	Load data segment register	Loads a 32-bit address into DS and dest (offset).
LEA dest,src	Load effective address	Transfers the offset of src to dest.
LES dest,src	Load extra segment register	Loads a 32-bit address into ES and dest (offset).
Loop		
LOOP short-label	Loop until count complete	Control is transferred to short-label if CX $\neq$ 0 and CX is decremented.
LOOPE short-label	Loop if equal	Same as LOOP but control transfers if ZF = 1, as an additional requirement.
LOOPNE short-label	Loop if not equal	Same as LOOPE except ZF must equal 0.
Stack		
POP dest	Pop word off the stack	Transfers a word from the stack (pointed to by SP) to dest.
POPF	Pop flags off the stack	Transfers the word from the stack top to the flags register.
PUSH src	Push word onto the stack	src is placed on the stack top.
PUSHF	Push flags onto the stack	The flags register is loaded onto the top of the stack.
Count		
DEC dest	Decrement	Subtract one from dest.
INC dest.	Increment	Add one to dest.
Flags		
CLC	Clear carry flag	Sets Cf = 0.
CLD	Clear direction flag	Sets DF = 0.
CLI	Clear interrupt flag	Sets IF = 0.
CMC	Complement carry flag	Changes setting of CF.
STC	Set carry flag	Sets CF = 1.
STD	Set direction flag	Sets DF = 1.
STI	Sets interrupt flag	Sets IF = 1.
Shift		
SAL dest,cnt	Shift arithmetic left	Shifts dest cnt bits left. CL contains cnt.
SHL dest,cnt	Shift logical left	Same as SAL.
SAR dest,cnt	Shift arithmetic right	Same as SAL except shift if to the right.
SHR dest,cnt	Shift logical right	Same as SAR.
Rotate		
RCL dest,cnt	Rotate left through carry	Rotates dest left in wrap-around fashion cnt bits where cnt is in CL.

memory area such as a shared segment defined using

`\SHAREMEM\SDAT.DAT`

where the path is common to both processes.

3.6. RAM semaphores are a good candidate for intertask communications when each thread is within the same process. When conducting interprocess communications, system semaphores provide a good method for synchronization because they share a common system memory area and can be passed back and forth.

3.7. The macro @pushs loads a 32-bit address for parm on the stack. The macro @pushd loads the 32-bit contents of the double-word parameter parm onto the stack. The fourth parameter of @DosWriteQueue must contain a pointer value, and the address of this the double word containing this value is not of interest.

3.8. By clearing the semaphore using @DosSemClear.

3.9. There could be a segment protection violation, as both processes contend for the speaker.

3.10. A pipe was used to pass buffered data directly. The queue, on the other hand, was used to pass pointers to buffers. The queue employed a shared memory area allocated by @DosAllocSeg, whose buffer address was passed via the queue.

3.11. See Table 3.1.

3.12. A shared segment is allocated with @DosAllocShrSeg and is used when two or more processes need to access a common buffer in interleaved or multiple asynchronous fashion. The fact that both have simultaneous access must be regulated using semaphores, for example. A giveable segment, on the other hand, would be used when a process desires to write to a common segment and then release the segment so that another process can access the "given" segment.

3.13. ***Process 1***

```
...
read_hdl          dw          ?
write_hdl         dw          ?
buf_flag          dw          256  ;for example
bytes_written     dw          ?
...
msize             dw          ?
ssell             dw          ?
shrname           db          '\SHAREMEM\SDAT1.DAT',0
...

@DosAllocShrSeg   msize,shrname,msell       ;allocate segment
...
@DosMakePipe      read_hdl, write_hdl, buf_flag ;Create pipe
...
@DosCreateSem     ...                 ; Synchronize
...
```

```
@DosWrite  write_hdl,message,length,bytes_written  ;Transfer
...
@DosSemSet         ...                 ;set  semaphore
...
@DosExecPgm        ...                 ;execute  child
@DosWait           ...                 ;wait  child  completion
...
```

Process 2

```
shrsel  dw  ?
shrname  db  '\SHAREMEM\SDAT1.DAT',0
...
read_hdl  ...
bytes_read  ...
...
@DosOpenSem  ...        ;open  semaphore
...
@DosGetShrSeg...        ;get  shared  segment
...
;load  read  handle  and  message  length  from  shared  segment
buffer
...
@DosRead  read_hdl,  buffer,  length,  bytes_read  ;read  buffer
...
@DosSemClear  ...       ;clear  semaphore
```

3.14. *Process 1*

```
...
q_hdl  dw  ?
q_name  db  '\QUEUES\QDAT.DAT',0
...
@DosCreateQueue  q_hdl,...     ;create  queue
...
@DosExecPgm...                 ;execute  child
...
@DosReadQueue  q_hdl,...       ;read  queue  buffer
...
;transfer  message  using  32-bit  queue  pointer  to  buffer
...
@DosFreeSeg...                 ;free  allocated  segment  (process)
...
@DosCloseQueue...              ;close  queue
...
```

Process 2

```
...
q_hdl                              dw  ?
q_name                  db  '\QUEUES\QDAT.DAT',0
```

```
...
q_w                 dw  0
q_v                 dd  0
q_rr                dw  0
...
@DosOpenQueue  ...                    ;open queue
...
@DosAllocSeg...                       ;allocate segment
...
;load segment with message
...
@DosGiveSeg...                        ;get read selector
...
@DosWriteQueue1...                    ;write address to queue
...
@DosFreeSeg      q_w                  ;free allocated segment
...
@DosCloseQueue...
```

3.15. The intraprocess thread appears as a FAR entry point within the same process segment. An interprocess thread appears as a FAR call to an entry point in a new segment, and it must be started with DosExecPgm.

3.16. The calls

```
@DosAllocSeg
@DosReAllocSeg
```

actually create and reallocate global memory space. This memory can exist as virtual addresses (on disk) or as actual data RAM. The call

```
@DosSubAlloc
```

subdivides an *existing* segment into smaller blocks of local memory. This call returns a block offset to be used as the start of the memory block. No consideration of previous writes to the segment is made; hence a block can overwrite a memory area if not properly handled. A recommended procedure is to suballocate the memory segment as early as possible, thereby obtaining a block offset from which to work.

Chapter 4

4.1. When IBM and Microsoft developed the two versions of the Toolkit, they modified the structures and calling sequences between them. Calling the physical screen buffer in Version 1.0 required a structure PhysBufData. In the Version 1.1 this structure was renamed _VIOPHYSBUF and the structure members have different tags between versions. Hence many of the Version 1.1 Toolkit entities require slightly different nomenclature than that of Version 1.0.

4.2. Many of the Standard C I/O library functions have been defined under Microsoft C Optimizing Compiler Version 5.1 to run as reentrant routines. Hence these library routines are callable in the usual fashion from Protected Mode.

4.3. All formal parameters specified as type int need not be type specified in the formal parameter list. Hence, when passing parameters, only types other than int need be specified.

4.4. The input value x is of int type, hence can be signed. The range of such signed variables is [–32,768, 32,768]. Had this parameter been of type double or float, a much greater range of values would be permissible.

4.5. The normal C local stack calling convention starts with the Nth parameter and continues to load the stack down to parameter 1. The API services require conventional loading from parameter 1 to N. In the Toolkit a type APIENTRY is defined using the pascal convention, which reorders the formal parameters on the stack from 1 to N.

4.6. The code

```
CHAR FAR *ptr;
```

defines a FAR pointer, ptr, which points to a byte value using a 32-bit address. The code

```
CHAR FAR *shrname = "\\SHAREMEM\\SDAT1.DAT";
```

defines a FAR pointer, shrname, which points to a string value using a 32-bit address and associates the string value with this 32-bit address.

4.7. This function generates a full 32-bit address for a FAR call. The variables sel and off correspond to selector and offset, respectively, and must be obtained separately.

4.8. This declaration associates a FAR pointer address of 0xB8000 with a BYTE value specified as the structure element PVBPrt2.pBuf. Note that this structure element is specified as part of the physical buffer structure _VIOPHYSBUF. The value in question is the start address (32-bit) for the physical screen buffer.

4.9. The return from sin() is double precision; hence the defining relation should be

```
...
y = (float)(sin(2. * PI * t));
```

4.10. The dot attribute for wdot() is 1, and the dot attribute for uwdot() is 0.

4.11. The column values represent horizontal increments, and the row values represent vertical increments.

4.12. This is the character code for putting the Epson dot matrix printer (FX-85) into graphics mode. The values 0x1B and 0x4V specify

```
ESC k
```

and 64 is the difference between 256 and 320. Here the "1" indicates one block of 256 columns plus "64," to get 320 columns in the printer graphics mode. The ESC k indicates that the printer must go to graphics mode.

4.13. This sends the Epson FX-85 printer the command

```
ESC A 8
```

which changes the printer output to case 8/72-inch spacing for lines. This removes any extra vertical spacing that might appear in the output and ensures that the eight dots of vertical spacing will butt together as each vertical set of pins is executed for each line (25 vertical lines of eight pins per line).

4.14. RAM semaphores.

4.15. The calls all employ type definitions for the formal parameters unless the parameters are of basic integer type (such as INT, UINT, USHORT, SHORT).

4.16. One process must establish the semaphore, open the second process, and *wait* until the second process is complete. The second process maintains synchronization by executing with the first process blocked until

```
DosSemClear()
```

is issued by the second process. This allows the first process to cease being blocked by

```
DosSemWait()
```

and continue execution beyond this instruction. Threads are handled in identical fashion.

4.17. To pass the pipe read handle, read_hdl, to the second process, where it is used to locate the pipe buffer and transfer the pipe message using

```
DosRead()
```

4.18. The thread's stack is separate from the calling thread's stack. Hence the compiler will detect an overflow condition because the second thread's stack is "outside" the stack originally defined for the overall process. To avoid compiler errors with the Microsoft C Optimizing Compiler Version 5.1, for example, a compiler option of -Gs must be used.

4.19. The programs have been debugged and it is assumed that no error checking is needed intrinsically. Good form would retain such error checking when dynamic errors can creep in. For clarity, we have avoided them in the code dynamics.

4.20. The points

$$
\begin{aligned}
(x,y,z) \;=\; & (1, 0, 0) \\
& (0, 1, 0) \\
& (0, 0, 0) \\
& (0, 0, 1)
\end{aligned}
$$

4.21. From the library cgraph.lib.

Chapter 5

5.1. Load the routine as a run-time dynamic linker library (DLL).

5.2. The dynamic link library (.dll) executable module is generated from a group of object modules as

```
link (group object modules),
     (.dll module),,(libraries),(.def module)
```

Here the definition file must start with the LIBRARY keyword. Next, the .lib file is created from

```
implib (.lib file) (.def module)
```

This definition file is the same one used earlier to define the .dll module.

5.3. The return address offset.

5.4. It appears in the definition file with LIBRARY, not NAME.

5.5. EXPORTS are the names of routines contained in a DLL which will be available to be called by other modules. IMPORTS are the names of external routines to be used by the DLL.

5.6. The two services needed are

```
DosLoadModule
DosGetProcAddr
```

The first is required to load the run-time .dll file, which must be specified in the initial calling program. This is the calling program's link with the DLL. The second is needed to specify the DLL procedure entry point so that a simple FAR call to this entry point can be made.

5.7. You would choose a flowchart because it illustrates the dynamic decision-oriented performance of the program. There are several types of flowcharts: literal and functional. The former spell out each individual programming step and tend to be very detailed. Functional flowcharts are more desirable and address program activity in a functional sense; that is, each block constitutes a major activity in the sequence of program flow, and this activity usually consists of many program steps. A Structure Chart merely illustrates the subordinate relationships among the program components. This device is most useful for providing the user with an overview of the program and a general picture of the major modules appearing in the program.

5.8. The dominant characteristic of a C function is a single entry and exit point and a single return value. These features serve to make the program structure very orderly with a downward flow of activity to the code instead of the varied branching found in programs typified by FORTRAN code, for example. In FORTRAN programs, the unrestricted use of GOTOs and conditional branching frequently makes following the program flow difficult.

5.9. C programs can efficiently use global variables when large array components of databases are to be manipulated. In this case the use of arrays as local variables would greatly expand the stack area and result in inefficient memory allocation. The difficulty with using global variables in any implementation is that external modules that call such variables frequently lose track of the time history of the variables. By treating variables as locally defined, the complete time picture of the performance of the variable is available in the accessing module.

5.10. This is accessible only through the Presentation Manager, which did not become available until OS/2 Version 1.1 was issued.

5.11. Three-dimensionally the x-axis points out of the image away from the plane of the CRT. Hence a two-dimensional display, such as a CRT, can only illustrate the image of a three-dimensional surface as it is projected onto such a two-dimensional display. This projection is achieved by setting one of the coordinate sets equal to zero.

5.12. Some of the hidden facets appearing in the three-dimensional surface of Figures 5.19 through 5.23 actually have positive direction normals to the facet surface. Hence these surfaces, even though hidden, will not satisfy the criterion that the normal points into the plane of the CRT.

5.13. This equation has the form

$$\lim_{u \to 0} A^2 \frac{\sin^2 u}{u^2}$$

using

$$\sin u = u - \frac{u^3}{3!} + \frac{u^5}{5!} - \ldots$$

This becomes

$$\lim_{u \to 0} A^2 \left[1 - \frac{u^2}{3!} + \frac{u^4}{5!} - \ldots \right]^2 = A^2$$

5.14. This routine must be called before setting the CGA mode because it prints a request to the text screen (asking for the name of the data file).

5.15. The selector, MMI, would no longer be referenced.

Index

A

L

M

N

O

P

W

X

Z

2.7. ...

```
kbd_buf          db        80
lkbd_buf         dw        $-kbd_buf
iowait           dw        0
kbdhdl           equ       0
freq             dw        1000
dur              dw        5000
...
@KbdStringIn     kbd_buf,lkbd_buf,iowait,kbdhdl
@DosBeep         freq,dur
...
```

2.8. Yes, all calls to the API can be made in full form, where each push and pop, as well as EXTRN declaration, is stated explicitly according to the rules of OS/2. The toolkit simply provided a set of assembler .inc files and C .h files that facilitated usage of the API services through very functional macros.

2.9. The key assumption is that segment selectors can be treated as segment addresses. Since the 80286 accesses segments using the selectors, the selector value must reside in a segment register. The address is then calculated in the usual Protected Mode fashion, where the segment selector acts as a segment address. The use of segment override addressing, such as

```
es:[bp]
```

simply permits specification of an address in the usual fashion, where the segment selector is made to correspond to the physical segment address when VioGetPhysBuf is exercised.

2.10. They represent FAR locations because the entry points are called from external API modules, hence a 32-bit address must be specified.

2.11. No hierarchy should have a single child subordinate to a parent. The box 310 should be absorbed in 300.

2.12. The command is

```
@DosWrite  dev_hand,in_buffer5,bytesin3,bytesout
```

where the undefined parameter is

```
in_buffer5  db  1BH,41H,0CH
```

2.13. It is intuitive that they cannot be preempted by an OS/2 task switch, or the possibility of losing data from the device would occur.

2.14. To access the screen buffer (physical) properly, the screen must be locked; hence if scr_ld is to load scr_buffer with the screen context, it must be locked. If prtscr is executed when the screen is locked, it could dominate access time for the physical display buffer. Hence the program should load a temporary buffer, release the screen context, and then begin the print operation.

2.15. Ten complete raster segments.

2.16. The DosExitCritSec corresponds to exit of a critical section of execution for a thread and returns control to a process. This could be used, for example, when an independent thread has a particular piece of code that must execute prior to any other operation for the parent process. Then it would be desirable to monitor and ensure that this code executed, before continuing. Clearly, this could be dynamic and change with the active chronology of execution.

2.17. DosExit is used to terminate an application and return to OS/2. All other returns NEAR or F.

Chapter 3

3.1. The drivers mentioned operate from the kernel, level 0. They must originate here because they have to be protected ahead of all other code. We cannot have a disk-write preempted in the middle, nor can we tolerate "jerky" mouse cursor movement as the mouse position changes.

3.2. The macro calls admittedly remove a layer of detail from the program code. This layer would tend to expand the code by a factor of 4 to 7. All the pushes to the stack have been suppressed prior to each API call and the call takes on the form of a higher-level-language (HLL) function call. The data area tends to expand considerably with all the macro parameter definitions, but the actual executable code remains compact. This requires the programmer to develop a general familiarity with the macro calls at the level of the IBM Programmer's Toolkit or Appendix C of this book. Once this familiarity has developed it is a very easy matter to read the resulting "structured" code and follow the flow of execution. Hence maintenance becomes an easy task. Clarity (of how the code executes) is also paramount, and much more so under the macro call format. The macro calls do, however, inhibit debugging in that the in-line code is missing. If the user prints a copy of the list file with macros expanded, tracing the source code is still an easy matter. In general, these approaches tend to be a matter of preference based on the programmer's orientation. We favor the HLL appearance of the code. It makes functional performance of the code the primary mechanism to be emphasized. Expansion of the in-line code makes it more obscure from a functional viewpoint but easier (and essential) to debug.

3.3. For the segment to be sharable, bit 0, to be sharable through @DosGiveSeg, or bit 1, to be sharable through @DosGetSeg, must be set in the flags word (the third parameter in the calling list). Bit 2 of this same flags word must be set if the segment is to be discardable.

3.4. The write to the huge segment must use the proper selector. When crossing the 64K-byte boundary the program must access a new but contiguous selector.

3.5. There must be some common link between the two processes. Usually, this is a common element name such as

```
\SEM\SDAT.DAT
```

or

```
\QUEUES\QDAT.DAT
```

which appears in both processes and is the same. The system then provides the connection. Alternative to this is the passing of a selector or printer in a common

Programming the OS/2 Kernel

J. TERRY GODFREY

Practical and versatile, this guide addresses the IBM® Operating System/2 (OS/2) Kernel services—the video (Vio), disk operating system (Dos), keyboard (Kbd), and mouse (Mou) functions. It demonstrates techniques for programming in an advanced multitasking environment. Other significant features of this sourcebook include:

- detailed examples of programs that actually accomplish things, unlike the small, trivial programs found in many other books
- concentration on the OS/2 Full-Screen Command Mode, which utilizes the entire display for presentation of a single program and makes other programs invisible
- use of keyboard services to pause graphics screens
- use of the IBM macro calls to the Application Program Interface (API)—book assumes that readers have access to the IBM API macros
- illustration of the Protected Mode Graphics
- extensive program examples similar in length and complexity to real-world cases

PROGRAMMING THE OS/2 KERNEL is the ideal reference for all readers interested in the extensive memory management afforded by IBM's OS/2.

Also by Terry Godfrey:

Applied C: The IBM Microcomputers, 1990 (03968-5)

IBM Microcomputer Assembly: Beginning to Advanced (44950-4)

ISBN 0-13-723776-6

90000

9 780137 237760

PRENTICE HALL
Englewood Cliffs, N.J. 07632