IBM

International Technical Support Centers

OS/2 Version 2.0
Volume 1: Control Program

GG24-3730-00

OS/2 Version 2.0
Volume 1: Control Program

Document Number GG24-3730-00

April 1992

International Technical Support Center
Boca Raton

Take Note

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xvii.

First Edition (April 1992)

This edition applies to OS/2 Version 2.0.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, International Technical Support Center
Dept. 91J, Building 235-2 Internal Zip 4423
901 NW 51st Street
Boca Raton, Florida 33432 USA

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Abstract

This document describes the Control Program component of OS/2 Version 2.0. It forms Volume 1 of a five volume set; the other volumes are:

- *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*, GG24-3731
- *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, GG24-3732.
- *OS/2 Version 2.0 - Volume 4: Application Development*, GG24-3774.
- *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775.

The entire set may be ordered as *OS/2 Version 2.0 Technical Compendium*, GBOF-2254.

This document is intended for IBM system engineers, IBM authorized dealers, IBM customers, and others who require a knowledge of the operating system features, functions, and implementation in OS/2 Version 2.0.

This document assumes that the reader is generally familiar with the function provided in previous releases of OS/2.

PS (174 pages)

Acknowledgements

The project leader and editor for this project was:

Hans J. Goetz
International Technical Support Center, Boca Raton

The authors of this document are:

Bill Bolton
Westpac Banking Corporation, Australia

Calvin Bradshaw
IBM Australia

Gert Ehing
IBM Germany

Bo Falkenberg
IBM Denmark

Darryl Frost
ISM South Africa

Eddie Griborn
IBM Sweden

Tim Sennitt
IBM UK

Neil Stokes
IBM Australia

Katsutoshi Suzuki
IBM Japan

This publication is the result of a series of residencies conducted at the International Technical Support Center, Boca Raton.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Craig Bennett
IBM Programming Center, Boca Raton.

Sam Casto and his staff
IBM Programming Center, Boca Raton.

Phil Doragh
IBM Programming Center, Boca Raton.

Ari Erev
IBM Israel, Hebrew Competence Group.

Alexandr Gregor
International Technical Support Center, Boca Raton.

Alfredo Gutiérrez
IBM EMEA Education Center, Boca Raton.

Jamie Jamison
IBM Programming Center, Boca Raton.

John Keenleyside
IBM OS/2 C Library Development, Toronto.

Michael Kogan
IBM Programming Center, Boca Raton.

Debbe McCombie
IBM Programming Center, Boca Raton.

Thanks also to the many people, both within and outside IBM, who provided suggestions and guidance, and who reviewed this document prior to publication.

Thanks to the following people for providing excellent tools, used during production of this document:

Dave Hock (CUA Draw)
IBM Cary.

Jürg von Känel (PM Camera)
IBM Yorktown Heights.

Contents

Figures

Tables

Special Notices

This publication is intended to help customers and system engineers to understand and utilize the new features in Version 2.0 of OS/2. The information in this publication is not intended as the specification of the programming interfaces that are provided by the Programming Tools and Information package for use by customers in writing programs to request or receive its services. See the PUBLICATIONS SECTION of the IBM Programming Announcement for OS/2 Version 2.0.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX
C/2
IBM
Micro Channel
Operating System/2
OS/2
PC AT
Personal System/1
Personal System/2
Presentation Manager
PS/1
PS/2
SAA
Systems Application Architecture
WIN-OS/2
Workplace Shell

The following terms, which are denoted by a double asterisk (* *) in this publication, are trademarks of other companies.

CompuServe is a trademark of CompuServe Incorporated.
HP and Hewlett-Packard are trademarks of Hewlett-Packard Corporation.
Intel is a trademark of Intel Corporation.
Logitech is a trademark of Logitech, Incorporated.
Lotus is a trademark of the Lotus Development Corporation.
MicroMaster is a registered trademark of AOX Inc.
Microsoft is a trademark of Microsoft Corporation.
Windows is a trademark of Microsoft Corporation.
Xenix is a trademark of Microsoft Corporation.
386, 486, SX are trademarks of Intel Corporation.
80286, 80386 and 80486 are trademarks of Intel Corporation.

Preface

This document is intended to provide readers with an understanding of the functions provided by the Control Program component in OS/2 Version 2.0 and its implementation. The document should enable readers to make an evaluation of the OS/2 Version 2.0 product and its components.

The document contains information on the implementation of operating system functions such as task management, memory management, and the 32-bit application programming interfaces provided by OS/2 Version 2.0. Information is also provided on hardware support and interrupt management, application debugging aids within the operating system, and the support for multiple bootable fixed disk partitions. The document also discusses pre-installation planning considerations, and the migration of existing applications from 16-bit versions of OS/2 to OS/2 Version 2.0.

This document is intended for:

- Planners and technical support personnel who require an understanding of the internal function implementation in OS/2 Version 2.0.
- Programmers who wish to develop applications which use the 32-bit application programming interfaces and extended flat memory model provided under OS/2 Version 2.0, and/or wish to migrate applications from previous versions of OS/2.

The code examples used in this document are available in electronic form via CompuServe** or through a local IBM Support BBS, as package RB3730.ZIP. IBM employees may obtain the code examples from the GG243730 PACKAGE on OS2TOOLS.

The document is organized as follows:

- *Chapter 1, "Overview"* provides a brief introduction to the topics covered in this document.

  This chapter is recommended for all readers of the document.

- *Chapter 2, "Memory Management"* describes the memory management under OS/2 Version 2.0, including the implementation of the 32-bit flat memory model and memory paging. The chapter also describes the allocation and protection of memory objects, and support for 16-bit applications under OS/2 Version 2.0.

  This chapter is recommended for those readers who require an understanding of the way in which OS/2 Version 2.0 manages the allocation of real and virtual memory, and the way in which this implementation differs from that in previous versions of OS/2. Those readers who require a knowledge of the hardware implementation of these components may also read Appendix A, "Intel 80386 Architecture."

- *Chapter 3, "Task Management"* describes the task management, dispatching implementation and interrupt handling under OS/2 Version 2.0. Process synchronization through the use of semaphores is also discussed.

  This chapter is recommended for those readers who are concerned with the implementation of multitasking under OS/2 Version 2.0.

- *Chapter 4, "Debugging Support"* describes the built-in application trace and debugging aids provided as part of OS/2 Version 2.0, which enable debugger and trace applications to provide additional information.

  This chapter is recommended for those readers who desire an understanding of these capabilities, or who may wish to design their own trace and/or debug applications.

- *Chapter 5, "Installation Considerations"* discusses pre-installation planning and installation considerations to be borne in mind when installing OS/2 Version 2.0, particularly when installing it over existing versions of OS/2. Hardware, memory, and fixed disk space requirements are also described.

  This chapter is intended primarily for planners who will be developing implementation schemes for OS/2 Version 2.0.

- *Chapter 6, "Hardware Considerations"* discusses the OS/2 Version 2.0 device driver structure, changes in the OS/2 V2.0 file system handlers and other topics related to hardware support.

  This chapter is recommended for those readers who require information about OS/2 V2.0 support for IBM and OEM personal computers and related hardware, such as disk subsystems.

- *Chapter 7, "Boot Manager"* describes the capability for multiple bootable fixed disk partitions, which is supported by OS/2 Version 2.0. This support enables different operating systems to be installed on different disk partitions within the same machine, and for the active partition to be switched in order to boot the machine with an alternative operating system.

  This chapter is intended for those planners who may wish to have multiple operating systems installed on machines in their enterprise.

- *Chapter 8, "National Language Considerations"* describes the support provided within OS/2 Version 2.0 for languages other than English, including support for double-byte character sets.

  This chapter is recommended for those readers who wish to use languages other than English.

The following appendixes are included in this document:

- *Appendix A, "Intel 80386 Architecture"* provides an overview of the Intel 80386 microprocessor architecture, with an emphasis on the exploitation of this architecture by OS/2 Version 2.0.

  This chapter is recommended for those readers who desire a more complete understanding of OS/2 Version 2.0's use of 80386 capabilities.

- *Appendix B, "Micro Channel Architecture and SCSI"* provides a brief description of the IBM Micro Channel architecture, as well as the Subsystem Control Block (SCB) architecture and SCSI.

  This chapter is recommended for those readers who desire a deeper understanding of the exploitation of these architectures in OS/2 Version 2.0.

- *Appendix C, "Lab Session - 32-Bit Memory Model"* provides an agenda for a laboratory session which deals with the 32-bit memory model and multithreading capabilities of OS/2 Version 2.0.

  This chapter is intended for those readers who will use this document as the base material for an OS/2 Version 2.0 classroom course.

Related Publications

The following publications are considered particularly suitable for a more detailed discussion of the topics covered in this document.

Prerequisite Publications

- *OS/2 Version 2.0 Installation Guide*
- *OS/2 Version 2.0 Overview Guide*
- *OS/2 Version 2.0 Online Documentation.*

Additional Publications

- *OS/2 V1.2 Standard Edition Internals and Evaluation*, GG24-3466
- *OS/2 V1.3 Volume 1: New Features*, GG24-3630
- *OS/2 V1.3 Volume 2: Print Subsystem*, GG24-3631
- *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*, GG24-3731
- *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, GG24-3732
- *OS/2 Version 2.0 - Volume 4: Application Development*, GG24-3774
- *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775
- *OS/2 Version 2.0 Remote Installation and Maintenance*, GG24-3780
- *Intel 80386 System Software Writer's Guide*, ISBN 1-55512-023-7
- *The Design of OS/2*, Harvey M. Deitel and Michael J. Kogan, Addison Wesley 1992 ISBN 0-201-54889-5 (SC25-4005)
- *Micro Channel Architecture Bus Master Release 1.0*, GG24-3477
- *SCSI - Architecture and Implementation*, GG24-3507.

Chapter 1. Overview

IBM* OS/2* Version 2.0 is an advanced multitasking, single-user operating system for IBM Personal System/2* computers and other machines equipped with the Intel** 80386** or compatible microprocessors. It inherits a rich set of features from previous versions of OS/2, such as support for multitasking, multiple threads, dynamic linking, interprocess communication, a graphical user interface, and a graphics programming interface. Features such as a High Performance File System, Extended Attributes, and long filenames are also available in OS/2 Version 2.0.

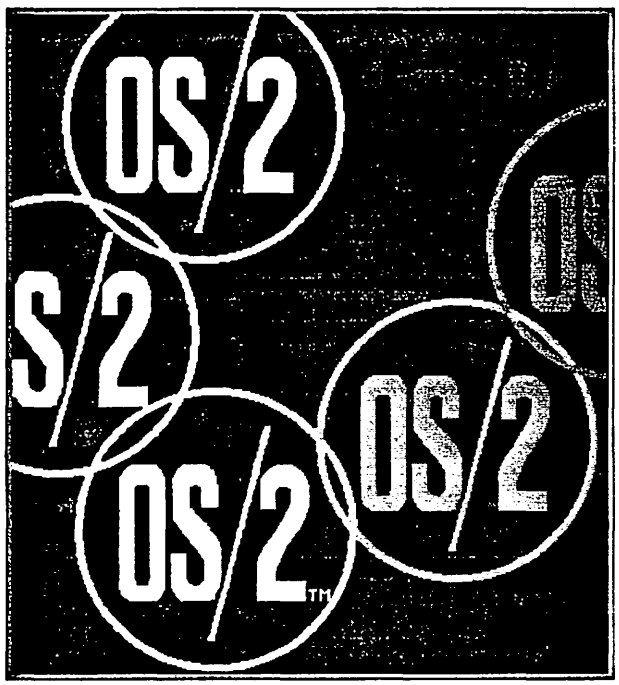


Figure 1. The New OS/2 Version 2.0 Logo

Unlike previous versions of OS/2, Version 2.0 takes advantage of the advanced features of the Intel 80386 processor architecture, such as a 32-bit memory model, paged virtual memory, and an enhanced processor instruction set. More extensive use is also made of IBM's Micro Channel* architecture for improved hardware performance.

OS/2 Version 2.0 requires the features of the Intel 80386 or compatible 32-bit microprocessors, and therefore does not run on computers that use the Intel 80286** processor, or its predecessors. In order to maintain compatibility, OS/2 Version 2.0 supports applications written for previous versions of OS/2 by providing both a 16-bit as well as a 32-bit application programming interface, allowing existing applications to execute under OS/2 Version 2.0 without modification. Note, however, that only the 32-bit interface provides the full features and performance of OS/2 Version 2.0, and that while existing OS/2 Version 1.x applications will execute under OS/2 Version 2.0, they must be modified in order to take full advantage of the new features of Version 2.0.

The following new features have been implemented in OS/2 Version 2.0:

- Support for the Intel 80386 32-bit microprocessor instruction set; the 80386 was previously supported only in 80286 emulation mode.
- 32-bit memory management.
- Enhanced hardware exploitation.
- Increased maximum file and disk partition sizes.
- Multiple Virtual DOS Machines.
- Support for Windows** applications.
- New graphical installation program, which includes the ability to install from a local area network (LAN) server.
- New, portable 32-bit programming environment.
- Binary compatibility with previous versions of OS/2, allowing 16-bit applications written for previous versions to execute under Version 2.0 without modification.
- An enhanced Presentation Manager* user shell, known as the **Workplace Shell***, which implements the 1991 IBM Systems Application Architecture* (SAA*) Common User Access (CUA) Workplace Environment.
- Implementation of SOM - System Object Model.

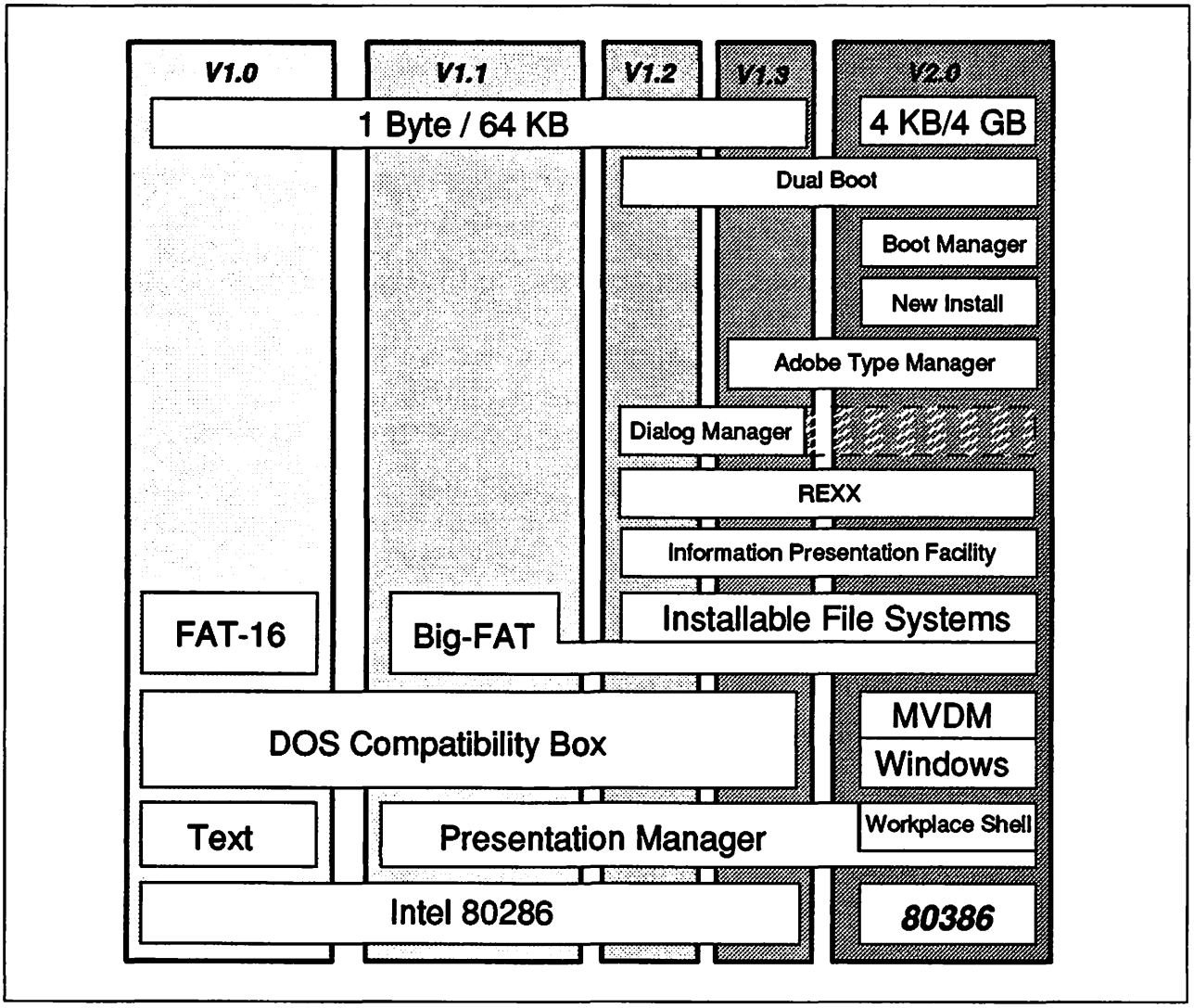


Figure 2. Evolution of OS/2

Many of these new features are described in this document. Others are described in the remaining volumes in the series; see "Related Publications" on page xxi.

1.1 Intel 80386 32-Bit Microprocessor Support

The basis for OS/2 Version 2.0 is its support for the Intel 80386 microprocessor. This support means that a powerful set of 32-bit features now becomes available to the operating system and applications, including enhanced memory management and more sophisticated multitasking. The Intel 80386 and 80486** offer significant improvements over the previous generation of 16-bit microprocessors, while retaining compatibility with these processors.

The Intel 80386 architecture incorporates the following on-chip features, which enhance the throughput and performance of the system:

- Multitasking support
- Memory management
- Instruction pipelining

- Address translation cache
- High-speed bus interface.

Along with these features also comes the greater addressing capacity of the Intel 80386:

- 4 gigabyte (GB) physical address space, with up to 512 megabytes (MB) per process under OS/2 Version 2.0.

  Note: This figure applies to the 80386DX processor and all 80486 processors; the 80386SX, 80386SL and 80386SLC may only address up to 16MB of physical memory.

- 1 byte to 4 gigabyte memory objects.

OS/2 Version 2.0 uses many of these processor features and capabilities to provide a more powerful and flexible operating system platform. Note that OS/2 Version 2.0 does not implement the full 64 terabyte virtual address space provided by the 80386, since this requires use of the segmented memory model; OS/2 Version 2.0 uses a flat memory model which is described below.

1.2 Memory Management

Memory management is the way in which the operating system allows applications to access the system's memory. This includes the way in which memory is allocated, either to a single application or to be shared by multiple applications. The operating system must check how much memory is available to an application, and handle the situation where there is no longer any real memory left to satisfy an application's requests.

In OS/2 Version 2.0, memory management has been enhanced to provide a **flat memory model**, which takes advantage of the 32-bit addressing scheme provided by the Intel 80386 architecture. This means that through memory management, the system's memory is seen as one large linear address space of 4GB. Applications have access to memory by requesting the allocation of **memory objects**. Under OS/2 Version 2.0, these memory objects can be of any size between 1 byte and 512MB. The use of a flat memory model removes the need for application developers to directly manipulate segments, and thereby removes a significant obstacle in porting applications between OS/2 Version 2.0 and other 32-bit environments such as AIX*. Application performance is also improved when manipulating memory, since the use of a linear address space eliminates pointer arithmetic and segment register loads.

OS/2 Version 2.0 manages memory internally using **pages**, each of which is 4KB in size. Each memory object is regarded by the operating system as a set of one or more pages. For practical purposes therefore, memory is allocated in units of 4KB, although a page may be broken down into smaller parts and may contain multiple memory objects.

One of the useful aspects of paged memory is the way in which memory overcommitment is handled, that is, what happens when there is no more real memory left to load applications. Under OS/2 Version 2.0, individual pages may be swapped to and from disk, rather than entire memory objects. This technique improves swapping performance, particularly when large memory objects exist in the system. The fixed page size also improves swapping performance since the operating system need not be concerned with moving objects in memory to

accommodate the various object sizes, as was the case with previous versions of OS/2.

Memory management under OS/2 Version 2.0 is described in more detail in Chapter 2, "Memory Management."

1.3 Task Management

The management of processes and threads executing in the system is greatly simplified and streamlined under OS/2 Version 2.0. This improvement is due primarily to the fact that support for processes executing in real mode (such as the DOS Compatibility Box in previous versions of OS/2) is no longer required, since the execution of DOS applications is supported using virtual DOS machines which run as protected mode processes under OS/2 Version 2.0 (see section 1.9, "Multiple Virtual DOS Machines" on page 8 for further information).

OS/2 Version 2.0 supports execution of the following types of applications:

- DOS applications, in full-screen mode or in windows on the Presentation Manager desktop.
- Windows applications, running in a full-screen session or in windows on the Presentation Manager desktop.
- 16-bit OS/2 applications developed for previous versions of OS/2.
- New 32-bit applications developed for OS/2 Version 2.0.

All applications execute as protected mode processes under OS/2 Version 2.0, and are therefore provided with pre-emptive multitasking and full memory protection between processes.

Interrupt handling under OS/2 Version 2.0 is simplified by removal of the need to handle real mode software interrupts. Interrupts issued by DOS and Windows applications are trapped and translated to the appropriate device access commands for the protected mode environment.

Signal handling has been combined with exception handling, giving a more elegant and flexible means for an application to handle events such as Ctrl+Break and **DosKillProcess()** calls. Applications may register exception handling routines to trap such events. These routines are registered on a per-thread basis, allowing multiple applications to trap the same exception type, and may be chained or nested. Exception handlers may be written in high-level languages such as C; the use of assembler language is not required.

The number of threads per process has been raised under OS/2 Version 2.0 to 4096, which is equal to the total system thread limit. Thus, a process may consume as many threads as required, up to the total available after the operating system itself has consumed a number of threads for its own use.

The process of dispatching secondary threads has been streamlined through the incorporation of stack allocation/deallocation into the **DosCreateThread()** function under Version 2.0. This improvement simplifies the process of thread dispatching by removing the need for an application to explicitly allocate and free memory for the secondary thread's stack.

Semaphores for maintaining synchronization between threads and processes are more sophisticated under OS/2 Version 2.0, and their implementation is less

dependent on the processor architecture. In addition, system functions are provided which enable a thread to wait for the completion of another thread, or to forcibly terminate another thread. This functionality provides additional flexibility compared to previous versions of OS/2.

1.4 32-Bit Programming Environment

OS/2 Version 2.0 provides a 32-bit programming interface, providing enhanced performance through use of the 80386 instruction set, and allowing applications to take full advantage of the 32-bit flat memory model. Since the application developer is no longer concerned with the details of manipulating segments, this simplifies the task of memory management within an application, particularly where large memory objects are used. The lack of a segmented memory model also facilitates porting of applications between OS/2 Version 2.0 and other 32-bit environments.

The 32-bit environment also provides performance improvements since applications have access to 32-bit processor instructions and 32-bit arithmetic. The flat memory model eliminates pointer arithmetic and segment register reloading, which further improves overall performance.

The lack of segmentation also makes program compilation much simpler, since the compiler (and hence the programmer) need no longer be concerned with calling sequences and *far* versus *near* memory references. Only when creating mixed model programs (see 1.5, "16-Bit Application Compatibility") does segmentation become an issue.

Operations such as stack allocation for threads are also simplified in the 32-bit environment. Additional capabilities have been built into the operating system functions, removing the need for the application developer to explicitly perform these functions within the application code. This allows easier exploitation of the multithreading capabilities of OS/2 Version 2.0.

OS/2 Version 2.0 also provides application-transparent emulation of numeric coprocessor functions. If a coprocessor instruction is issued by an application in a machine with no coprocessor installed, the operating system traps the resulting exception condition and emulates the coprocessor instruction, returning the result to the application. Emulation is performed on a per-thread basis, thereby preventing interference between instructions from different threads or processes. All applications developed for Version 2.0 may therefore be compiled with in-line coprocessor instructions, removing the need for conditional testing or coprocessor emulation within the application itself.

Programming for the 32-bit environment is described in more detail in *OS/2 Version 2.0 - Volume 4: Application Development*.

1.5 16-Bit Application Compatibility

An important feature of OS/2 Version 2.0 is its ability to run existing 16-bit applications, written for previous versions of OS/2, without alteration. This compatibility is achieved internally by the operating system through a special interface called a **thunk**. A thunk is a piece of code responsible for converting 16-bit segmented memory references to 32-bit linear references and vice versa, as well as realigning the thread's stack and data structures where necessary. This restruc-

turing allows the 16-bit applications to interface with 32-bit service layers within the operating system, and allows 32-bit applications to utilize 16-bit service layers, modules and resources.

Applications may also be written for the 32-bit environment, while making use of existing 16-bit DLLs and resources. This technique is known as **mixed model programming**, and allows the application developer to make the best possible use of existing service routines, private window classes, etc., developed for previous versions of OS/2.

Note, however, that the use of mixed 16-bit and 32-bit modules within an application requires special consideration by the application developer, since the 16-bit modules utilize the segmented memory model, and therefore are restricted in the size and location of the memory objects to which they have access. The subject of mixed-model programming is discussed in greater detail in *OS/2 Version 2.0 - Volume 4: Application Development*.

1.6 Installation

The installation process for OS/2 Version 2.0 has been enhanced over that provided for previous versions of OS/2. After partitioning the fixed disk (if required) and loading the base operating system, the remainder of the installation is carried out using a graphical installation procedure based on Presentation Manager, with full mouse and keyboard support. Indicators are provided to allow the user to determine the progress of the installation.

Installation of optional features is carried out by selecting the required options from a graphical menu displayed by the installation procedure. Default system settings such as display type, mouse type, etc. are set by interrogating the hardware; these defaults may be modified by the user during installation.

Many parameters such as disk cache size, maximum number of threads and so on, may be set from within the installation procedure. These parameters are set using a dialog box which is displayed when the user selects the *Software Configuration* option from the *OS/2 Setup and Installation* panel.

The installation procedure for OS/2 Version 2.0 is described in detail in Chapter 5, "Installation Considerations."

OS/2 Version 2.0 may also be installed over a local area network (LAN) from a server machine. When installing in this manner, optional features may be specified either by the user or in a response file tailored by a network administrator. The process of installation over a LAN is described in *OS/2 Version 2.0 Remote Installation and Maintenance*.

1.7 Hardware Exploitation

OS/2 Version 2.0 will operate on IBM and other manufacturer's personal computers which are based on either the Intel 80386 or Intel 80486 (both SX and DX) microprocessors. Included in this group are systems based on the 80386SL (the low power version of the 80386SX) and the 80386SLC (the IBM version of the 80386SL with a 8KB cache included on the chip for improved performance). Models in the IBM PS/2 range, which meet the microprocessor requirement, will run OS/2 V2.0.

In order to run OS/2 V2.0, systems should be configured with a minimum of 4MB of memory. A hard disk of at least 60MB is also recommended.

IBM has established a large compatibility test facility in order to verify that certain key functions of OS/2 Version 2.0 perform correctly on hardware manufactured by companies other than IBM. The list of systems, which have passed these tests, is continually being updated and your PC dealer or IBM representative should be able to provide information on the OS/2 V2.0 compatibility of a particular system.

Where a system unit is equipped with more than 16MB of real memory, OS/2 V2.0 will use all the available memory, providing the system unit is able to support memory above 16MB. Before installing additional memory in a system, it is advisable to check that the system unit is able to address memory above 16MB.

Cache management for disk access in OS/2 V2.0 is more intelligent than previous versions of OS/2. In particular the file system will attempt to prefetch records when doing sequential file access.

OS/2 Version 2.0 exploits the capabilities of the IBM SCSI Busmaster adapters. Requests to the adapter can be chained together and a single command can initiate multiple reads from and writes to the disk subsystem. Devices other than disks attached to the adapter are also supported. OS/2 V2.0 also provides support for some non-IBM SCSI adapters. Details will be found in the "*Readme*" file which is installed with OS/2 V2.0.

The use and exploitation of hardware facilities by OS/2 Version 2.0 is described in more detail in Chapter 6, "Hardware Considerations" on page 67.

1.8 Boot Manager

OS/2 Version 2.0 provides a tool that can enable execution of multiple operating systems on the same physical machine, with selection of an operating system by the user at boottime. This capability is provided by an OS/2 Version 2.0 feature known as **Boot Manager**.

Using Boot Manager, the user may select from various operating systems such as DOS, OS/2 Version 1.x, OS/2 Version 2.0 or AIX. Other operating systems may also be supported, provided they do not directly modify the master boot record of the fixed disk. With this feature, OS/2 V2.0 is now capable of installing to and initializing from a logical disk, other than C:. For further details on Boot Manager and its implementation, see Chapter 7, "Boot Manager."

1.9 Multiple Virtual DOS Machines

A significant part of OS/2 Version 2.0 is its ability to multitask DOS sessions along with OS/2 sessions, using the **Multiple Virtual DOS Machines** feature of OS/2 Version 2.0. In previous versions of OS/2, support for DOS applications was limited, with low available memory and a single DOS session, which operated in full-screen mode only and suspended when in the background.

The DOS support has been totally rewritten in OS/2 Version 2.0, and allows *multiple* concurrent DOS applications where each is executed as a singie-threaded, protect mode OS/2 program. This method of implementation uses the **virtual**

8086 mode of the 80386 processor, and provides normal OS/2 levels of memory protection; that is, it provides protection of system memory and other applications, isolation from illegal memory accesses by ill-behaved applications, and the ability to terminate sessions where applications are "hung". DOS sessions may also be multitasked along with all other OS/2 sessions.

DOS support is achieved through the use of virtualization techniques, allowing the creation of multiple instances of separate, independent **virtual DOS machines**. In this way, a virtual interface is provided to each DOS machine, giving the impression that each application owns all the required resources, both hardware and software.

Each virtual DOS machine has more base memory than the DOS Compatibility Box implemented in previous versions of OS/2; more than 630KB of free base memory (that is, memory below the 640KB line) is available for each virtual DOS machine. OS/2 Version 2.0 also supports the use of Lotus**-Intel-Microsoft** (LIM) expanded memory (EMS) Version 4.0 emulation and extended memory (XMS) Version 2.0 to provide additional memory for those DOS applications which are capable of using such extensions. OS/2 Version 2.0 maps this extended or expanded memory into the system's normal linear memory address space, and manages it in the same manner as any other allocated memory. The DOS Protect Mode Interface (DPMI) specification is also supported to allow access to memory above 1MB.

The ability of a virtual DOS machine to run within a Presentation Manager window provides immediate productivity gains to existing DOS applications, since they may utilize Presentation Manager desktop features. These features include window manipulation and the ability to cut/copy/paste information between applications using the clipboard.

Application compatibility in the virtual DOS machine is also enhanced over previous versions of OS/2. The virtual DOS machine can be used to execute DOS-based communications applications and other applications which address hardware I/O devices, through the use of **virtual device drivers** which map the device driver calls from the applications to the appropriate physical device driver within the operating system. Applications using hardware devices which are not required to be shared with other applications in the same system may be accessed using the standard DOS device drivers, without the need for a virtual device driver. Certain restrictions still apply with respect to communications line speed and time-critical interrupt handling.

Application compatibility in a virtual DOS machine is further enhanced by the **DOS Settings** feature, which allows virtual DOS machines to be customized to suit the requirements of the applications running in them. Properties such as video characteristics, hardware environment emulation, and the use of memory extenders can all be customized using this feature.

The concept of Multiple Virtual DOS Machines, and its implementation in OS/2 Version 2.0, is described in more detail in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

1.10 Windows Application Support

OS/2 Version 2.0 provides the capability for Windows applications to run under OS/2 Version 2.0 using its WIN-OS/2* component. This support allows applications written for Windows 3.0 and previous versions of Windows to coexist and execute in the same machine.

Each Windows application executes as a protected mode process. Windows applications are, therefore, subject to the full application protection facilities provided to protected mode applications under OS/2 Version 2.0, and are protected from one another and from DOS or OS/2 applications executing in the system. This protection is in contrast to the native Windows 3.0 environment, where protection is limited to Windows 3.0 applications only, provided these applications use Windows' memory management services. Protection for DOS applications is provided only when Windows is running in 386 enhanced mode.

The execution of Windows applications as protected mode tasks also allows these applications to take full advantage of the pre-emptive multitasking capabilities of OS/2 Version 2.0, with full pre-emptive multitasking between Windows applications, DOS applications, and OS/2 applications. This protection is again in contrast to the native Windows 3.0 environment, where pre-emptive multitasking is available only when Windows 3.0 is running in 386 enhanced mode, and only for DOS applications, thereby impacting performance and preventing many applications written for previous versions of Windows from executing. OS/2 Version 2.0 has no such restriction.

Support for Windows applications under OS/2 Version 2.0 is discussed in more detail in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

1.11 Workplace Shell

The component of the operating system which is responsible for interaction with the end user is known as the user shell. OS/2 Version 2.0 provides an enhanced, object-based user shell known as the **Workplace Shell**, which implements the 1991 SAA CUA Workplace Environment.

In the Workplace Shell, system resources such as files, printers, etc. are regarded as objects, and represented by graphical icons on the screen. Users may manipulate objects (open them for editing, copy them, print them, etc.) by direct manipulation of the icons. For example, a file is copied from one location to another by pointing to it with the mouse, *dragging* the object's icon to the required destination, and *dropping* the icon by releasing the mouse button. This action is known as **drag and drop** manipulation.

The Workplace Shell allows users to become more task-oriented by simplifying the user interface and reducing the amount of system-specific knowledge required to perform work tasks. The performance of these tasks using the Workplace Shell is more analogous to the way in which such tasks would be performed manually, thereby requiring less user education on the operation of the system.

The Workplace Shell is described in detail in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*.

1.12 Summary

OS/2 Version 2.0 provides significant enhancements over previous versions of OS/2. It provides a sophisticated memory management and task management architecture, allowing full exploitation of the power of the Intel 80386 processor.

OS/2 Version 2.0 provides the ability to execute multiple concurrent DOS applications, in full-screen mode or in windows on the Presentation Manager desktop. These applications may address I/O devices such as printers, scanners, and communications adapters. Each DOS application typically has approximately 630KB of memory in which to execute and store its data; for those applications which require more memory, OS/2 Version 2.0 provides emulation of the LIM, EMS, and XMS memory extenders.

OS/2 Version 2.0 provides the ability for Windows applications, written for Windows 3.0 and previous versions of Windows, to execute under OS/2 Version 2.0, concurrently with DOS and OS/2 applications. Windows applications execute as protected mode processes, and full memory protection and pre-emptive multitasking are, therefore, provided for these applications.

OS/2 Version 2.0 provides a more powerful 32-bit programming environment which, due to the use of the flat memory model, is free from the limitations and inherent complexity of the segmented memory model used by DOS and previous versions of OS/2. Memory management within applications is greatly simplified, allowing applications to be developed faster, with better performance due to reduced memory manipulation overheads. Through the use of the flat memory model, applications may be more easily ported to or from other operating system platforms.

OS/2 Version 2.0 also provides an enhanced user shell, known as the Workplace Shell, through enhancements to Presentation Manager. The Workplace Shell is object-based and implements the 1991 SAA CUA workplace environment. This shell is more intuitive than the Presentation Manager shell implemented in previous versions of OS/2, and allows users to become familiar with the system more quickly.

Chapter 2. Memory Management

OS/2 Version 2.0 is based on the Intel 80386 microprocessor architecture, and exploits the 32-bit features of the 80386 processor. The features used by OS/2 Version 2.0 are:

- 32-bit register set
- 32-bit instructions/addressing
- Large memory objects (greater than 64KB)
- Paging.

Version 2.0 introduces a flat memory model with a linear address space of 4 gigabytes (GB), and removes many of the memory management restrictions experienced in previous versions of OS/2. The purpose of this chapter is to provide the reader with an understanding of the way that OS/2 Version 2.0 manages memory and the impact of this on system administrators and application developers.

2.1 Introduction

Memory management is the way in which an operating system allows applications to access memory, either for private use by a single application or to be shared between applications. In either case, it is the responsibility of the operating system's memory management component to supervise the correct use of memory and to ensure that no application gains access to memory outside its own address space.

Previous versions of OS/2 were based upon the Intel 80286 processor architecture. In this architecture, there is a limitation on the amount of memory that can be addressed as a single unit. This is due to the fact that memory is managed in segments of up to 64KB in size. Previous versions of OS/2 maintained a series of descriptor tables for memory segments, and 16 bits were allocated in each table entry for the length of the segment. Thus, each descriptor table entry gave access to a segment of up to 2^{16} = 64KB in size.

The particular implementation of the segmented memory model within the 80286 processor allowed a minimum segment size of 1 byte. Hence an application could request the allocation of a memory segment from 1 byte to 64KB in size, in a single operation. These segments formed the basis of memory allocation within the system, and of the virtual memory implementation by which memory overcommitment was supported.

If the need arose for more than 64KB to be used for a single memory object or data structure, the programmer and the operating system had to take this limitation into consideration, and implement appropriate algorithms to use multiple memory segments for a single logical structure.

OS/2 Version 2.0, however, is based upon the Intel 80386 processor architecture. This processor has a 32-bit addressing scheme in place of the 24-bit overlapped scheme used in the 80286, thereby giving access to 2^{32} = 4GB of memory in a single logical unit.

However, if a unit of this size were to be allocated and manipulated in the same segment-oriented manner as implemented in the 80286, severe problems would arise. For instance, the segment could potentially be larger than the available memory in the system. An alternative mechanism for memory manipulation is therefore required with the 80386.

In the 80386 architecture, memory is split into fixed size units of 4KB. All memory allocation, addressing, swapping and protection is based on pages. As with previous versions of OS/2 the total memory, allocated to all processes running in the system, may exceed the physical memory available. Memory objects or parts of memory objects, which are not required by the currently executing process, may be temporarily migrated out to secondary storage (disk). When used with a paged memory management scheme, this procedure is known as *paging*. An application may request a large amount of memory, in which case multiple pages are allocated. However, since virtual memory is managed on a page-by-page basis, such units of storage may now exist partly in real memory and partly in a file on disk, thereby significantly easing the constraints on memory overcommitment.

OS/2 Version 1.3 moved complete segments between main memory and the swap file. The fact that segments were variable in length complicated the management of both main memory and space in the swap file. There was also the requirement to compress memory regularly to reclaim the gaps, which formed when memory was freed. Under OS/2 V2.0, in most cases there is no requirement to find contiguous pages in memory to satisfy an allocation request. Consequently there is no need to move pages around in memory. The exception to this is the need for buffers used in DMA I/O transfer, which must be in contiguous locations in memory.

The remainder of this chapter will explore the memory management capabilities of the 80386, as implemented by OS/2 Version 2.0, in more detail.

2.2 Flat Memory Model

The memory model used by OS/2 Version 2.0 is known as a **flat memory model**. This term refers to the fact that memory is regarded as a single large linear address space of 4GB, using 32 bits for direct memory addressing. This view applies for each and every process in OS/2 Version 2.0. Memory addresses are defined using a 32-bit addressing scheme, which results in a linear address space of 4GB in size.

Like the 80286 processor, the 80386 also supports a segmented memory model, except that in the case of the 80386 the maximum segment size is 4GB. While the 80386 processor does not explicitly provide a facility for disabling the segmented memory model, OS/2 Version 2.0 implements the flat memory model by mapping the 4GB address space as a single code segment and a single data segment, each of which has a base address of zero and a size of 4GB. Only two segment selectors are therefore required in the system; an executable/readable code segment in the CS register, and a read/write data segment in the DS, ES, FS, GS, and SS registers. These selectors are known as *aliases*, since they all map to the same linear address range. The way, in which the address space is implemented using the Intel 80386 processor, is explained in Appendix A, "Intel 80386 Architecture."

The 32-bit addressing scheme used by OS/2 Version 2.0 will hereafter be referred to as *0:32*, in order to differentiate it from the 16-bit segmented addressing scheme used by previous versions of OS/2, which will be referred to as *16:16*. These terms reflect the fact that the older segmented memory model uses a 16-bit segment selector and a 16-bit offset to refer to a specific memory location, whereas the newer flat memory model has no need of a segment selector, and simply uses a 32-bit offset within the system's linear address space.

The system's global address space is the entire 4GB linear address space. Each process has its own **process address space**, completely distinct from that of all other processes in the system. All threads within a process share the same process address space. This address space is theoretically also 4GB in size. However, the maximum size for process address spaces is defined at system initialization time and is somewhat less than 4GB, to allow space for memory used by the operating system.

Figure 3 shows the mapping of a process address space into the system's global address space. The *NN* shown in Figure 3 represents the maximum defined linear address of the process address space, set at system initialization time. OS/2 Version 2.0 sets this limit to 512MB, reserving the linear address range above this point for operating system use. The space above 512MB is known as the **system region**.

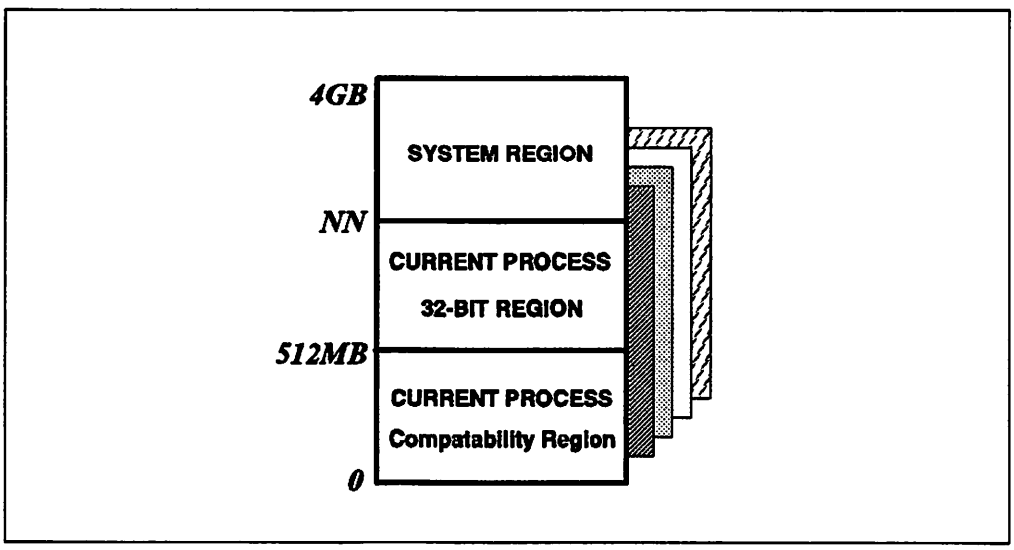


Figure 3. 4GB Global Linear Address Space. Note that the 32-bit region within the process address space is not used by OS/2 Version 2.0, since "NN" is set to 512MB.

This limitation on the size of the process address space is used by the operating system to ensure protection of the system region from access by applications. See 2.3.5, "Memory Protection" on page 21 for further explanation.

Conceptually, the process address space is divided into two different regions, as shown in Figure 3. One of these regions may be accessed by both 16:16 and 0:32 applications, and is known as the **16/32-bit region** or **compatibility region**. The other region is accessible by 0:32 applications only, and is known as the **32-bit region**.

The 16:16 addressing scheme allows access of up to 512MB per process, since the local descriptor tables used in this model contain up to 8192 entries, each of which can point to a segment of up to 64KB in size. In order to ensure that there

is no problem in coexisting 16-bit and 32-bit applications under OS/2 V2.0, the maximum size of the process address space has been set at 512MB. This means that all memory in the process address space can be addressed using either the 16:16 or the 0:32 addressing scheme. This capability is important since it allows applications to be composed of mixed 16-bit and 32-bit code, allows 32-bit applications to make function calls to 16-bit service layers, and permits 16-bit applications written for OS/2 Version 1.x to run unmodified, effectively allowing a "hybrid" memory management environment.

The 32-bit flat memory model greatly simplifies the migration of 32-bit applications to OS/2 Version 2.0 from other operating system platforms, and the migration of OS/2 Version 2.0 applications to other platforms. This is in contrast to the segmented memory model implemented by the 80286 processor. All the features described above arise from the fact that a flat linear memory model is used, taking advantage of the advanced features of the 80386 processor. The paging scheme is more general than the segmentation scheme used by the 80286, and the flat memory model will facilitate any future migration of OS/2 to a hardware platform other than the Intel 80x86 family.

2.3 Memory Objects

A memory object is the term used under OS/2 Version 2.0 for a linear, contiguous range of memory addresses, which is regarded and manipulated as a single logical unit by an application. A memory object is actually composed of one or more discrete 4KB pages, and is viewed as such by the operating system. With minor exceptions however, an application need not be aware of the paged nature of a memory object, since OS/2 Version 2.0 handles all paging internally. See 2.4, "Physical Memory Management" on page 22 for further discussion of paging.

Each process under OS/2 Version 2.0 uses memory objects, and all memory allocation and sharing from an application viewpoint is based on memory objects. The reader may conceptually regard memory objects as similar to segments, but the means of addressing memory objects is greatly simplified, since there is no need to construct the address using a segment selector.

2.3.1 Allocation and Management

OS/2 Version 2.0 allows a memory object to have any size between 1 byte and 512MB, which is the maximum amount of memory addressable in a process address space. A program uses the **DosAllocMem()** and **DosAllocSharedMem()** function calls to allocate memory objects. The use of these and other available functions to manipulate memory objects is described in *OS/2 Version 2.0 - Volume 4: Application Development.*

Allocating a memory object with a size of 1 byte will in reality reserve a full 4KB page, since the operating system allocates memory on a page-by-page basis. In order to avoid large amounts of wasted memory, an application, which uses many small memory objects, should request the allocation of a storage pool from the operating system, and then suballocate this storage as required. This technique reduces memory fragmentation and allows more efficient use of memory resources. It is described in detail in *OS/2 Version 2.0 - Volume 4: Application Development.*

When a memory object is allocated, its base address and maximum size are defined. The location of the object and the size of the object is fixed for the lifetime of the object. It can be neither re-sized or moved within the virtual address space. By default, however, no physical storage is reserved for a memory object at allocation time; the operating system merely reserves a range of addresses in the process address space for that object. A memory object which is allocated in this way is known as a **sparse object**.

Before an application can write to a memory object, the object must be *committed*; upon commitment, physical storage is reserved for the memory object. Storage can be committed at either of two points:

- The memory object may be committed in its entirety at the time the object is allocated. This method is typically used for small memory objects, the size of which is fixed and predetermined prior to execution.
- The memory object or any part of it may be committed after allocation, in 4KB (page) units. This method is typically used for memory objects such as documents or spreadsheets, which are likely to increase in size during execution.

This effectively allows an application to increase the size of a memory object in a series of steps, as the storage requirements for that object increase during application execution. A memory object can therefore be allocated at its maximum possible size during initialization, without imposing large memory overheads on the system as a whole, since real storage is reserved *only* as it is required.

Each page within the memory object can be individually committed, or a group of pages may be committed at the same time, up to the maximum size of the memory object stipulated during allocation. Note that this is one of the few instances where an application developer must be aware of the paged memory architecture.

2.3.2 Guard Page Technique

When the amount of real storage reserved for a memory object is increased dynamically by progressive commitment of pages, the application is *not* required to explicitly determine whether the next write operation will exceed the limit of the storage already reserved. When such a write operation occurs, the operating system may notify the application, by raising a **guard page fault** exception.

This technique is useful for situations where storage requirements grow linearly. OS/2 V2.0 implements automatic stack growth by using guard pages. When wishing to limit physical storage requirements but at the same time allow for situations where large data areas might be needed, an application should consider using guard pages. A memory object is allocated with the largest possible size that might be required. The application then commits the minimum number of pages, that are required. Usually this would start at the lowest address in the memory object and proceed upwards. The application then marks the next highest page as a **guard page**. The guard page is also a committed page.

The application must also register an exception handler. Exception handlers are described in 3.3, "Signal and Exception Handling" on page 38. When the application tries to write data into the guard page, a guard page fault will be raised and control passed to the application's exception handler. The application must then unguard the current guard page, commit the next highest page and then set

guard on that page. Should the application try to access uncommitted storage above the guard page, a general protection exception will occur.

OS/2 Version 2.0 provides a default guard page exception handler, which commits the next *lower* page in the object; this is done because the default handler is written to handle dynamic stack growth. Stacks are always propagated downward. For all threads other than the first in any process, OS/2 V2.0 allocates the stack as a sparse object. The page with the highest address is committed and the page immediately below it is marked as a guard page. No other pages are committed. When the guard page is accessed, the default handler tries to get another guard page below the current one. If successful, the original guard page becomes a normal stack page. An application could allow the default handler to process guard page exceptions on its private memory objects; however, in most cases the actions taken by this default handler will not be appropriate and an application should register its own exception handler.

Use of the guard page technique is strongly recommended whenever the amount of data to be written into a memory object is variable, or when the size of a memory object may increase during execution. The process of creating and registering an exception handler and of using guard pages and the handling of guard page exceptions within applications is described in *OS/2 Version 2.0 - Volume 4: Application Development*.

2.3.3 Virtual Memory Management

The virtual address space is split into two regions:

- **System Region**

  This is the region above the 512MB, which is only accessible to tasks running at operating system privilege level.

- **Process Region**

  This is the first 512MB of the virtual address and only memory objects in this region are mapped into a process's address space when that process is running at user privilege level. Each process present in the system has its own mapping of this region. The process region is further split into:

  - A shared area

    This is used to hold shared memory objects such as DLL code and shared data areas.

  - A private area

    This contains EXE code and process private data.

In order to manage virtual memory, OS/2 V2.0 uses the concept of an **arena**. There are three arena types in the system:

- The system arena
- The shared arena
- Per-process private arenas.

Associated with each arena is the virtual address space, which it maps. The system arena contains all the memory objects that are in the system region. It maps the virtual address space between 512MB and 4GB. The shared arena describes all the shared memory objects in the process region.

Each process has its own private arena, which contains EXE code and the process's private data. The private arena starts at the lowest address of the process region's virtual address space and has a minimum size of 64MB. A program loaded into the address space will be loaded at the low end of the address space. Because of this, a particular EXE will always occupy the same range of addresses. If a program is used by more than one process it is possible to share one copy of the program code.

The shared arena is allocated starting at the top end of the process region and moves down towards the private arena. It has a minimum size of 64MB. The upper limit of the private arena and the lower limit of the shared move towards one another. Each object in the shared arena is allocated its own linear address range. It will have the same address range in each of the process address spaces, into which it is mapped.

Each process has its own address space, which maps memory objects in the process's private arena and the shared arena. Only those objects in the shared arena, which a process requires access to and is authorized to access, will be mapped into the process's address space.

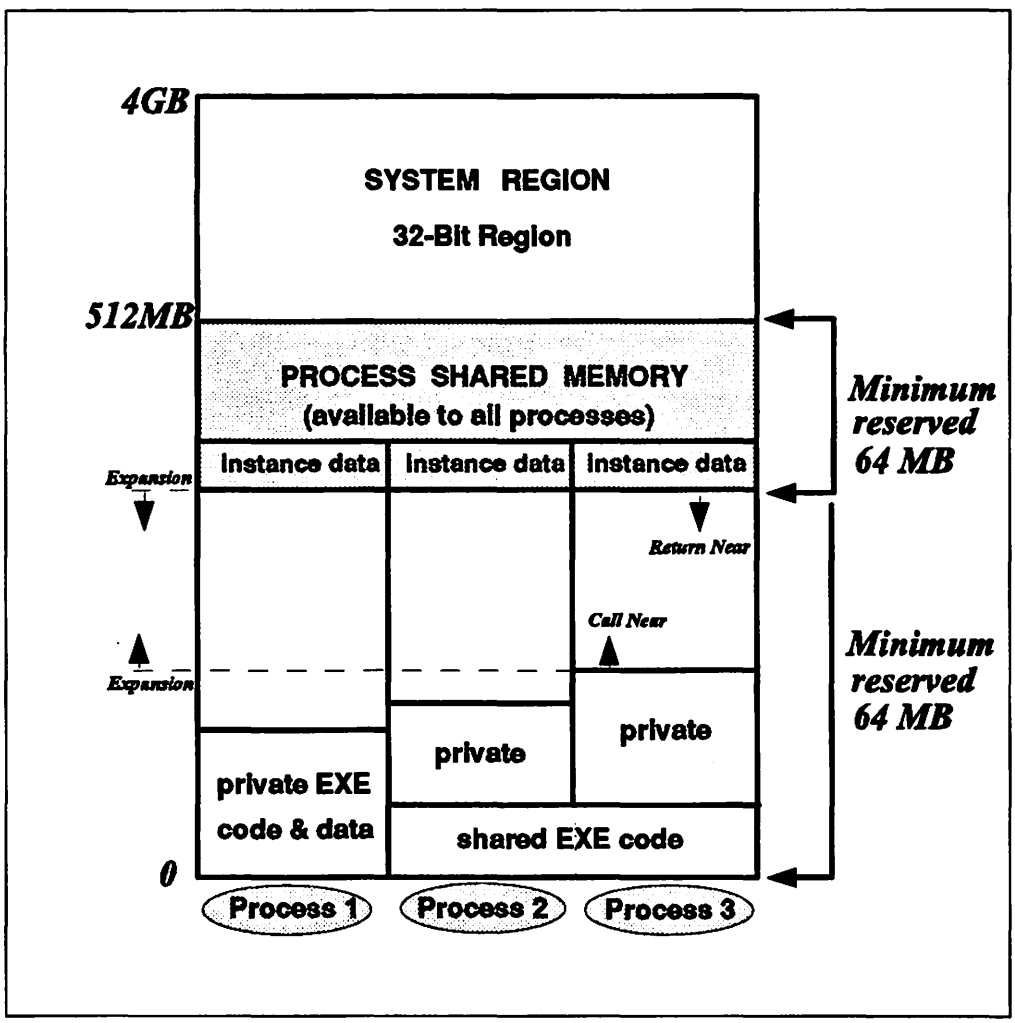


Figure 4. Process Address Space Layout

Both private and shared storage for memory objects may be allocated within each arena. For example, DLL instance data is located within the shared memory arena, but each instance of the data uses a separate memory object in

order to preserve data integrity, and hence each memory object is treated as private storage. Although separate memory objects, they each map to the same range of addresses in the shared arena. Table 1 on page 20 shows the types of storage (private or shared) available within each memory arena, and the uses to which these types of storage may be put by applications.

Table 1. Memory Object Classes. This table shows the way in which memory objects may be placed in shared or private storage.

Arena	Private Storage	Shared Storage
Private	EXE read/write data Process run-time data	Shared run-time data Shared DLL data
Shared	DLL instance data	DLL code and global data
Note: Code includes read-only objects such as Presentation Manager resources		

Private storage in the private arena is used for read/write data, allocated at either loadtime or runtime, which is accessed only by a single process. Shared storage in the private arena is used for executable code, which may be shared between processes.

Shared storage in the shared arena is used for DLL code and read-only data, as well as DLL read/write data, which is not instance-specific. Such objects may be accessed by all processes in the system. Private storage in the shared arena is used for DLL instance data, which is unique to each process accessing the DLL.

For a more detailed discussion of the process of virtual memory management, readers should refer to *The Design of OS/2*.

2.3.4 Page Attributes

An application may specify the types of access permitted for memory objects when those objects are allocated, thereby ensuring the proper use of each memory object. The type of access for individual pages within the memory object may be altered subsequent to allocation. The attributes available for memory objects and their component pages are:

- **Commit**

  The pages within a memory object must be committed in order to be used for read or write operations. Until it is committed, the system merely reserves a linear address range without reserving physical storage. The committing of a page obtains a page frame for the page; see 2.4.1, "Address Translation" on page 23 for further explanation.

- **Read**

  Read access to the page is allowed. All other access attempts will result in a page fault.

- **Write**

  Write access to the page is allowed. Write access implies both read and execute access.

- **Execute**

  Execute access to the page is allowed. Execute access implies read access.

- **Guard**

When an application attempts to write into the guard page, a guard page fault exception is generated for the thread that referenced the guard page. This exception can be handled by an application-registered exception handler for this thread. This process is described in 2.3.2, "Guard Page Technique" on page 17.

- **Tile**

  Defining a memory object to be tiled causes it to be placed in the compatibility region and mapped using the 16:16 addressing scheme, even though the object may be used by a 0:32 process.

  Specifying this attribute has no effect under OS/2 Version 2.0 as all application-created storage must reside below the 512MB address limit, and is therefore within the compatibility region. This attribute is provided to allow applications to be developed that will be forwardly compatible with future versions of the operating system. It is likely that in a future release of OS/2 the 32-bit region (above 512MB) will be enabled for application use. This attribute must then be specified for a memory object that will be used by 16-bit code.

The 80386 processor does not distinguish between read and execute access. The one implies the other. Write access implies both read and execute access.

At allocation, all pages within the object will be given the attributes specified on the **DosAllocMem** or **DosAllocSharedMem**. After allocation, attributes of individual pages or groups of pages within the memory object may be changed using the **DosSetMem()** function. This function is described in *OS/2 Version 2.0 - Volume 4: Application Development*.

2.3.5 Memory Protection

With the flat memory model, OS/2 V2.0 implements memory protection using two machine states (user and supervisor) and by providing separate address spaces for the supervisor and each of the processes running in the system. The global address space encompasses the entire linear address space and consists of the system region and the current process's address space. The global address space is only accessible when the processor is running at ring 0, which is reserved for the operating system. All other processes run in ring 3 (privilege level 3). The ring protection architecture of the 80386 processor is described in A.5.3, "Privilege Levels" on page 120. While executing at ring 3, the system region is not visible to the current process. Neither are the addresses spaces of any of the other processes running in the system accessible.

Since memory is managed by the operating system on a page-by-page basis, even the allocation of a 1 byte memory object will actually reserve a full page (4KB) in memory. Furthermore, as the memory protection scheme has also changed under OS/2 Version 2.0, a memory reference outside the expected range but within the 4KB page boundary will *not* give the Trap 000D segmentation violation experienced in previous versions of OS/2. Instead, an exception is generated only when an invalid page is referenced or an invalid access occurs (such as write operation to a page previously declared as read-only). An invalid page is a page that has not been committed in the process address space or is outside the limit of the address space.

Here is where we see a major difference between the *segmented* memory model and the *linear* or *flat* memory model. A 32-bit program can address the entire

4GB address space with a 32-bit offset. Memory is seen as a single continuum. 16-bit applications see memory as discrete areas each with their own defined size. 16-bit applications running under OS/2 V2.0 are subject to segment limit checking and generally behave as they did under previous versions of OS/2. The discussion here refers to 32-bit applications.

For example, an application may request the allocation of 1KB of memory; the operating system will allocate a full 4KB page. The application can then write up to 4096 bytes of data into the memory object, and the operating system will not detect an error. However, if the application attempts to write 4097 bytes into the memory object, a general protection exception (Trap 000D) may occur. Such an exception is *only* generated when the next page in the process address space is invalid. If the next page exists in the process address space, no exception is generated.

Note that any of these exceptions may be trapped and processed using exception handlers registered by the application. See Chapter 3, "Task Management" for further discussion of exception handling.

Since 32-bit programs can address the entire address space with a 32-bit offset, it is easier for 32-bit programs to corrupt data in the shared region than for 16-bit programs. OS/2 V2.0 provides a facility for DLL routines to have their shared data areas allocated in a *protected* area of memory, which is not accessible to 32-bit programs, thereby providing a level of protection. There is a new **PROTECT** option on the **MEMMAN** statement in the CONFIG.SYS, which is used to enable memory protection for DLLs.

2.4 Physical Memory Management

The previous discussion has concentrated on the application's view of memory management. Applications running under OS/2 Version 2.0 need not even be aware of the mechanism through which the 0:32 addressing scheme is implemented; the application *only* deals with 32-bit addresses, and is not concerned with the way these are mapped to physical addresses in processor storage.

When using the flat memory model, the 80386 processor regards each memory address as a 32-bit address, seen solely as an offset from address 0 into a linear address space. If the 80386 processor is running with paging disabled, this linear address and the physical address in memory are equal. Physical memory is now divided 4KB blocks, which is allocated as required to processes running in the system. There is no direct correlation between the linear address of a page and its address in physical storage. In fact, not all pages in the linear address space will be represented in physical memory. The 80386 processor has a paging subsystem, which handles the conversion of the linear address into a physical address and also detects situations where there is no physical page corresponding to a page in the linear address space.

Paging is usually carried out without any awareness on the part of an application.

2.4.1 Address Translation

With paging enabled the 80386 processor maps the linear address using an entry in the page directory and an entry in one of the page tables currently present in the system. The page directory and page tables are structures created in the linear address space. The address translation process is shown in Figure 5. The linear address is split into three parts:

- **Page Directory Entry** or PDE (10 bits)
- **Page Table Entry** or PTE (10 bits)
- **Page Offset** or PO (12 bits).

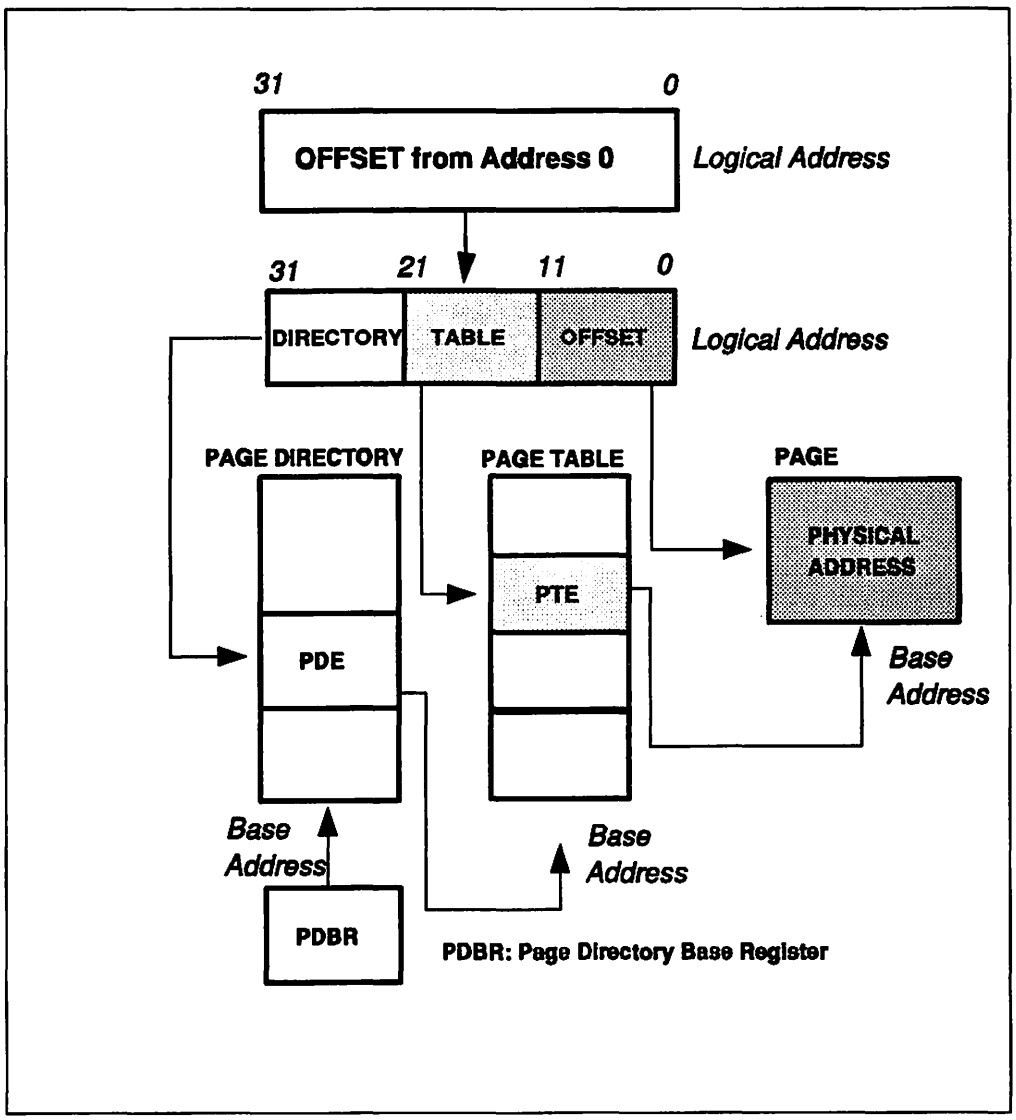


Figure 5. Address Translation - Linear Address to Physical Address

OS/2 Version 2.0 maintains a single page directory for the entire system; the entries contained within the page directory are unique for each process in the system, and are copied into the page directory by the operating system when a task switch occurs. The page directory entries contain pointers to the page tables, which in turn point to the memory pages (both shared and private) belonging to the current process. The page directory and each page table are defined to be one page (4KB) in size, and are aligned on 4KB page boundaries. There is a maximum of 1024 entries per page table, and a maximum of 1024 page tables per page directory. Since each page is 4KB in size, this means a

single page table gives access to 4MB of memory. 1024 page tables, the maximum, gives access to the full 4GB global address space.

The format of page directory entries and page table entries are identical. The upper 20 bits in each page directory entry specify the address of the page table, and the lower 12 bits are used to store control and status information. This 20-bit address is possible since each page table is aligned on a 4KB boundary. Hence, the lower 12 bits of the address are assumed to be zero for addressing purposes; these 12 bits are in fact used to contain control and status information. A page table contains entries pointing to the physical memory location of the page.

The address resolution may appear complex, but in fact very little overhead is involved since the 80386 maintains a cache buffer for page table entries, known as the **translation lookaside buffer** (TLB). The TLB satisfies most access requests for page tables, avoiding the necessity to access system memory for PDEs and PTEs.

When page table entries are changed or during a task switch, the TLB must be flushed in order to remove invalid entries. Otherwise, invalid data might be used for address translation.

For each page frame, a bit in the page table entry known as the *present bit* indicates whether the address in that page table entry maps to a page in physical memory. When the present bit is set, the page is in memory. When the present bit is clear in either the page directory or in the page table, a page fault is generated if an attempt is made to use a page table entry for address translation.

2.4.2 Managing Paging

Pages can have the following types:

Fixed These are pages that are permanently resident in storage. They may not be moved or swapped out to secondary storage.

Swappable When there is a shortage of physical memory, these pages can be swapped to disk.

Discardable It is possible to reload these pages from either an EXE or a DLL file. When memory becomes overcommitted, space used for discardable pages can be freed up, and when the pages are required again they are loaded from the original file.

Invalid These are pages that have been allocated but not committed.

The operating system needs information over and above that contained in the page directories and the page tables to manage the paging process. OS/2 V2.0 builds three arrays of data structures that represent:

1. Committed pages in the process and system address spaces
2. Pages in physical memory
3. Pages held on secondary storage.

The following sections describe these arrays and the way in which OS/2 V2.0 uses them.

2.4.2.1 Virtual Pages

A virtual page structure (VP) is allocated whenever a page is committed in response to an application request. No physical memory is allocated for the page at this time. The PTE is updated to point to the VP but the present bit in the page table is *not* set. When the page is first referenced, a page fault occurs. The allocation of physical memory is left to the last possible moment.

The virtual page structure describes the current disposition of a page. When a page fault occurs, the virtual memory manager obtains the address of the VP from the page table entry, and uses the information held in the VP to determine the required action which must be taken to resolve the page fault. The possible actions are:

- The page manager will provide a page, initialized to zeros if required
- The page will be loaded from an EXE or DLL file
- The page will be loaded from the swap file on secondary storage.

If the page is to be loaded from an EXE or DLL, the VP contains a pointer to the loader block. If from the swap file, it points to a page in the swap file.

2.4.2.2 Page Frames

A page frame (PF) exists for each page of physical memory in the system. Page frames are stored in an array, which is indexed by the physical frame number within the system.

A page frame may have one of three states:

- *Free*, in which case the page frame is available for allocation to a process in the system. The page frame addresses of all the free pages in the system are held in a doubly linked list known as the *free list*, with PFs for fast planar memory at one end of the list, and PFs for the slower memory on adapters at the other end. This permits the allocation of faster memory before the slower memory.
- *In-use*, in which case the page has been allocated to the current process in the system.
- *Idle*, in which case the page frame has been allocated to a process, but no page table entries for the current process reference this frame. This lack of reference normally indicates that the process, which is using the page, has been switched out; that is, the process is *not* the current process.

When the system is overcommitted, the number of free and idle page frames begins to fall. When it reaches a threshold level, pages are migrated from the in-use state to the idle state by a **page ager**. The page ager looks for pages that have not been accessed since the last time the page ager ran. The ager examines the accessed bit in the PTE. If clear, it marks the page not present. If a page is referenced by more than one PTE, a shared page, all PTEs must be marked not present before the page is placed on the idle list. The *idle list* is also a doubly linked list and least recently used (LRU) entries are at one end of the list and most recently used (MRU) entries are at the other end of the list.

Pages are also classified as *dirty* or *clean*. A dirty page is one that has been written to and must have its contents swapped to disk before it can be allocated to another process. A clean page does not need to be swapped out, since it typically contains code or read-only data, which can be reloaded from the ori-

ginal file on disk. It could also be that a copy of the page currently exists in the swap file.

Placing a page frame on the *idle list* does not destroy its contents. The page frame is only reused when the operating system is forced to *steal* a page frame from the *idle list* in order to accommodate the loading of a page after a page fault. The swapping of an idle swappable page to disk is also usually delayed until there is a need to reuse the page frame to satisfy a page fault. To take advantage of the capability of certain I/O adapters to chain operations, other pages on the idle list may be swapped out at the same time.

2.4.2.3 Swap Frames

A swap frame (SF) is similar to a page frame except that an SF refers to a slot in the swap file, which can be used to hold a page when main storage becomes overcommitted. The swap frame array is used to control allocation of space in the swap file. If the **SWAP** option is present on the **MEMMAN** statement in the CONFIG.SYS, the SWAPPER.DAT file will be created in the directory pointed to in the **SWAPPATH** statement. The initial size of the file is 512KB.

The size of the swap file is determined by the amount of memory overcommitment in the system. The algorithm used in this calculation takes into account the amount of storage needed for all the fixed pages and swappable pages in the system and the amount by which this exceeds the physical storage installed in the system. The memory overcommitment is recalculated each time pages are committed. It should not be necessary to increase the size of the swap file each time pages are committed because of the fact that increases to the swap file will always be in steps of 512KB.

However allocation of VPs to SFs does not take place when the page is committed. All the operating system ensures at this time is that there will be space in the swap file for the page when it becomes necessary to swap it out. The SF is allocated to the page when it is first selected from the idle list for swapping. When the page is swapped back in, the SF will not be immediately freed up, but a link to the VP is maintained. If the page is then again selected for swapping out before it has been changed, it will not be necessary to write it to disk as a copy of it still exists in the swap file.

The OS/2 V2.0 swap file can also decrease in size. Decrements to the size of the swap file will be in 512KB blocks. When the overcommitment calculation indicates that the swap file is too large by one or more multiples of 512KB, the area at the end of the swap file is marked for shrinking. No new allocations of SFs in the area marked for shrinking will take place. When all SFs in the shrinkable area are free, the swap file is reduced in size. No attempt is made to force the freeing of SFs in this area; consequently there can be a longish delay in the swap file becoming eligible for shrinking and the shrinking actually taking place.

2.4.3 Processing Page Faults

When a process attempts to access a page, for which the present bit in the PTE is not set, a page fault occurs. The page fault is passed to OS/2 Version 2.0's page fault handler, and the following sequence of events takes place:

1. A PF is allocated from the free list or the LRU end of the idle list. Should the PF be taken from the idle list, and its current contents be marked as "*dirty*", it will be necessary to first write the page to the swap file.

2. Once the PF is available, its contents will be loaded based on information contained in the VP. The source could be one of the following:
   - If the page is marked "*allocate on demand*", the physical memory manager will provide the page. If requested the page is initialized to zeros.
   - If the page is discardable, it is reloaded from an executable file on disk.
   - If the page is swappable but is currently on the idle list, it can be reclaimed because it is still present in memory.
   - If the page is swapped out, it is reloaded from the swap file.
3. The PTE is updated with the PF address and the present bit is set on.
4. The TLB is flushed.
5. The program instruction that caused the exception is restarted.

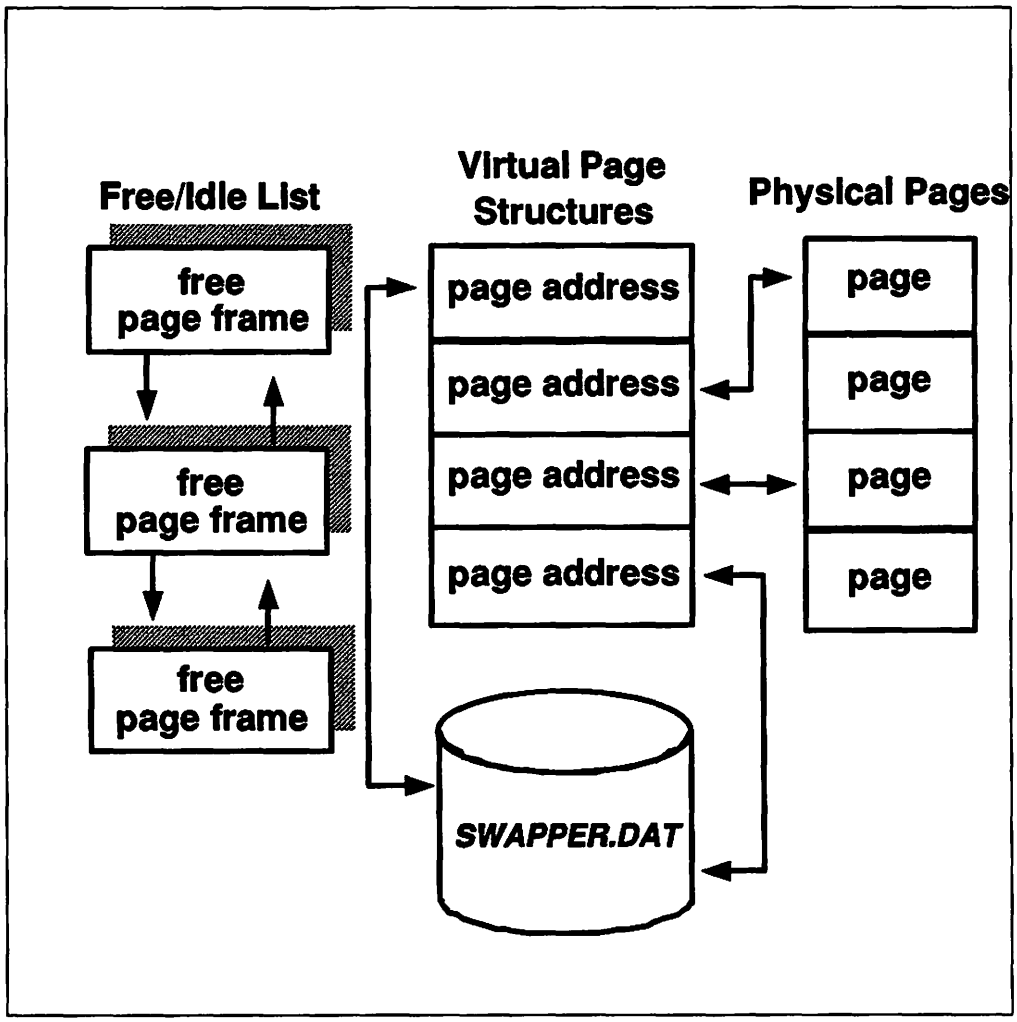


Figure 6. Page Swapping

2.5 16-Bit Applications in a 32-Bit Environment

One of the major concerns when developing OS/2 Version 2.0 was to ensure compatibility with applications written for previous versions of OS/2. Since there are a significant number of 16:16 modules and resources in existence, consideration was also given to the coexistence and interaction of 16:16 and 0:32 modules and resources within an application, with issues such as:

- 0:32 applications using existing 16:16 DLLs and libraries
- 16:16 applications using 0:32 DLLs and libraries
- "Huge" segments (greater than 64KB in size).

The 16:16 applications are placed in the compatibility region and accessed through **tiled local descriptor tables (LDTs)**; see 2.5.1, "Address Conversion and Translation" for further explanation. Such applications take advantage of the fact that memory in the compatibility region may be addressed with both 16:16 and 0:32 addressing schemes. Addressing memory objects from both 16-bit and 32-bit applications is therefore quite simple, since no physical movement of the memory object is required. Since the entire process address space under OS/2 Version 2.0 is located within the compatibility region, this also means that communication between 16-bit and 32-bit applications and modules may take place.

2.5.1 Address Conversion and Translation

Under previous versions of OS/2, an application addressed memory using the segmented memory model, which was translated to physical memory addresses by the operating system using a local descriptor table (LDT), with one LDT per process. Under OS/2 Version 2.0, 16:16 references (see 2.2, "Flat Memory Model" on page 14) are managed by a technique called **LDT tiling**, which provides compatibility for 16-bit applications and provides a mechanism for 32-bit applications to utilize 16-bit libraries.

A tiled LDT contains up to 8192 descriptors, where the segment base address in each descriptor is a multiple of 64KB, and each descriptor therefore points to a 64KB region of memory. Contiguous descriptors map into a contiguous linear address space, thereby using a potential maximum of 512MB, and allowing a 16-bit application to address the 512MB of the process address space in the compatibility region. Figure 7 on page 29 shows the way in which memory addresses within the tiled LDT are mapped into the process address space.

The addresses within the process address space may be referenced by applications or modules using the 16:16 addressing scheme, in a similar manner to previous versions of OS/2. However, the same physical memory locations may also be accessed by 32-bit applications and modules using the 0:32 addressing scheme. Both the LDT entries used by the 16:16 scheme and the page table entries used by the 0:32 scheme may translate to the same memory locations. This translation enables 32-bit applications to make use of 16-bit modules and resources, and allows 32-bit and 16-bit applications to coexist and communicate with one another.

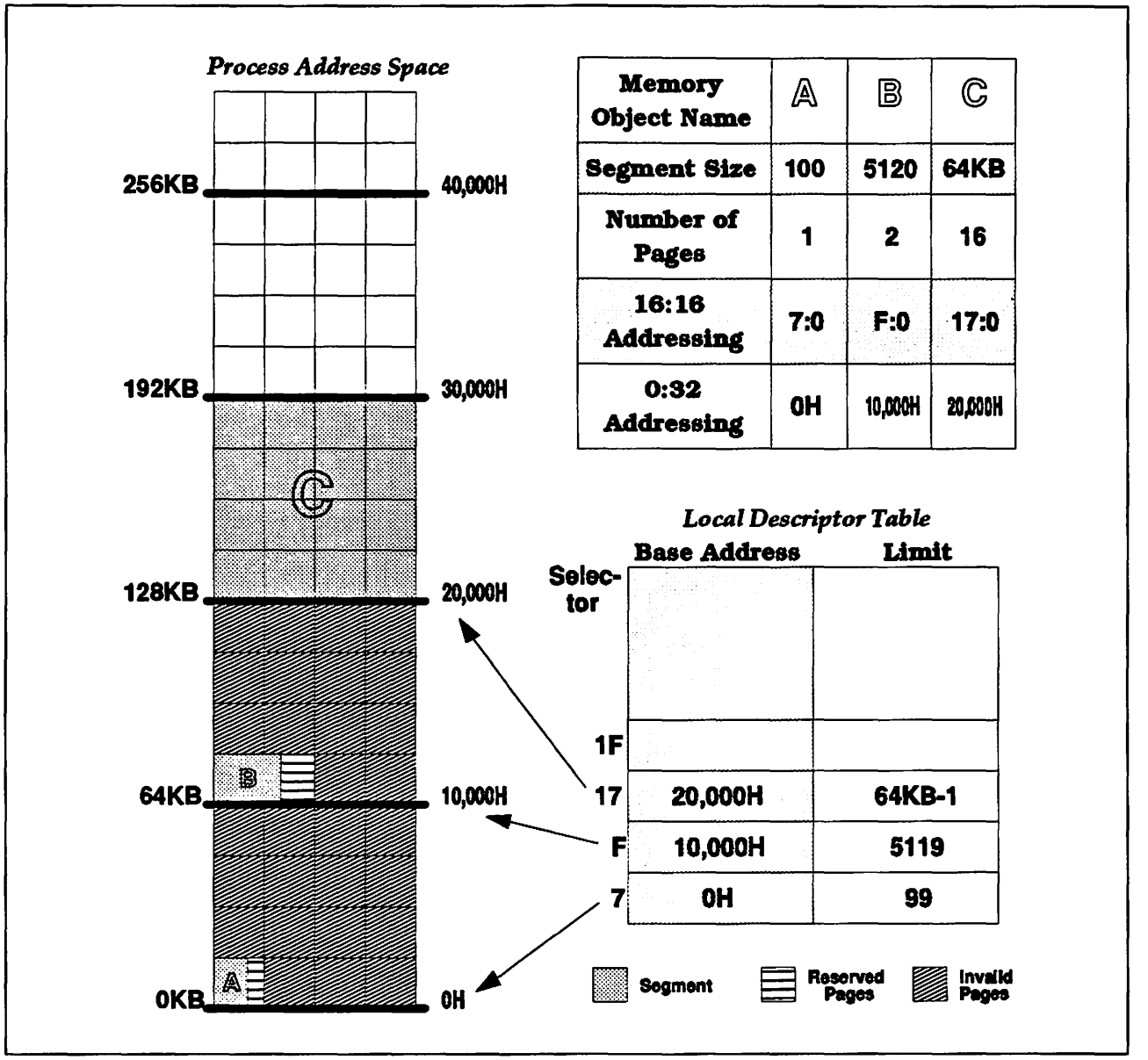


Figure 7. Mapping 16:16 Memory References

LDTs are managed in a different way from previous versions of OS/2. Each LDT is allocated as a sparse object until descriptors are inserted upon loading an application. Descriptors for shared memory objects are inserted downwards commencing at the top of the LDT, whereas private memory object descriptors are inserted upwards commencing at the bottom of the LDT. This order reflects the management of the linear address space by the operating system. Therefore the minimum LDT size is 8KB, using one page for the shared descriptors and one page for the private descriptors. Note that each code or data selector reserves a full 64KB of linear address space to be able to handle an eventual reallocation of segments to the maximum size of 64KB. See Figure 7.

The following memory objects use LDT descriptors:

- 16-bit .EXE files
- 16-bit .DLL files
- **DosAllocSeg()** calls

- **DosAllocMem()** calls with tiling
- 32-bit .EXE files with tiling
- 32-bit .DLL files with tiling.

This use of descriptors is effectively equivalent to the implementation used in the previous versions of OS/2. It must be noted, however, that a memory object greater than 64KB may not be handled in an orderly manner by a 16-bit application.

A memory object allocated in the compatibility region has both a 16:16 address (far16) *and* a 0:32 address (near32), allowing access by applications using either addressing scheme. The two types of addresses are related by the following arithmetic functions:

- ***near32 = SEL (far16) >> 3 << 16 + OFFSET(far16)***
- ***far16 = MAKEP (HIGH (near32) << 3 + 7, LOW (near32)).***

The term *near32* refers to the fact that all 0:32 calls are of type *near* (based on offset only). The term *far16* refers to the fact that 16:16 calls are based on the segment selector and offset used in combination. Bit shift right is shown as >>, and bit shift left is shown as <<.

32-bit executable modules can therefore create 16:16 aliases for memory objects in the compatibility region and conversely, 16-bit modules may create 0:32 bit aliases. As these conversions are arithmetically based, a high performance address conversion layer may be implemented, assuring automatic address conversion between 16:16 and 0:32 memory objects. See 2.5.2, "Thunking."

2.5.2 Thunking

Address conversion between 16:16 and 0:32 addressing models is achieved by the use of a **thunk**. A thunk exists for each programming interface in the system. Thunking implies:

- **Converting the addressing scheme used (0:32 - 16:16)**

  OS/2 Version 2.0 uses a flat linear (0:32) addressing scheme. 16:16 program modules expect a selector:offset (16:16) addressing scheme. The thunk converts memory references between these two schemes. See Figure 8.

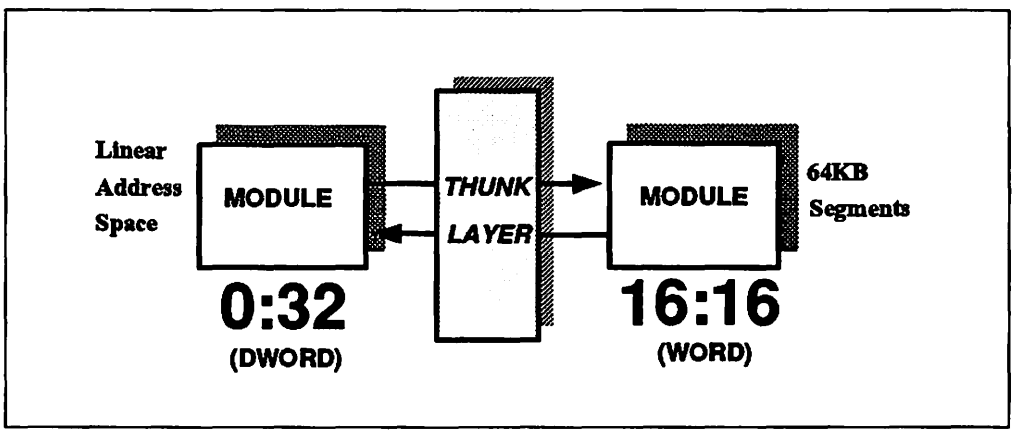


Figure 8. Thunk Concept. Note that a thunk performs services other than address translation, such as structure and stack realignment. See text.

- **Different parameter sizes (DWORD versus WORD)**

The 0:32 addressing scheme uses 32-bit (LONG or DWORD) values as the basic data type. The 16:16 scheme uses a 16-bit (SHORT or WORD) value as the basic data type. The thunk converts between WORD and DWORD length data.

- **Structure alignment**

  The 16:16 addressing scheme normally causes data structures to be WORD aligned, whereas the 0:32 addressing scheme defaults to DWORD alignment; blank space is included within the structure so that each element is aligned on a DWORD boundary. The thunk must realign data structures where necessary.

- **Stack conversion**

  The 0:32 stack is DWORD-based. The 16:16 stack is WORD-based. Stack-based addressing between the two schemes is therefore different. The thunk must make a new copy of the parameters on the stack, realigning when needed.

- **Restrictions on the 16:16 addressing scheme**

  The 16:16 code can only address up to 64KB in any segment. The only limit on the 0:32 code is the maximum size of the linear address space (4GB). This disparity creates a problem when a data item in the 0:32 module or resource is larger than 64KB or is allocated across a 64KB boundary, and must be passed to a 16:16 routine. Where possible, the thunk must make the data item addressable by the 16:16 routine.

- **Different call models**

  The 0:32 addressing scheme uses *near* calls for all operating system functions. The 16:16 scheme uses *far* calls for operating system functions. If a procedure using one scheme tries to call a procedure of the other scheme, different return values may be placed on the stack. The thunk is responsible for producing the correct calling sequence.

The above considerations apply to all executable programs, libraries (both statically and dynamically linked) and Presentation Manager messages.

OS/2 Version 2.0 provides a number of thunks to handle function calls from 16-bit applications to 32-bit service layers, and vice versa. Almost all of the thunks in OS/2 Version 2.0 are of the 0:32 to 16:16 conversion kind, except for those that handle semaphores and **DosSubxxxx()** service calls. Thunks are packaged with the module that contains the supporting code for the 16-bit and/or the 32-bit entry point. See Figure 9 on page 32 for an overview of the two different types of thunks.

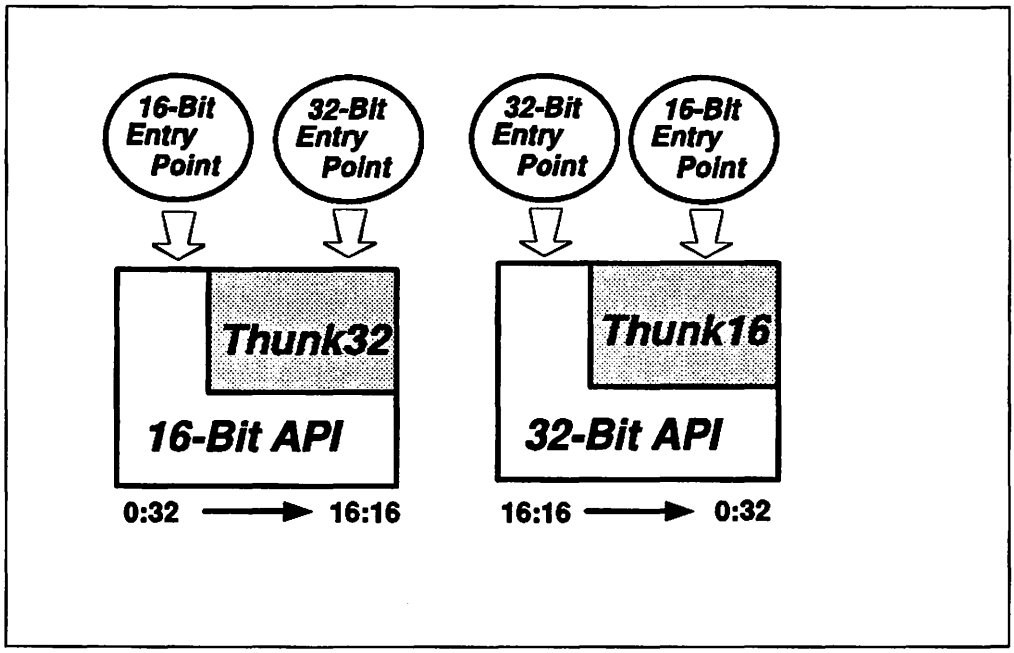


Figure 9. Thunks -16-Bit versus 32-Bit

This implementation of mixed 16:16 and 0:32 applications also raises a number of considerations:

- Each resource will effectively occupy at least 4KB (one page), since this is the minimum allocated in the 32-bit environment.
- Stack size is a maximum of 64KB, since this is the maximum addressable by the 16-bit portion of the application.
- In Presentation Manager applications, caution must be exercised when passing user-defined messages between window procedures in modules that use different addressing models; such messages will require application-defined thunks.
- Presentation Manager hooks should *only* be used with system-defined message contents.

When developing 32-bit applications using 16-bit modules, the programmer is responsible for supplying appropriate address conversion via thunks, for pointers passed as parameters and for application-defined messages under Presentation Manager. These considerations and restrictions are discussed further in *OS/2 Version 2.0 - Volume 4: Application Development*.

2.5.3 Shared Memory

When memory objects are shared between processes in the 16-bit environment, the processes may share either the linear address spaces or the actual physical pages, depending upon the way in which the memory is shared. The allocation of shared memory by 16-bit applications using the **DosAllocShrSeg()** function involves shared linear addresses, which are referenced by different LDT entries in each 16-bit process. This mapping of LDT entries to linear addresses is shown in Figure 10 on page 33.

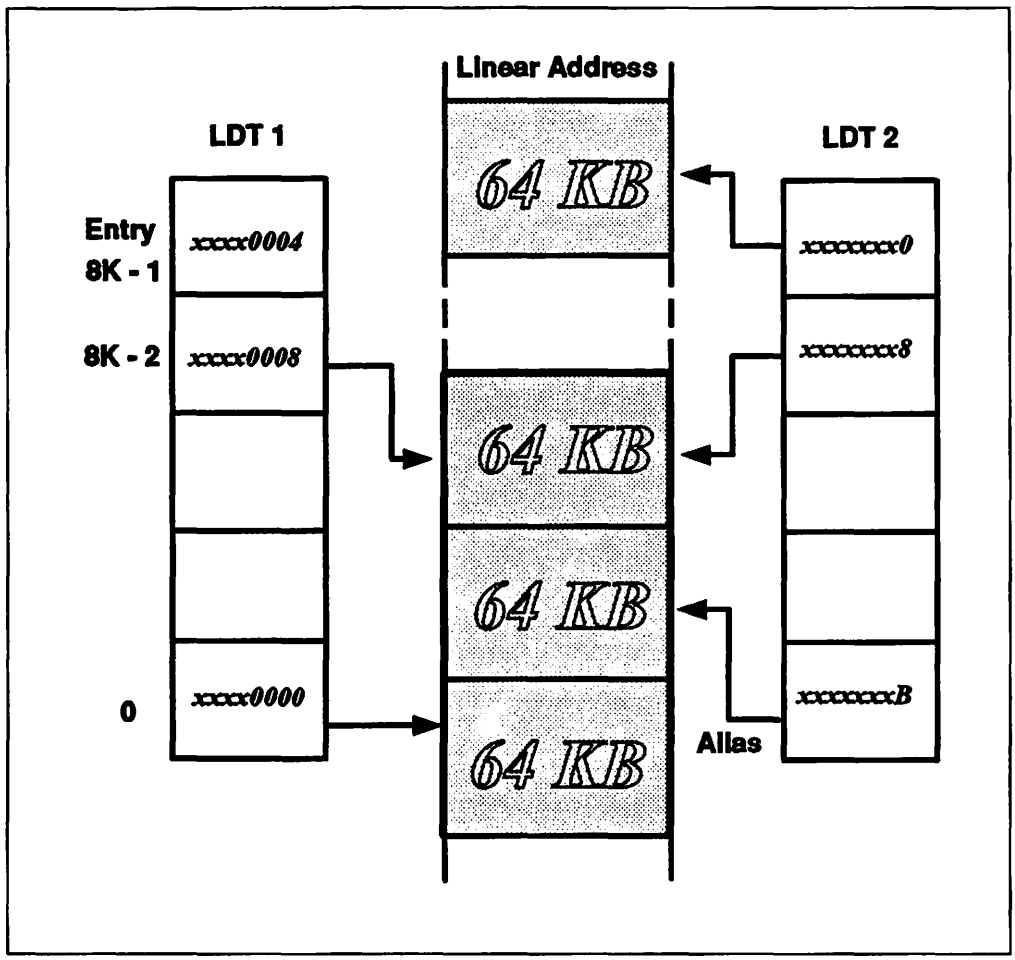


Figure 10. 16:16 Shared Address Ranges

Memory objects are shared using the alias technique, whereby the different 16:16 memory references used by each application are mapped by the operating system, using the tiled LDTs, to the same locations in memory. This is very similar to the technique used for shared memory in a pure 0:32 environment.

2.5.4 Program Loading

OS/2 V2.0 will load programs linked by both LINK386 (the linear executable linker) and by LINK (the segmented executable linker). Executables linked under previous versions of OS/2 can be executed without relinking on OS/2 V2.0.

Included in the executable files are relocation records (also referred to as fixup records), because not all address references can be resolved at the time of linking. These records are used by the OS/2 program loader to resolve addresses that were unknown at link time. There is a difference in the way that LINK386 and LINK packages fixup records. LINK386 fixup records are identified by the page to which they belong, whereas LINK produces fixup records by segment.

LINK386 executables are page orientated. LINK executables are segment orientated. The loader employs different strategies when handling modules produced by the different link editors. In both cases there is no preloading of any part of the executable. Loading only takes place when an attempt is made to access a section of the module, which is not already loaded.

In the case of LINK386, the executable will be loaded on a page-by-page basis. Reloading parts of a module that has been discarded is no problem since when a page fault occurs, the contents of the page and its fixup records can easily be recovered from the EXE or DLL file on disk.

With LINK executables, the situation is different as everything is segment based. The procedure that is followed depends on whether a particular segment is swappable or discardable. Again there is no preloading of segments. The initial load of a particular segment is triggered by a segment not present fault. At this time, sufficient pages to hold the segment will be allocated. The complete segment is loaded and fixups applied. If the segment and hence the pages that contain it are swappable, the fixup records are discarded. In the case of discardable segments, the fixup data is reorganized on a per-page basis and held in swappable memory. Pages of a swappable segment are recovered from the swap file. When a page of a discardable segment needs to be recovered, it must be reloaded from the EXE or DLL file. After the first load of the segment, only the page needs loading, as the fixup data is now available by page. Generally read-only pages are discardable, read/write pages are swappable.

However there is a further complication. Memory is always allocated in a multiple of 4KB pages. If a segment contains 20 bytes or 4000 bytes, one page will be allocated to contain it. There is a potential for using far more memory to load a program than the program actually requires. To overcome this, OS/2 V2.0 uses a technique called segment packing. A single page may contain parts from two or more segments. When segments are packed onto a page, the page is always marked swappable. This is true even if all segments on the page are read-only.

By default, segment packing is enabled. The **MEMMAN** statement has a new option **NOPACK**, which turns off segment packing. A segment will be considered for packing, if there are other qualifying segments, its is less than a certain size and if loaded on its own, would result in there being significant free space in the last or only page into which the segment is loaded. The current recommendation would be to leave segment packing enabled.

2.6 Summary

OS/2 Version 2.0 provides significant enhancements in memory management over previous versions of OS/2. Full use is made in Version 2.0 of the 32-bit addressing and paged virtual memory capabilities of the Intel 80386 processor, giving theoretical access to 4GB of memory per process. In practical terms, however, it is unlikely that the full amount of memory could be used, due to other constraining factors such as hardware limitations.

OS/2 Version 2.0 manages its memory as a single linear address space of up to 4GB in size. This global address space is divided into a number of regions. The region below 512MB is known as the process address space, and is available to applications for storage of executable code, resources and data. The region above 512MB is reserved for operating system use. The choice of 512MB as the dividing line between the two regions allows 16-bit applications and resources, written for previous versions of OS/2, to be executed and address memory within the process address space.

OS/2 Version 2.0 allocates memory in multiples of 4KB; each 4KB unit is known as a page. An application may request larger memory objects, and may access

and manipulate these objects as logical entities, but the operating system internally manages each page as a separate unit. This allows a more efficient virtual memory implementation, since individual pages may be swapped in and out of real storage, rather than entire memory objects. Page swapping is typically faster, especially when memory objects become very large.

OS/2 Version 2.0 allows 16-bit applications and resources to execute within any process, since all processes execute in memory within the 16:16 addressing scheme limit of 512MB. OS/2 Version 2.0 also allows interaction between 16-bit and 32-bit modules within a process, and provides address conversion, parameter conversion, stack alignment, etc., between 16:16 and 0:32 addressing schemes using thunk layers.

The memory management implementation under OS/2 Version 2.0, therefore, provides access to larger amounts of physical and virtual storage in a more efficient manner than previous versions of OS/2, and removes many of the constraints imposed by previous versions. At the same time, it maintains compatibility with 16-bit applications and resources.

Chapter 3. Task Management

The task management component of OS/2 Version 2.0 controls the execution and dispatching of processes and threads started by the user or by applications. The design objective of task management in OS/2 Version 2.0 was to meet the following criteria:

- Support *all* existing 16-bit applications written for previous versions of OS/2
- Support the development and execution of 32-bit applications
- Relieve system constraints on resources such as threads
- Simplify the management of processes and threads within applications.

This chapter discusses the implementation of task management under OS/2 Version 2.0 in order to satisfy these criteria.

3.1 Dispatching

OS/2 Version 2.0 treats all processes executing in the system as protected mode processes. No real mode support is provided by OS/2 Version 2.0; such support is not necessary since the Multiple Virtual DOS Machines feature allows DOS and DOS extender applications such as Microsoft Windows 3.0 applications to run in virtual DOS machines, which are regarded by the operating system as protected mode tasks in their own right. In this way, full pre-emptive multitasking and memory protection are available to *all* applications executing in the system, including DOS and Microsoft Windows applications.

3.1.1 16-Bit Application Support

OS/2 Version 2.0 supports the execution of 16-bit applications directly, with no modification required. These applications execute as protected mode processes under operating system control, exactly as they did under previous versions of OS/2. Full memory protection and interprocess communication facilities are available to such applications.

OS/2 Version 2.0 also allows both 16-bit and 32-bit modules and resources to reside and execute within the same process. Address conversion and translation is accomplished using tiled local descriptor tables and thunks. These concepts are fully described in 2.5, "16-Bit Applications in a 32-Bit Environment" on page 28.

3.1.2 32-Bit Application Support

To take full advantage of the 32-bit flat memory model implemented by OS/2 Version 2.0, a full set of 32-bit application programming interfaces is provided for developers. This allows memory allocation in logical units dictated by the requirements of the application rather than by the constraints of the segmented memory model implemented under previous versions of OS/2. Simplified memory management within an application allows developers to write applications that may be more easily migrated to other operating system platforms, perhaps running on non-Intel processors.

These 32-bit applications are executed as protected mode processes within the system, and have the same multitasking, multithreading, memory protection and

interprocess communication facilities available to 16-bit applications, plus some additional facilities implemented under OS/2 Version 2.0.

3.2 Interrupt Handling

The interrupt manager component of the operating system is responsible for maintaining the interrupt context for the system. It dispatches hardware interrupts to device drivers which are registered for those interrupts. The interrupt manager is also responsible for supervising the use of the programmable interrupt controller (8259 PIC). Since performance is always a major issue in interrupt handing, particularly with time-critical applications, special considerations have been taken to ensure the maximum possible performance levels.

The functions performed by the interrupt manager are basically similar to those implemented for previous versions of OS/2. The interrupt manager provides the following:

- High-performance routing of interrupts to device drivers
- Compatibility with 16-bit **DevHlp()** interfaces
- Support for PS/2-like interrupt devices (level-triggered interrupt sharing)
- Support for PC AT-like interrupt devices (edge-triggered interrupts)
- Efficient interrupt packaging/adaptation for OEM extensions.

The internal implementation of the interrupt manager is very similar to previous versions of OS/2. The only major difference between the interrupt manager in OS/2 Version 2.0 and previous versions is that real mode interrupt handling is no longer supported; this method of interrupt handling is no longer needed due to the provision of the Multiple Virtual DOS Machines feature in OS/2 Version 2.0, which supports DOS applications as virtual 8086 mode tasks. All hardware and software interrupts from virtual 8086 mode tasks in an 80386 system cause a switch to protected mode. Thus interrupt requests issued by DOS applications are translated to the appropriate protected-mode device service requests by the virtual DOS machine. This process is explained more fully in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

Upon servicing an interrupt, the interrupt manager reschedules all threads currently pending in the system. This rescheduling is done to ensure that high priority threads are scheduled and dispatched as soon as possible. The only exception is the operating system kernel, which cannot be preempted. Forced actions such as Ctrl+Break or **DosKillProcess()** must be serviced before more application code is executed. This priority is necessary to avoid situations where a process is terminated by the user and an instruction from this process is then executed before the operating system processes the interrupt. Interrupt priorities are therefore checked by the interrupt manager upon receiving each interrupt.

3.3 Signal and Exception Handling

Under previous versions of OS/2, signals were dispatched from a number of sources and were used to interrupt executing processes. These signals included Ctrl-C, Ctrl-Break, and signals generated by the **DosKillProcess()** function call.

Under OS/2 Version 2.0, signal handling has been merged with exception handling to provide a general, portable mechanism for handling all such events. OS/2 Version 2.0 provides four new system functions for exception handling on a per-thread basis:

- **DosRaiseException()**
- **DosSetExceptionHandler()**
- **DosUnsetExceptionHandler()**
- **DosUnwindException()**.

The **DosSetSignalExceptionFocus()** function is used to allow 32-bit applications to inform the operating system that they are to be the focus for Ctrl+Break and Ctrl-C signals. However, these signals are dispatched and treated as exceptions.

Applications may register their own routines using the **DosSetExceptionHandler()** function to handle specific types of exceptions, including general protection exceptions which could not be trapped by applications under previous versions of OS/2. These exception handlers may be written in high-level languages such as "C"; they are not required to be written in assembly language, as was the case with previous versions of OS/2.

One significant difference in exception handling between OS/2 Version 2.0 and previous versions of OS/2 is the ability, under Version 2.0, to handle general protection exceptions within the application. Under previous versions, such exceptions invariably resulted in application termination with a Trap 000D error; with Version 2.0 however, an application may recover or at least terminate in an orderly manner.

The generalized exception-handling approach has several advantages:

- Exception handlers may be chained and/or nested, and each handler may decide whether or not to pass an exception to subsequent handlers.
- Exception handlers under OS/2 Version 2.0 may be written entirely in a high-level programming language; assembly language routines are not required. This reduces the dependence on the Intel 80386 architecture, and facilitates porting applications to other operating system and hardware platforms.

Note that OS/2 Version 2.0 does not allow applications to register an exception handler for a "coprocessor not found" exception. Instead, the operating system itself traps such exceptions and provides transparent floating-point emulation on a per-thread basis.

Note that OS/2 Version 2.0 provides its own exception handlers within the service layers for all 32-bit system functions. These exception handlers allow the service routines to recover from page fault exceptions and general protection exceptions encountered due to bad pointers in applications' function calls. The function call returns an ERROR_BAD_PARAMETER code rather than a Trap 000D or Trap 000E code, thereby allowing the application to more easily recover from the error. This recoverability represents a significant enhancement over previous versions of OS/2, since it allows easier debugging and more flexible pointer handling, without the necessity for application-registered exception handlers to process these exceptions.

Exception handling within applications is described in detail in *OS/2 Version 2.0 - Volume 4: Application Development*. The reader will also find programming examples of other topics, covered in this chapter, in that volume.

3.4 Thread Management

Under previous versions of OS/2, there was a limitation of 53 threads per process. Under Version 2.0, this limitation is lifted, and the thread limit for each process is now the same as the overall system limit, that is, a total of 4096 threads. Note, however, that the operating system itself consumes some threads, so the practical number of threads available to an application will be fewer than 4096.

The number of threads available is also affected by the THREADS= statement in CONFIG.SYS. This statement sets the maximum number of threads available in the system; hence the number of threads available to applications will be the number specified in this statement, less the number of threads consumed by the operating system.

3.4.1 Creating Threads

The preparation and dispatching of secondary threads within applications is greatly simplified under OS/2 Version 2.0. Under previous versions of OS/2, the application was required to explicitly allocate and deallocate memory for the stack segment of a secondary thread. Version 2.0 allows the operating system to perform the allocation of the stack memory object on the application's behalf as part of the **DosCreateThread()** function, and to free the stack segment when the thread terminates. The application need only specify the required size of the segment in the **DosCreateThread()** call.

OS/2 Version 2.0 uses the guard page technique described in Chapter 2, "Memory Management" to progressively commit each thread's stack during execution, up to the maximum size specified in the **DosCreateThread()** function. This minimizes the real storage requirement of the stack, makes the most efficient use of available real storage, and thereby improves overall system performance.

3.4.2 Controlling Threads

Under previous versions of OS/2, control information for threads was stored in a number of different locations and accessed via a number of function calls. Under Version 2.0, much of this information has been combined into a **thread information block (TIB)**. The TIB contains the following information for the current thread:

- Process identifier
- Parent process identifier
- Module handle
- Command line pointer
- Environment block pointer
- Thread identifier
- Priority
- Base address of the thread's stack
- Stack size
- Pointer to the exception handler chain for the thread.

The TIB is accessed using the **DosGetInfoBlocks()** function, which replaces the **DosGetPrty()**, **DosGetPID()**, **DosGetPPID()** and **DosGetInfoSeg()** functions.

3.4.3 Waiting On and Terminating Threads

Under previous versions of OS/2, when an application wished to determine whether an asynchronous thread had completed execution, it was required to do so through the use of semaphores, which were periodically checked by the waiting thread. OS/2 Version 2.0 provides a **DosWaitThread()** function which allows an application to determine whether a given thread is currently executing, and to either wait for the thread to complete, or time-out immediately.

A thread may also be terminated from within an application using the **DosKillThread()** function. Under previous versions of OS/2, a thread could not be forcibly terminated by another thread, unless its parent process was also terminated.

3.5 Semaphores

OS/2 Version 2.0 provides significantly enhanced semaphore facilities for synchronization between threads and processes. In order to maintain compatibility with 16-bit applications, the task management component handles both the older semaphores implemented in previous versions of OS/2, and the new types introduced in Version 2.0.

OS/2 Version 2.0 provides two classes of semaphores: **private** and **shared** semaphores. A process may have up to 64KB of private semaphores, available only to threads within that process, and may also access up to 64KB of shared semaphores, available to all processes in the system.

There are three different types of semaphores in OS/2 Version 2.0:

- **Mutex** semaphores provide mutually exclusive access to a particular resource.
- **Event** semaphores are used to signal system or application events.
- **MuxWait** semaphores may be used when waiting for multiple events to occur or multiple mutex semaphores to clear.

A particular area of enhancement is that of the suspension/reactivation of threads based on semaphores. In previous versions of OS/2, the internal reactivation of threads was based on a "multiple wakeup" approach. When a thread wished to wait for some event, it was placed in an event ID table. Threads waiting for that event were queued in the order in which their wait request occurred. When the event occurred, all threads waiting for this event would contend for the semaphore, and the first queued thread or the thread with the highest priority would be dispatched. However, this algorithm led to considerable system overhead.

Under previous versions of OS/2, the event ID table was examined using a linear search to find waiting threads. While this method was acceptable for a system with a maximum of 512 threads, it is clearly not suitable for a system with 4096 threads. OS/2 Version 2.0 provides more sophisticated semaphore services, allowing both "single wakeup" and "multiple wakeup" services in a more efficient manner.

Under Version 2.0, the event ID table is implemented using a hashing algorithm, thereby increasing the speed of access to the table and thereby improving performance when resolving thread contention. This algorithm provides a more efficient method of handling the 4096 threads allowed under Version 2.0.

3.6 Summary

The task management component of OS/2 Version 2.0 is greatly improved over that implemented in previous versions of OS/2, while maintaining backward compatibility for applications and resources written to previous versions. Significant effort has been made to enhance performance and remove system limitations experienced in previous versions.

The result of these efforts is summarized below:

- All 16-bit applications and resources may be used in the OS/2 Version 2.0 environment.
- New 32-bit applications are able to exploit the benefits of the 80386 processor.
- The thread limitation of 53 threads per process is removed.
- The thread limitation for the entire system is increased from 512 to 4096 threads.
- The process limitation for the entire system is increased to 4095.
- The operating system contains more swappable code, thereby reducing its real memory requirements and allowing more storage to be used by applications.

These enhancements provide a greater diversity of function for the application developer, allowing more sophisticated applications to be written and executed under OS/2 Version 2.0 using multiple threads and processes, while reducing the effort required to synchronize and communicate between threads and processes.

Threads may be more easily created, controlled and terminated under OS/2 Version 2.0 than under previous versions of OS/2, thereby reducing the effort required to generate high-function, multithreaded applications which fully exploit the OS/2 Version 2.0 platform.

Improved efficiency in module loading and execution results in enhanced application performance. Improved mechanisms for resolution of semaphore contention allow improved dispatching and thereby reduce overhead, further enhancing overall performance.

Chapter 4. Debugging Support

The debug component of OS/2 Version 2.0 consists of a set of system services which make it possible to trace and locate problems in programs, in order to increase productivity during the testing phase of the application development process. These services make use of the debug registers present in the Intel 80386 processor to implement features such as data watchpoints and soft instruction breakpoints. By using these registers, it becomes possible to implement watchpoints and breakpoints without making modifications to the code being debugged.

The debug component *does not* provide a complete debugging tool, but merely provides a mechanism whereby one process may control and monitor the execution of one or more related processes. It provides the services which are used by debugging tools such as the C Set/2 debugger. Debug provides functions such as single-stepping through a program, interrogation of processor registers and locations in memory and changing the contents of registers and locations in memory. The debugging tool provides the user interface which displays source code listings, allows the user to set breakpoints, start execution of a program and to monitor changes in the value of program variables.

4.1 Functional Description

The main objective of the debug component is to provide the required services to enable developers to build powerful debugging tools. The interface provided by the debug component is designed to be:

- Fast
- Powerful
- Flexible
- Non-intrusive
- Accurate
- As portable as possible
- Able to debug programs and DLLs
- Able to debug mixed or pure 16 and 32-bit applications.

The term "non-intrusive" means that debugging services must:

- In no way compromise the integrity of the operating system
- Minimize any impact on system performance
- Have no effect on processes, other than the process or processes being debugged.

Debuggers usually require that a program, to be debugged be preprocessed accordingly, compiled and linked with special options. If a program is prepared for debugging but not actually being debugged, the debugging services should have no impact on the execution of the program. Particularly application performance should not be affected in any way. These design objectives of the OS/2 V2.0 debug component should be kept in mind by the developers of debugging tools.

The power of the debug component is an important issue to consider when designing and developing debugging tools. Since the debugger gains access to the memory and registers of the application being debugged, it is the responsi-

bility of the debugging tool developer to ensure the correct use of these resources to prevent deadlocks or inadvertent corruption of application resources.

Debuggers should not be used as tools for evaluating program performance. The debug component does not provide reliable timings for the execution of program steps. It is useful, however, for examining the execution path of a particular transaction through a program.

4.2 What Can Be Debugged

The debug component sees a process as the unit that will be debugged. Debugging is managed on the basis of a connection between a debugger and a process that was started to run the program that is to be debugged. The debugging tool will usually issue either a **DosExecPgm** or a **DosStartSession** call to start the program. When starting the program, flags must be set in the call to request that the program be started for debugging. Whether the program is started as a child process in the same session as the debugger or as a process in a child session, the connection between the debugger and the program is process to process.

The debug component can be used to debug programs running in:

- The same screen group (foreground session)
- Another screen group
- Detached mode (background session).

A debugger may debug more than one process simultaneously, but any process can only be connected to one debugger at any one time. All threads within a process are accessible to the debugger.

The debugging services allow access to code and data in both a primary executable module and in DLLs. Resources such as DLLs are not associated directly with threads, but are accessed and manipulated by threads. The debug component, therefore, supports debugging of the DLLs by permitting the debugger to observe the memory of the DLL and the registers of a thread while it uses the DLL.

The debug component notifies the debugging tool when a DLL is loaded, and provides the debugging tool with the handle of the module table entry (MTE) of the DLL. From this the debugging tool is able to determine the name of the DLL by issuing a **DosQueryModuleName** call.

The debug component provides control and information at a hardware level. It provides access to a process's registers and memory. It can be used to start a process executing and provide feedback when the process stops executing, usually because a breakpoint has been encountered. It provides a notification when a new thread is created in a process and when the contents of designated memory location change. It does not trace the passage of system API calls through the system. It does not understand resources implemented by the operating system. The debug component provides no information about:

- Semaphores
- Pipes
- Queues
- Files.

The debugging tool, therefore, cannot give feedback on the status of semaphores or the contents of pipes and queues. It is up to the debugging tool to provide facilities that the application programmer would find useful. One way of providing these facilities is for the debugging tool to include additional routines in an executable module that is being prepared for debugging. The execution of these routines could be controlled in the same way that the debugging tool controls execution of the program being debugged.

The debug component cannot be used to debug processes running in virtual 8086 mode (that is, applications executing within a virtual DOS machine).

4.3 DosDebug() Function

The debugging services provided by the operating system are accessed using the **DosDebug()** function. Debugging services are obtained by repeated calls to **DosDebug()**, thereby simplifying the interface and increasing portability.

This function replaces the **DosPTrace()** function implemented in previous versions of OS/2. However, the **DosPTrace()** function is still available under OS/2 Version 2.0 as a 16-bit entry point, thus allowing debugging tools written for previous versions of OS/2 to be used to debug 16-bit applications.

The typical application programmer is unlikely to use the DosDebug() function. It is intended to be used by developers of high-level application debugging tools.

The syntax of **DosDebug** call is shown in Figure 11.

```
DosDebug enables the calling application to control another application for
debugging purposes.

#define INCL_DOSPROCESS
#include <os2.h>

PVOID   pDbgBuf;          /*Pointer to the Debug Buffer Structure*/
APIRET  rc;               /* Return code */

rc = DosDebug(pDbgBuf);

Return codes

DosDebug returns the following values:

0       NO_ERROR
87      ERROR_INVALID_PARAMETER
95      ERROR_INTERRUPT
115     ERROR_PROTECTION_VIOLATION
```

Figure 11. DosDebug Function

The *DbgBuf* parameter is a pointer to a data structure, which contains values used to control the execution of the **DosDebug()** function. One of the fields within the structure is used to hold a command word. On return, the same field contains a notification code. Commands may result in the reading or writing of memory, single stepping through the program, changing values in registers or

the setting of watchpoints. The notifiers indicate events such as the loading of a DLL, encountering a breakpoint during program execution and watchpoint hits. The command to be executed is placed in the *DbgBuf* structure together with the necessary additional information for that specific command. The structure contains fields for all registers present in the system. There are other fields in the structure which are command specific. There are also fields for the process ID and the thread ID which identify the target for the debug command.

The **DosDebug()** function, its commands, parameters and return codes are described in detail in the *IBM OS/2 Version 2.0 Control Program Reference*.

The **DosDebug()** function executes synchronously; that is, the debugging process executing the call must wait for the function to complete and return control before it may continue execution.

The result of the **DosDebug()** call is a return code and an update of the command field with a notification code. See Figure 12.

```
DbgBuf.Pid = Pid;       /* The process ID to receive the command */
DbgBuf.Cmd = DBG_C_Go;  /* The command to be issued              */

rc = DosDebug (DbgBuf); /* Issue the command                     */

                        /* Display the result                    */
printf("Return code = %u, command field = %u \n", rc, DbgBuf.Cmd);
```

Figure 12. Sample DosDebug Function Call

If this is the first *DBG_C_Go* command to be issued, a typical return code is RC = 0, and the notifier placed in the command field is set = *DBG_N_ModuleLoad*. This return implies that the application has just loaded an executable module and will always occur at program start. A notification is generated for each module loaded. Because there may be several DLLs loaded when the program is started, there will be several notifications pending. Only a single notification is returned on each **DosDebug** call. The debugger must issue *DBG_C_Stop* commands to retrieve all the outstanding *DBG_N_ModuleLoad* notifications.

4.4 Summary

OS/2 Version 2.0 provides a comprehensive set of debugging facilities which may be used to develop powerful application debugging tools. These facilities make use of the debug registers present in the 80386 processor, and allow debugging on a per-process basis. The debugging of multi threaded applications is possible. It is also possible to debug code residing in DLLs.

Access to the debugging facilities is provided through the **DosDebug()** function call. Debugging services are requested by repetitive use of this function call with different command parameters.

The **DosDebug()** function replaces the **DosPTrace()** function implemented in previous versions of OS/2. However, the **DosPTrace()** function is implemented under OS/2 Version 2.0 as a 16-bit entry point, thereby allowing debugging tools written for previous versions of OS/2, and which use the **DosPTrace()** function, to be used under Version 2.0.

Chapter 5. Installation Considerations

The installation of OS/2 Version 2.0 is greatly simplified when compared to previous versions of OS/2, through the provision of a graphical installation procedure. This installation procedure addresses a number of the shortcomings experienced in previous versions of OS/2:

- It provides greater feedback to the user during installation.
- It provides a more flexible way for users to customize their system.

  The user is able to add optionally installable features at any time and to change the system configuration without having to boot from diskette.
- It uses the standard OS/2 Presentation Manager user interface to guide the user through the installation process.
- It supports installation from any drive to any drive.

  This feature makes it possible to install OS/2 V2.0 from a drive on a LAN server, CD-ROM or any other media which can be accessed as a "normal" drive letter. The process of installation over a LAN is described in *OS/2 Version 2.0 Remote Installation and Maintenance*.

The installation procedure provided in Version 2.0 will in most cases be able to sense the hardware configuration of the system on which it is being installed, and will select the required device drivers. This awareness applies to the video and disk subsystems, keyboard and pointing devices. The user will be given the option to change or add to the configuration determined by the installation procedure.

5.1 Pre-Installation Planning

The following considerations should be addressed prior to commencing installation of OS/2 Version 2.0.

5.1.1 Processor Requirements

OS/2 Version 2.0 is designed to use the instruction set of the Intel 80386 processor, and therefore requires a system unit equipped with either an Intel 80386 DX, SX, SL or SLC processor, or subsequent Intel processors such as the 80486 series, which implement the same processor instruction set. IBM machines equipped with such processors include:

- IBM Personal System/1* (2121)
- IBM Personal System/2 Model N33 SX (8533)
- IBM Personal System/2 Model 35 SX (8535)
- IBM Personal System/2 Model 40 SX (8540)
- IBM Personal System/2 Model L40 SX (8543)
- IBM Personal System/2 Model N51 SLC/SX (8551)
- IBM Personal System/2 Model CL51 SX (8554)
- IBM Personal System/2 Model 55 SX (8555)
- IBM Personal System/2 Model 56 SX (8556)
- IBM Personal System/2 Model 57 SX (8557)
- IBM Personal System/2 Model 57 SLC (8557)
- IBM Personal System/2 Model 65 SX (8565)
- IBM Personal System/2 Model 70 386 (8570)

- IBM Personal System/2 Model 70 386 (8570) with the IBM Personal System/2 Power Platform*
- IBM Personal System/2 Model 70 486 (8570)
- IBM Personal System/2 Model P70 386 (8573)
- IBM Personal System/2 Model P75 486 (8575)
- IBM Personal System/2 Model 80 386 (8580)
- IBM Personal System/2 Model 90 XP 486 (8590)
- IBM Personal System/2 Model 95 XP 486 (8595).

Exception

The IBM PS/2 N33 SX Model A13, which was available only in certain countries, does not support OS/2 V2.0.

OS/2 Version 2.0 will *not* run on machines equipped with an Intel 80286 processor. Hence machines such as the IBM PC AT*, PS/2 Model 30-286, and Models 50, 50Z, and 60 may not be used to run OS/2 Version 2.0. However, OS/2 V2.0 does support 80286-based machines which have been upgraded with a 386 or 486 processor using cards such as the AOX MicroMaster** card.

The use of OS/2 Version 2.0 is not limited to IBM hardware; the operating system will execute on other manufacturers' machines, provided they are sufficiently compatible and equipped with an appropriate processor and hardware configuration. For information on hardware compatibility with OS/2 Version 2.0, readers should consult their dealer, hardware supplier or IBM marketing representative. Information is also available on CompuServe** and IBM bulletin board systems. Documentation shipped with OS/2 V2.0 contains details of gaining access to these systems.

Note

This new 32-bit OS/2 Version 2.0 uses advanced features of PC systems that were not exploited with the earlier 16-bit operating systems. In order to avoid any problems with installing or running OS/2 V2.0, it is advisable to update the PC system configuration to the latest release levels. Depending on the manufacturer of the system, this may involve different steps from running the system configuration software through to replacing the BIOS. Users of IBM PS/2s should obtain *the latest* version of their PS/2 reference diskette from IBM. Running automatic system configuration with this reference diskette will ensure the system is updated to the new level. This particularly applies to PS/2s such as the Models 90 and 95 which load their BIOS from disk.

5.1.2 Memory Requirements

The OS/2 Version 2.0 installation procedure examines the hardware of the machine to determine the amount of installed memory. Since the recommended minimum memory configuration is 4MB, the installation procedure will indicate an error if less than this amount of memory is present at installation time.

Although 4MB is the minimum memory to start OS/2 V2.0, better performance will be achieved with 6MB or more. This is especially true if the workplace scenario involves running many large or complex applications, or multiple DOS or

Windows applications. Some examples of typical user scenarios can be found in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell.*

5.1.3 Fixed Disk Requirements

When all installable options are selected, OS/2 Version 2.0 requires 30MB of fixed disk space in which to install itself. Approximately 26MB is used by the operating system files and 4MB used in the initial SWAPPER.DAT file. This usage may be substantially reduced if some of the installable options are omitted. Users should ensure that the required amount of fixed disk space is available in their system.

If partitioning the fixed disk for multiple partitions, users should ensure that the partition into which OS/2 Version 2.0 is to be installed is of sufficient size to accommodate the operating system and required options. A recommended *minimum* partition size is 40MB.

Don't forget to plan for the appropriate amount of fixed disk space to hold the SWAPPER.DAT file. This file will grow rapidly when overcommitting the system resources (memory). Also, the \SPOOL subdirectory holds print jobs that have been spooled and are waiting to be printed. This subdirectory can quickly use up available disk space if large or complex print jobs are sent. The default path for the spool path is \SPOOL on the drive the operating system is installed on. The spool path can be changed by opening *Spooler* settings by selecting in turn *OS/2 System* from the Desktop, *System Setup* and *Spooler*.

Partitioning fixed disks for support of OS/2 Version 2.0 is also discussed in Chapter 7, "Boot Manager."

5.2 Beginning Installation

Upon commencing the installation of OS/2 Version 2.0, using the installation diskette, the user is presented with a simple character-based screen which requests the user to insert the initial operating system diskette into the diskette drive. The first six diskettes (installation diskette and operating system diskettes 1 to 5) are used to prepare and partition the fixed disk, install Boot Manager and dual boot if required, and load the basic code required to drive the graphical installation procedure.

Once this code is transferred to the system's fixed disk, the installation procedure prompts the user to reboot the system. When the system restarts, the Presentation Manager shell is loaded and the graphical installation procedure is immediately invoked. Since the operating system has now been loaded off the fixed disk, paging is enabled, relieving constraints on real memory.

The graphical installation procedure supports both mouse and keyboard input. A generic mouse driver is provided by the graphical installation procedure, in order to support mouse input before the specific mouse driver is identified and loaded. This means that certain pointing devices might not be usable at this time and only keyboard input will be accepted.

The graphical installation procedure first presents the user with a single dialog box, centered on the screen. This dialog box enables the user to select one of several options:

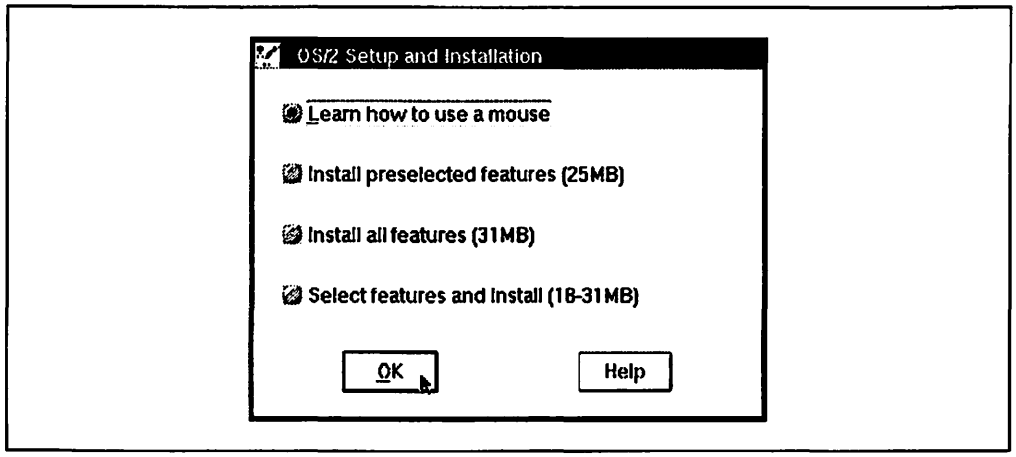


Figure 13. OS/2 Version 2.0 Installation

- Learn how to use a mouse.
- Install a basic set of preselected features.
- Install all features of OS/2 Version 2.0, accepting the default values for system options and parameters as assigned by the installation procedure.
- Select particular options and features to install, allowing modification of system parameters.

These options are displayed to the user as radio buttons. Since both mouse and keyboard support are fully enabled within the graphical installation procedure, the user may use either method to select an option. The user then selects the *OK* pushbutton to continue.

If the user elects to install all options, the installation procedure immediately requests that the user insert the next diskette into the diskette drive.

If the user wishes to select particular options to install, the installation procedure presents a further series of dialog boxes from which the user selects the required options and features. See 5.3, "Installation Options," below.

5.3 Installation Options

The user is presented with a panel which displays icons for mouse, keyboard, display, and country options, along with the default setting for each. These settings are determined by the installation procedure by interrogating the installed hardware in the system. The user may elect to alter one or more of these settings by selecting a check box next to each icon, and then selecting the *OK* pushbutton on the dialog box.

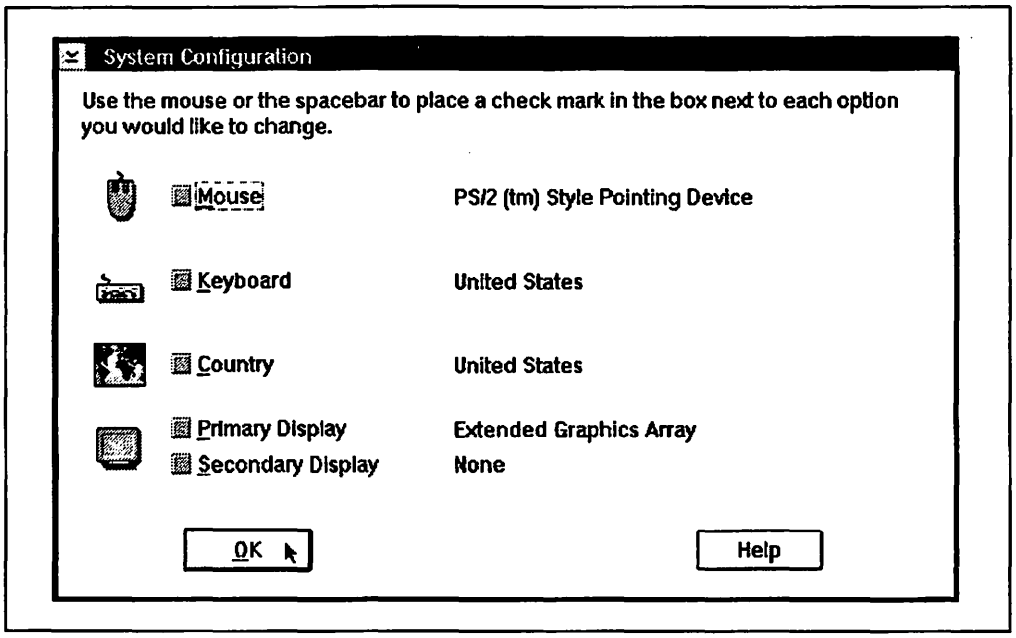


Figure 14. The Initial System Configuration Screen

For each selected option, the installation procedure then displays a dialog box which contains a list of the valid settings for that option. This list is displayed using a standard Presentation Manager list box. The user selects the required setting with the mouse or keyboard, and commits the choice by selecting the *OK* pushbutton.

The default options selected by the installation procedure are as follows:

Setting	**Default**
Mouse	The installation procedure automatically detects the type of mouse currently attached to the system (if any) and sets the default option accordingly. The user need only make alterations to the default if mouse support is required and no mouse is currently attached.
Keyboard	The installation procedure automatically detects the type of keyboard currently attached to the system and sets the default option accordingly.
Display	The installation procedure automatically detects the type of display adapter currently installed in the system, and sets the default option accordingly. The user need only make alterations if some nonstandard type of display option is required (for example, the user has an XGA adapter installed, but only wishes to use it in VGA emulation mode).
Country	Unless a National Language Support (NLS) version of OS/2 Version 2.0 is being installed, the country settings default to those for the United States, with appropriate time, date, and numeric representations, and use of primary codepage 437 and secondary codepage 850. Users who require codepages other than these should make the appropriate alterations.

5.3.1 Installing Optional Features

For optional installable features, the user is prompted to indicate which options are required. The name of each option is displayed, along with the amount of fixed disk space required by that option. Selections are made using check boxes, as shown in Figure 15.

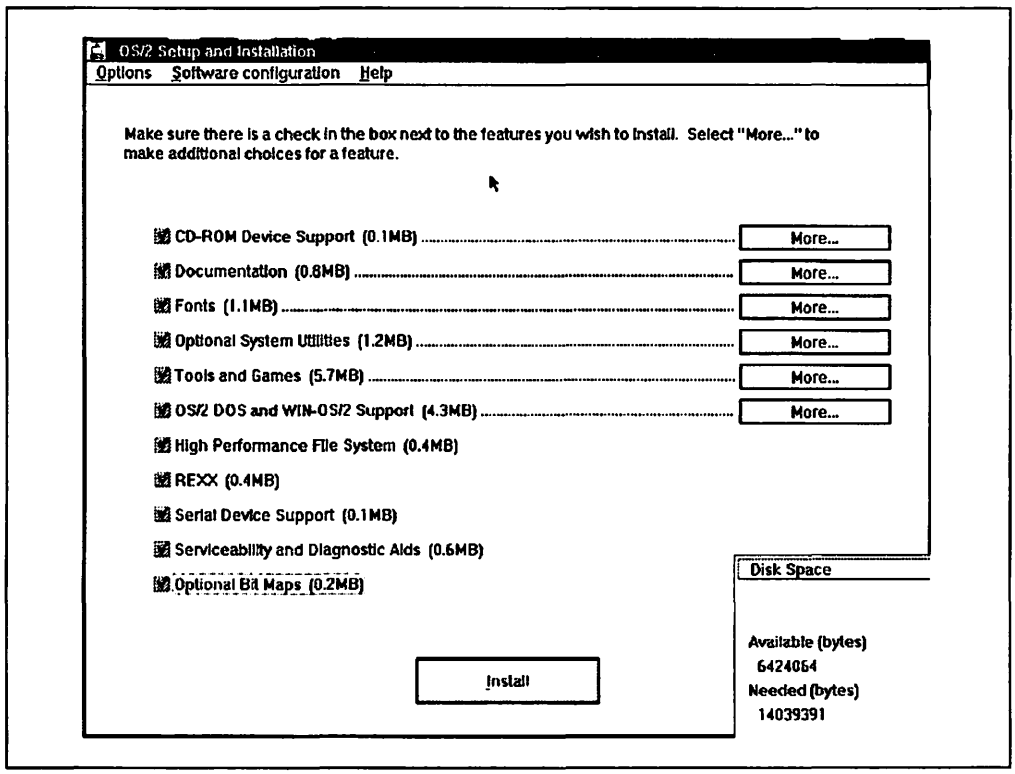


Figure 15. Selecting Features to Install

In certain cases, additional information may be specified to install a particular option; for example, the *Tools and Games* option represents a group of individually installable applications, of which the user may elect to install any or all. The presence of additional information is indicated using *More...* pushbuttons adjacent to each such option. Selecting one of these pushbuttons displays a dialog box which allows the user to configure the option.

When the user has selected all the required options to be installed, the installation is commenced by selecting the *OK* pushbutton on the dialog box.

5.3.2 Configuring System Parameters

The graphical installation procedure removes the need for a user to directly access and modify system configuration files such as CONFIG.SYS and AUTOEXEC.BAT during installation. Instead, the installation procedure provides a simple interface to allow the alteration of system parameters. This interface is invoked by the user selecting the *Software Configuration* menu item in the *OS/2 Setup and Installation* window. Selection of this item displays an additional window, as shown in Figure 16 on page 53.

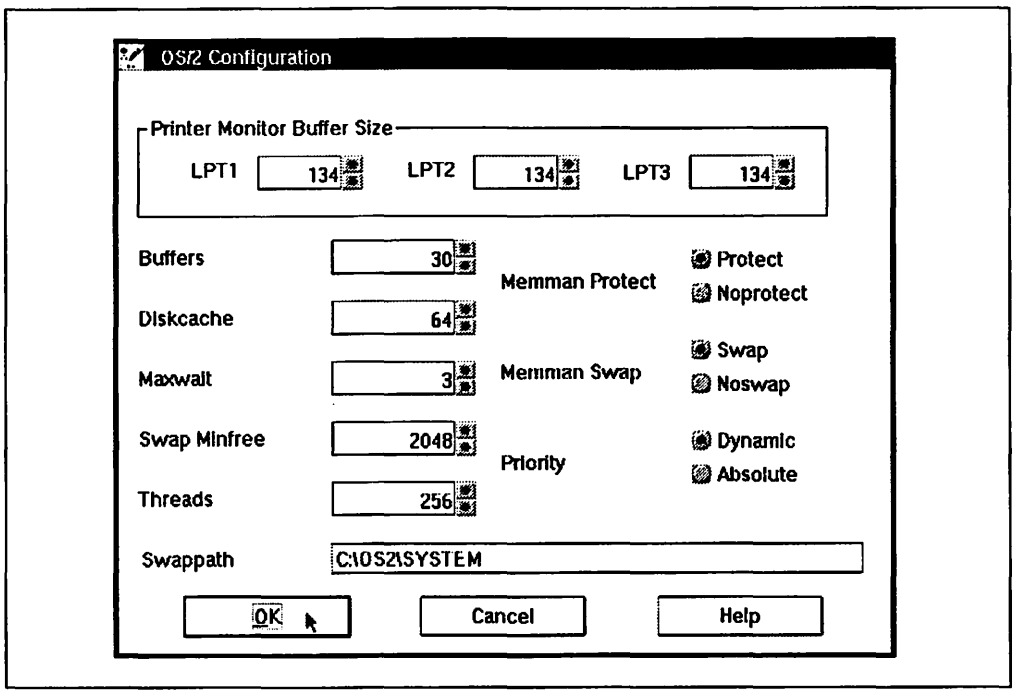


Figure 16. Configuration Details

Other parameters are added to CONFIG.SYS based upon the chosen settings for country, mouse type, etc.

5.4 Progress Indication

Once options are configured (or the user indicates that the default options are to be used), the user is prompted to insert the next diskette into the machine. A progress indicator is then displayed, which shows the name of the file currently being installed, and the percentage of data on the current diskette that has already been copied.

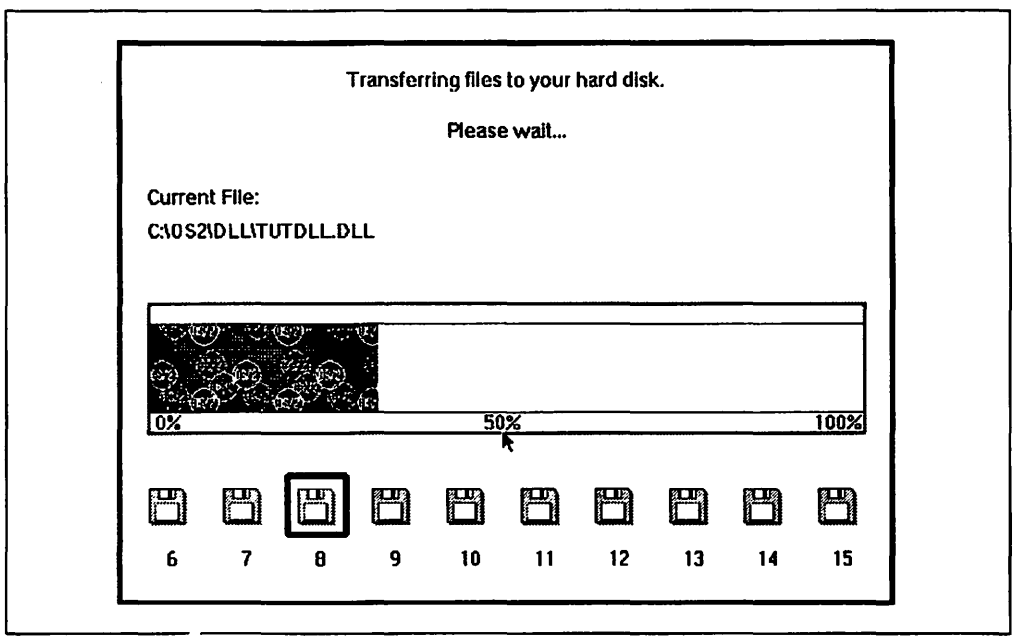


Figure 17. Installation Progress Indication

The progress through the installation is also displayed by the use of diskette icons at the bottom of the screen; the diskette currently being copied is displayed with a highlighted border around it.

5.5 After Installation

When installation is complete, the installation procedure instructs the user to reboot the system. OS/2 Version 2.0 is then loaded in the normal way.

Notes

1. Upon loading the operating system for the first time, the Workplace Shell must initialize itself and build its desktop. This initialization means that, for this first IPL after installation, the loading time will be significantly longer than normal.
2. After the system has been started for the *first* time, it is strongly recommended that the system be immediately shutdown, followed by a reboot.

   It is important that this shutdown be performed before any DOS or Windows applications are started.

When OS/2 V2.0 is first started after installation, startup time will be longer than normal. However OS/2 V2.0 will load the tutorial so that the user has the option to view the OS/2 Version 2.0 Tutorial while the operating system is initializing itself. This tutorial provides a basic introduction to OS/2 Version 2.0 and the graphical user interface, including use of the mouse and keyboard to interact with Presentation Manager.

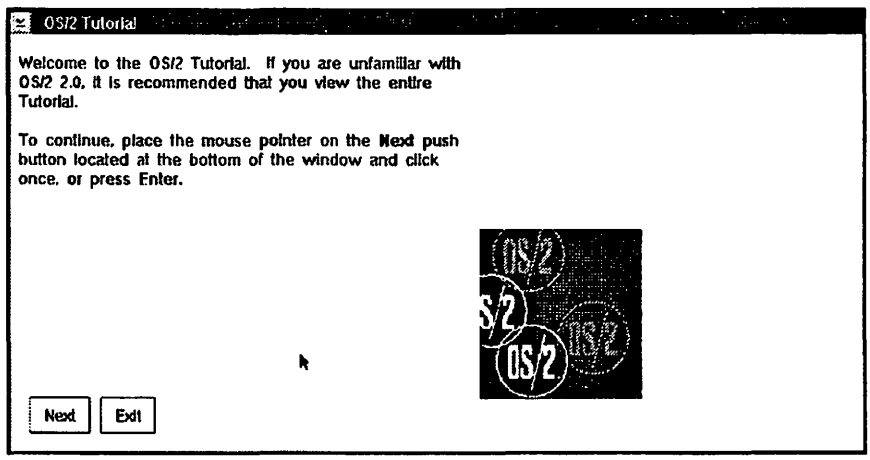


Figure 18. OS/2 Version 2.0 Tutorial

The tutorial provides very simple instructions to allow the user to navigate through the tutorial panels. Using the tutorial teaches the user about the various capabilities of OS/2 Version 2.0, while at the same time allowing the user to become familiar with the ways of interacting with a typical OS/2 Presentation Manager application.

5.6 Understanding the System Parameters

The installation of OS/2 Version 2.0 provides the user with a chance to tailor the OS/2 and DOS system parameters from the *OS/2 Setup and Installation* panel. During normal operation of OS/2 V2.0 there is little need to directly edit system files such as CONFIG.SYS. The installation of OS/2 generates a CONFIG.SYS file in the root directory of the installation drive, similar to that shown in Figure 19. Note that in this example the user has installed OS/2 V2.0 on the "D:" logical drive.

```
01  IFS=D:\OS2\HPFS.IFS  /CACHE:384 /CRECL:4 /AUTOCHECK:CDEFG
02  PROTSHELL=D:\OS2\PMSHELL.EXE
03  SET USER_INI=D:\OS2\OS2.INI
04  SET SYSTEM_INI=D:\OS2\OS2SYS.INI
05  SET OS2_SHELL=D:\OS2\CMD.EXE
06  SET AUTOSTART=PROGRAMS,TASKLIST,FOLDERS
07  SET RUNWORKPLACE=D:\OS2\PMSHELL.EXE
08  SET COMSPEC=D:\OS2\CMD.EXE
09  LIBPATH=.;D:\OS2\DLL;D:\OS2\MDOS;D:\;D:\OS2\APPS\DLL;
10  SET PATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;D:\OS2\MDOS;D:\OS2\APPS;
11  SET DPATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;D:\OS2\BITMAP;D:\OS2\MDOS;
       D:\OS2\APPS;
12  SET PROMPT=$i[$p]
13  SET HELP=D:\OS2\HELP;D:\OS2\HELP\TUTORIAL;
14  SET GLOSSARY=D:\OS2\HELP\GLOSS;
15  PRIORITY_DISK_IO=YES
16  FILES=20
17  DEVICE=D:\OS2\TESTCFG.SYS
18  DEVICE=D:\OS2\DOS.SYS
19  DEVICE=D:\OS2\PMDD.SYS
20  BUFFERS=30
21  IOPL=YES
22  DISKCACHE=64,LW
23  MAXWAIT=3
24  MEMMAN=SWAP,PROTECT
25  SWAPPATH=D:\OS2\SYSTEM 2048 4096
26  BREAK=OFF
27  THREADS=256
28  PRINTMONBUFSIZE=134,134,134
29  COUNTRY=001,D:\OS2\SYSTEM\COUNTRY.SYS
30  SET KEYS=ON
31  REM SET DELDIR=C:\DELETE,512;D:\DELETE,512;E:\DELETE,512;F:\DELETE,512;G:\DELETE,512;
32  BASEDEV=PRINT02.SYS
33  BASEDEV=IBM2FLPY.ADD
34  BASEDEV=IBM2SCSI.ADD /LED
35  BASEDEV=OS2SCSI.DMD
36  BASEDEV=OS2DASD.DMD
37  REM IFS=D:\OS2\CDFS.IFS /Q
38  REM DEVICE=D:\OS2\CDROM.SYS /Q /I /N:4
39  SET BOOKSHELF=D:\OS2\BOOK
40  SET EPATH=D:\OS2\APPS
41  DEVICE=D:\OS2\APPS\SASYNCDB.SYS
42  PROTECTONLY=NO
43  SHELL=D:\OS2\MDOS\COMMAND.COM D:\OS2\MDOS /P
44  FCBS=16,8
45  RMSIZE=640
46  DEVICE=D:\OS2\MDOS\VEMM.SYS
47  DEVICE=D:\OS2\MDOS\VMOUSE.SYS
48  DOS=LOW,NOUMB
49  DEVICE=D:\OS2\MDOS\VDPX.SYS
50  DEVICE=D:\OS2\MDOS\VXMS.SYS /UMB
51  DEVICE=D:\OS2\MDOS\VDPMI.SYS
52  DEVICE=D:\OS2\MDOS\VWIN.SYS
53  DEVICE=D:\OS2\MDOS\VCDROM.SYS
54  DEVINFO=SCR,VGA,D:\OS2\VIOTBL.DCP
55  SET VIDEO_DEVICES=VIO_VGA
56  SET VIO_VGA=DEVICE(BVHVGA)
57  DEVICE=D:\OS2\MDOS\VVGA.SYS
58  DEVICE=D:\OS2\MDOS\V8514A.SYS
59  DEVICE=D:\OS2\POINTDD.SYS
60  DEVICE=D:\OS2\MOUSE.SYS
61  DEVICE=D:\OS2\COM.SYS
62  DEVICE=D:\OS2\MDOS\VCOM.SYS
63  CODEPAGE=437,850
64  DEVINFO=KBD,US,D:\OS2\KEYBOARD.DCP
```

Figure 19. A Typical OS/2 Version 2.0 CONFIG.SYS

The CONFIG.SYS files for OS/2 V2.0 are similar to those in previous versions of OS/2 with a few exceptions. Here is a brief explanation of the function performed by each line in the CONFIG.SYS used as an example above. A full description of each of the statements used can be found in the *Online Command Reference*.

```
01  IFS=D:\OS2\HPFS.IFS  /CACHE:384 /CRECL:4 /AUTOCHECK:CDEFG
```

This line installs the High Performance File System (HPFS) driver. Installable File System (IFS) drivers load code to manage disks and other storage media with file systems other than FAT (File Allocation Table).

The /CACHE parameter specifies the amount of memory (KB) that the HPFS file system driver will use for file system disk caching. The cache for the FAT file system is handled by the DISKCACHE= statement, line 22 in this example. The amount of cache specified in the CONFIG.SYS can have a significant impact on performance. The OS/2 V2.0 installation process will select an amount of cache for the system based on the amount of installed memory, the disk size and the file system being used. The optimum amount of cache to use will depend on the factors just mentioned, and the mix of operations performed by the system. For example, an I/O intensive system, such as a server, may perform better with a larger cache.

The /CRECL parameter specifies the maximum record size (in multiples of 2KB) for caching. This value is also important for performance as it represents the maximum size an I/O can be, and still be read into the HPFS cache.

For more information on the parameters associated with the HPFS driver, refer to the *Online Command Reference*.

```
02  PROTSHELL=D:\OS2\PMSHELL.EXE
```

This line loads PMSHELL.EXE as the user interface program and OS/2 command processor. PROTSHELL replaces the default OS/2 command processor (CMD.EXE) with another command processor.

```
03  SET USER_INI=D:\OS2\OS2.INI
04  SET SYSTEM_INI=D:\OS2\OS2SYS.INI
05  SET OS2_SHELL=D:\OS2\CMD.EXE
06  SET AUTOSTART=PROGRAMS,TASKLIST,FOLDERS
07  SET RUNWORKPLACE=D:\OS2\PMSHELL.EXE
08  SET COMSPEC=D:\OS2\CMD.EXE
10  SET PATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;
      D:\OS2\MDOS;D:\OS2\APPS;
11  SET DPATH=D:\OS2;D:\OS2\SYSTEM;D:\OS2\MDOS\WINOS2;D:\OS2\INSTALL;D:\;
      D:\OS2\BITMAP;D:\OS2\MDOS;D:\OS2\APPS;
12  SET PROMPT=$i[$p]
13  SET HELP=D:\OS2\HELP;D:\OS2\HELP\TUTORIAL;
14  SET GLOSSARY=D:\OS2\HELP\GLOSS;
```

These lines set OS/2 V2.0 *system variables* stored in the OS/2 V2.0 *environment*. The environment is a special place in storage used by the operating system and applications to store and look up values of variables. DOS sessions and OS/2 sessions operate independently; therefore each command processor that starts a session can have its own environment. For information on how command processors

within a session inherit the environment, refer to the *Online Command Reference*.

The main change from OS/2 V1.3 is the inclusion of some variables used by the Workplace Shell, namely lines 3 though 7.

```
09  LIBPATH=.;D:\OS2\DLL;D:\OS2\MDOS;D:\;D:\OS2\APPS\DLL;
```

This sets the LIBPATH variable. It is used to identify a set of directories to be searched when OS/2 loads dynamic link libraries. LIBPATH is not a part of the environment and therefore cannot be viewed with the SET command. Also, unlike the PATH environment variable, the current directory is not searched first. The entry " .; " at the beginning of the LIBPATH statement is used to force OS/2 V2.0 to search the current directory.

```
15  PRIORITY_DISK_IO=YES
```

Specifies disk input/output priority for applications running in the foreground. When PRIORITY_DISK_IO=YES is specified, applications running in the foreground will receive disk I/O priority over applications running in the background.

```
16  FILES=20
```

Determines the maximum number of files available in DOS sessions. Regardless of the FILES= setting, all DOS programs are initialized to a maximum of 20 files. It is the responsibility of an application to increase the number of files *up to the maximum* set by the FILES= statement. Each DOS session can also be customized by changing the appropriate DOS setting. This statement has no effect in OS/2 sessions.

```
17  DEVICE=D:\OS2\TESTCFG.SYS
18  DEVICE=D:\OS2\DOS.SYS
19  DEVICE=D:\OS2\PMDD.SYS
```

Install device drivers using the DEVICE= statement. Both DOS and OS/2 device drivers can be loaded with this statement in the CONFIG.SYS. OS/2 V2.0 device drivers are initialized when you start OS/2 and can process requests from either DOS or OS/2 programs. DOS device drivers are also initialized when you start the OS/2 operating system but they can only process requests from DOS programs.

TESTCFG.SYS is a new device driver in OS/2 V2.0 which is used during the install process to test the system configuration. This DEVICE statement should not be removed from the CONFIG.SYS as it is also used by the selective install process and during device driver installation.

```
20  BUFFERS=30
```

Sets the number of disk buffers the system will keep in memory. Each buffer uses 512 bytes of available memory. If you run many programs in OS/2 sessions, you can increase the speed of your system by increasing the value specified for BUFFERS (for example, BUFFERS=70). However, remember that when you increase the number of disk buffers, you decrease the available memory by 512 bytes for each buffer specified. In memory-constrained systems (4MB) reduce the number of buffers to 20.

21 IOPL=YES

Allows I/O privilege to be granted to requesting processes in OS/2 sessions.

22 DISKCACHE=64,LW

Specifies the number of blocks of storage (KB) allocated to the FAT file system cache. This parameter can have a marked effect on performance. See the comments on line 01 of this example CONFIG.SYS and also the *Online Command Reference* for more information on the parameters associated with DISKCACHE.

23 MAXWAIT=3

Sets the length of time, in seconds, a process waits before the system assigns it a higher priority. The most appropriate length of time to set MAXWAIT to depends on the number of applications that must run concurrently and the kinds of activities the applications perform. The default is 3 seconds.

24 MEMMAN=SWAP,PROTECT

This line specifies the various memory management options for the OS/2 V2.0 environment. SWAP enables paging whereas NOSWAP disables paging. PROTECT enables the use of protected memory by DLLs. There is also the MOVE/NOMOVE parameter which has no effect under OS/2 V2.0 and is provided for compatibility with OS/2 V1.3 only.

25 SWAPPATH=D:\OS2\SYSTEM 2048 4096

The swap file (SWAPPER.DAT) is used to temporarily store pages that the system has removed from physical memory in order to free up space to satisfy page-in requests generated as a result of page faults. If paging is enabled, this command specifies the location and initial size of the swap file. The first parameter specifies the amount of disk space (in KB) at which the system will begin to warn you that there is less than this amount of space left on the partition containing your SWAPPER.DAT file. The second parameter specifies the size of the swapper file (in KB) initially allocated at the time OS/2 is started.

26 BREAK=OFF

The Ctrl-Break keys sequence will stop a command from completing its task. The BREAK parameter instructs the system to check if you pressed Ctrl-Break before the system carries out a program request. BREAK=ON could decrease overall performance, but means the operating system will probably intercept Ctrl-Break faster.

27 THREADS=256

This specifies the maximum number of threads available to OS/2 V2.0 and its applications. The maximum that can be specified here is 4095. Normally the system default of 256 is sufficient. In memory-constrained systems (4MB) reduce this to 128.

28 PRINTMONBUFSIZE=134,134,134

This sets the parallel-port device-driver buffer size. Each number corresponds to the buffer size for LPT1, LPT2, or LPT3 respectively. The minimum value that can be specified is 134 bytes and the maximum is 2048 bytes. The system will default to 134 bytes if PRINTMONBUFSIZE is not specified or is out of this range. Changing these values will allow you to increase the size of the parallel-port device-driver buffers and thereby increase performance of data transfer to devices connected to the parallel ports.

29 `COUNTRY=001,D:\OS2\SYSTEM\COUNTRY.SYS`

This specifies the country code and the file containing the country information used. This information is selected by the user at installation. Refer to the *Online Command Reference* for more details and information on code-page switching.

30 `SET KEYS=ON`

This permits commands issued at the OS/2 command prompt to be retrieved later and reissued and/or edited.

31 `REM SET DELDIR=C:\DELETE,512;D:\DELETE,512;E:\DELETE,512;F:\DELETE,512;`
`G:\DELETE,512;`

The DELDIR environment variable is new with OS/2 V2.0 and is used by the UNDELETE command. DELDIR specifies a path used to store files that have been DELETED or ERASED. A separate directory and maximum directory size must be specified for each logical disk. Installation will add an appropriate DELDIR statement to the CONFIG.SYS but remarks it out. To enable UNDELETE, remove the 'REM' from the beginning of this line.

32 `BASEDEV=PRINT02.SYS`
33 `BASEDEV=IBM2FLPY.ADD`
34 `BASEDEV=IBM2SCSI.ADD /LED`
35 `BASEDEV=OS2SCSI.DMD`
36 `BASEDEV=OS2DASD.DMD`

These lines install base device drivers for the following:
PRINT02.SYS; Device support for locally attached printers on Micro Channel workstations.
IBM2FLPY.ADD; Device support for diskette drives on Micro Channel workstations.
IBM2SCSI.ADD; Device support for Micro Channel SCSI adapters. The /LED parameter is only applicable when running on a PS/2 Model 95 and if present, the device driver will use the system information display panel to simulate a disk activity light.
OS2SCSI.DMD; General-purpose device support for non-disk SCSI devices.
OS2DASD.DMD; General-purpose device support for disk drives.

In addition, OS/2 V2.0 includes the following base device drivers:
PRINT01.SYS; Device support for locally attached printers on non-Micro Channel workstations.
IBM1FLPY.ADD; Device support for diskette drives on non-Micro Channel workstations.
IBM1S506.ADD; Device support for non-SCSI disk drives on non-Micro Channel workstations.

IBM2ADSK.ADD; Device support for non-SCSI disk drives on Micro Channel workstations.
IBMINT13.I13; General-purpose device support for non-Micro Channel SCSI adapters.

A device driver is a file that contains the code that the OS/2 operating system needs to recognize a device and correctly process information received from or sent to that device. A base device driver is one that is needed when the OS/2 operating system is first started.

The BASEDEV statement is used to load base device drivers. Device support for disks, diskettes, printers connected to the workstation, and other devices, is loaded with the BASEDEV statement.

Unlike the DEVICE statement, the BASEDEV statement cannot contain either drive or path information because the OS/2 operating system cannot process such information at the stage of the startup sequence when the BASEDEV statements are processed. The root directory of the startup partition is first searched for the specified file name, then the \OS2 directory of the startup partition. If drive or path information is included in a BASEDEV statement, an error is generated.

In addition, BASEDEV statements are not necessarily processed in the order in which they appear in your CONFIG.SYS file. The extensions of the file names specified in the BASEDEV statements are examined; the statements are then processed in the following order of file name extensions: .SYS .BID .VSD .TSD .ADD .I13 .FLT then .DMD

```
37  REM IFS=D:\OS2\CDFS.IFS /Q
38  REM DEVICE=D:\OS2\CDROM.SYS /Q /I /N:4
```

These lines contain the CD-ROM file system IFS driver and the IBM CD-ROM device driver for OS/2 V2.0 If CD-ROM device support was selected at the OS/2 Setup and Installation screen during installation, then these lines will be present. Refer to the *Online Command Reference* for an explanation of the parameters used.

```
39  SET BOOKSHELF=D:\OS2\BOOK
40  SET EPATH=D:\OS2\APPS
```

Lines 39 and 40 are two more environment variables used by OS/2 V2.0 or applications. For example, the BOOKSHELF environment variable points to the path of the system .INF files. These files are used by the system VIEW.EXE.

```
41  DEVICE=D:\OS2\APPS\SASYNCDB.SYS
```

The SASYNCDx.SYS device driver loads the asynchronous communications device interface (ACDI) support for the PM terminal program. If the PM Terminal program was not installed, this line may be REMarked out. If this is the case, delete "REM" from the beginning of the line to enable ACDI support. Remember, after changes are made to the CONFIG.SYS the system needs to be rebooted for the changes to come into effect. Refer to the *Online Command Reference* entries for Device Drivers (COM.SYS) for an explanation of how device drivers such as SASYNCH.SYS and COM.SYS are used.

```
42  PROTECTONLY=NO
```

This line allows OS/2 to run both DOS and OS/2 processes. If only OS/2 applications are required, specify PROTECTONLY=YES.

```
43  SHELL=D:\OS2\MDOS\COMMAND.COM D:\OS2\MDOS /P
```

The SHELL= line specifies the DOS command processor (COMMAND.COM), or allows you to replace the DOS command processor with another command processor. In this example we are loading and starting the DOS COMMAND.COM processor. The parameters following it are specific to the command processor. In this case the parameters are the path for COMMAND.COM and /P, which is used to retain COMMAND.COM in storage. The SHELL= statement does not affect either the OS/2 SET command, or the SHELL command in BASIC.

```
44  FCBS=16,8
45  RMSIZE=640
46  DEVICE=D:\OS2\MDOS\VEMM.SYS
47  DEVICE=D:\OS2\MDOS\VMOUSE.SYS
48  DOS=LOW,NOUMB
49  DEVICE=D:\OS2\MDOS\VDPX.SYS
50  DEVICE=D:\OS2\MDOS\VXMS.SYS /UMB
51  DEVICE=D:\OS2\MDOS\VDPMI.SYS
52  DEVICE=D:\OS2\MDOS\VWIN.SYS
53  DEVICE=D:\OS2\MDOS\VCDROM.SYS
```

Lines 44 through 53 are needed to configure and load drivers for the DOS environment.

The first line sets the file control block (FCB) management information for DOS sessions. A FCB is a record that contains all of the information about a file (for example, its structure, length, and name). If a program tries to open more than the number of files specified in the FCBS statement, the system closes the least recently used file control block and opens the new file. This parameter should not need to be changed and has no effect on OS/2 sessions.

The RMSIZE= statement specifies the highest storage address allowed for the DOS operating environment. In certain circumstances this can be used to limit the size of the DOS environment.

In line 48, the DOS= parameter specifies whether the DOS kernel will reside in the high memory area (HMA) and whether the operating system or DOS applications will control upper memory blocks (UMBs). The HMA refers to space between 1MB and 1MB+64KB and UMBs reside between 640KB and 1MB.

If DOS=HIGH/LOW,UMB is specified in CONFIG.SYS, then the operating system controls the UMBs. This means that DOS applications can be loaded into upper memory but cannot allocate UMBs.

On the other hand, if DOS=HIGH/LOW,NOUMB is specified in CONFIG.SYS, then the operating system will not control any UMBs and DOS applications can allocate UMBs but cannot be loaded there.

Lines 46 and 47, and 49 through 53 contain the device drivers for the virtual DOS/Windows environment. More information on these specific drivers and a complete discussion of tailoring the DOS environment under OS/2 V2.0 may be found in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

```
54  DEVINFO=SCR,VGA,D:\OS2\VIOTBL.DCP
55  SET VIDEO_DEVICES=VIO_VGA
56  SET VIO_VGA=DEVICE(BVHVGA)
57  DEVICE=D:\OS2\MDOS\VVGA.SYS
58  DEVICE=D:\OS2\MDOS\V8514A.SYS
```

These lines configure the display environment under OS/2 V2.0

The DEVINFO= statement in line 54 is used to prepare a device (such as a keyboard (KBD), display terminal (SCR), or printer (LPT#)) for codepage switching. Separate DEVINFO statements are required for each device to be used for codepage switching.

This line prepares the display screen for codepage switching. The display statement specifies your display name and a file named VIOTBL.DCP that contains a video font table for displaying characters in each of the codepages supported by the system.

Lines 55 and 56 set environment variables related to the display driver being used.

Lines 57 and 58 load virtual device drivers for the VGA and 8514/A devices, for use by the DOS/Windows environment. In this example, the user specified VGA resolution with an 8514/A adapter installed.

If OS/2 V2.0 were installed on a system with an XGA adapter, and high resolution was specified for the Windows environment (that is, Windows was to be run full-screen only), then the appropriate lines in the CONFIG.SYS would be:

```
DEVINFO=SCR,VGA,D:\OS2\VIOTBL.DCP
SET VIDEO_DEVICES=VIO_XGA
SET VIO_XGA=DEVICE(BVHVGA,BVHXGA)
DEVICE=D:\OS2\MDOS\VVGA.SYS
DEVICE=D:\OS2\MDOS\VXGA.SYS
```

Adding or changing your display adapter support after installation can be performed automatically using the OS/2 V2.0 *selective install* process. A complete description of how this is done can be found in the *Master Help Index* under *adding display adapter support*.

```
59  DEVICE=D:\OS2\POINTDD.SYS
60  DEVICE=D:\OS2\MOUSE.SYS
```

POINTDD.SYS provides mouse-pointer draw support in all text modes for OS/2 sessions. To function, an appropriate mouse device driver must also be loaded. In this example, this is done in line 60 with the MOUSE.SYS driver. These two lines will provide support for a PS/2-style mouse.

If the pointing device used was a Microsoft mouse attached to the COM1 port, line 60 would change to:

```
60    DEVICE=D:\OS2\MOUSE.SYS SERIAL=COM1
```

Some pointing devices such as the Logitech** mouse need a device-dependent device driver as well as the MOUSE.SYS device-independent device driver.

Refer to 6.3.6, "Pointing Device Support" on page 81 and the *Online Command Reference*, if installing support for a pointing device not directly supported by the OS/2 installation procedure.

```
61  DEVICE=D:\OS2\COM.SYS
62  DEVICE=D:\OS2\MDOS\VCOM.SYS
```

The COM.SYS device driver supports ports COM1, COM2, COM3, and COM4. It does not provide support for specific devices that are attached to the COM port. Instead, it provides enabling support for the asynchronous communications interface itself. Application programs, subsystems, and system programs must provide the support needed to use devices attached to the COM port. VCOM.SYS is a virtual device driver that provides support in DOS sessions for up to four COM ports.

Note

The COM.SYS device driver is used for all IBM PS/2 models. There is *no* COMDMA.SYS device driver for the IBM PS/2 Models 90 and 95 as described in the *Online Command Reference*.

Some devices which attach to the COM ports provide their own device drivers. COM.SYS allows other drivers which support COM ports to be installed, provided they are listed in the CONFIG.SYS before COM.SYS. When COM.SYS is loaded, it will claim all COM ports not already allocated to other device drivers.

In the following example DEVCOM1.SYS uses COM1 and PRINTER4.SYS uses COM4. COM2 and COM3 are available for use by COM.SYS.

```
DEVICE=C:\OS2\DEVCOM1.SYS
DEVICE=C:\OS2\PRINTER4.SYS
DEVICE=C:\OS2\COM.SYS
```

```
63  CODEPAGE=437,850
64  DEVINFO=KBD,US,D:\OS2\KEYBOARD.DCP
```

Line 63 sets the system codepages (defined character sets) to be prepared by the OS/2 operating system for codepage switching. Line 64 is DEVINFO specification for the keyboard. It specifies the keyboard layout and a file named KEYBOARD.DCP that contains a keyboard layout table for translating keystrokes into the characters of each codepage supported by the system. Codepage support in OS/2 V2.0 is similar to that provided under previous versions of OS/2. More information on DEVINFO and CODEPAGE can be found in the *Online Command Reference*.

5.7 Starting Programs Automatically

While the system is starting, programs will be started as a result of:

Inclusion of commands in STARTUP.CMD

This command file is used exactly as it was in previous releases of OS/2. During installation of IBM OS/2 LAN Server components, a STARTUP.CMD file is optionally built, which automatically starts the LAN Server components that were installed. STARTUP.CMD is useful for running programs where:

- The order in which programs run is important; one program must complete before the next one is started
- A program must be run only if the previous program ran successfully.

Program objects in the *Startup* folder

The *Startup* folder contains program objects which are to be run when the Workplace Shell is started. An example is LOGON.EXE which performs a logon to the server. It should not be used to start programs which are to run continuously. The *Startup* folder is found in the *OS/2 System* folder which appears on the OS/2 desktop. For a detailed discussion of folders and objects see *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*.

Programs not closed before the last system shutdown

All programs that were started and were still running when the system was last shutdown will be restarted when the system is next started. This is the preferred method for starting those programs which will run continuously. An example is the Communications Manager when it is used to provide host terminal emulation.

It possible to suppress automatically starting programs from the Startup folder and programs that were not closed. During startup of the Workplace Shell after rebooting the operating system, press and hold the Ctrl, left Shift and F1 keys when the white screen first appears and hold them down till the icons appear on the screen.

5.8 Selective Install

The graphical installation procedure provides for the reinstallation of particular operating system features, and the subsequent installation of required features which have not been installed during the initial installation process. This is achieved by allowing the graphical installation procedure to run as a stand-alone process under Presentation Manager.

The graphical installation procedure may, therefore, be executed at any time by choosing *Selective Install* from the *System Setup* option in the *OS/2 System* folder.

5.9 Recovering the Desktop

During the installation of OS/2 V2.0, copies of certain system files (CONFIG.SYS, OS2.INI, and OS2SYS.INI) are placed in the \OS2\INSTALL subdirectory. If the system becomes corrupted and the desktop is no longer usable (after rebooting no icons appear on the desktop or the system TRAPs when starting), it is possible to use these copies to restore the system files to their initial state.

Warning

Using this function will restore the desktop to the same state it was in immediately after installation. Any customization done to the desktop after installation will be lost.

To perform this restore function, do the following;

1. Restart the computer.
2. Before the first OS/2 Logo panel appears, hold down Alt+F1 for 20 seconds.

When you perform the recovery function described above, the current versions of those files are automatically renamed and are replaced by the default installation versions. If you have a STARTUP.CMD file, that file is also renamed, so that it will not be executed during the next system startup.

5.10 Installation from a LAN

OS/2 Version 2.0 may be installed from another machine on a local area network. This method of installation is typically much faster than installation using diskettes. Remote installation over a LAN is described in detail in *OS/2 Version 2.0 Remote Installation and Maintenance*.

5.11 Installing over Existing Versions

OS/2 Version 2.0 is designed to be installed over OS/2 Version 1.2, OS/2 Version 1.3, OS/2 Version 2.0 Limited Availability (LA), DOS, or Windows 3.0. No special considerations are necessary when carrying out such an installation. However, if Boot Manager is to be used, the machine's fixed disk might need repartitioning to make space for the Boot Manager partition. See Chapter 7, "Boot Manager" for further information.

5.12 Summary

The process of installation has been greatly improved for OS/2 Version 2.0 over previous versions of OS/2. Once partitioning of the fixed disk and installation of the base operating system has been accomplished, the remainder of the installation procedure is carried out using a graphical installation procedure which runs under Presentation Manager. Full mouse and keyboard support is provided.

Progress indicators are given, allowing the user to easily determine the current point in the process. Optional features are installed by selecting them from a list of icons. Default settings for properties such as display and mouse type are determined by interrogating the hardware; these defaults may easily be altered by the user.

For those users who may not be familiar with use of the system, a tutorial is provided as part of the graphical installation procedure. This tutorial illustrates the use of windows, the keyboard and mouse, and drag-and-drop manipulation using icons.

Installation of specific optional features may be carried out after the operating system is installed, in an easier manner than with previous versions of OS/2. The graphical installation procedure is available from the Presentation Manager desktop or from the command line, and may be invoked to install only specific optional features.

Chapter 6. Hardware Considerations

This chapter describes the enhancements made to OS/2 Version 2.0 in the area of hardware support. We begin by focusing on the I/O-related components of the operating system which are collectively known as the **I/O supervisor**. Later in this chapter we investigate IBM and OEM hardware support. The changes in OS/2 Version 2.0 are specifically targeted to optimize disk device requests through improvements to the High Performance File System (HPFS) and FAT file system drivers, and to make use of the capabilities of the new SCSI adapters and disks. Figure 20 provides an overview of the I/O related components of OS/2 Version 2.0.

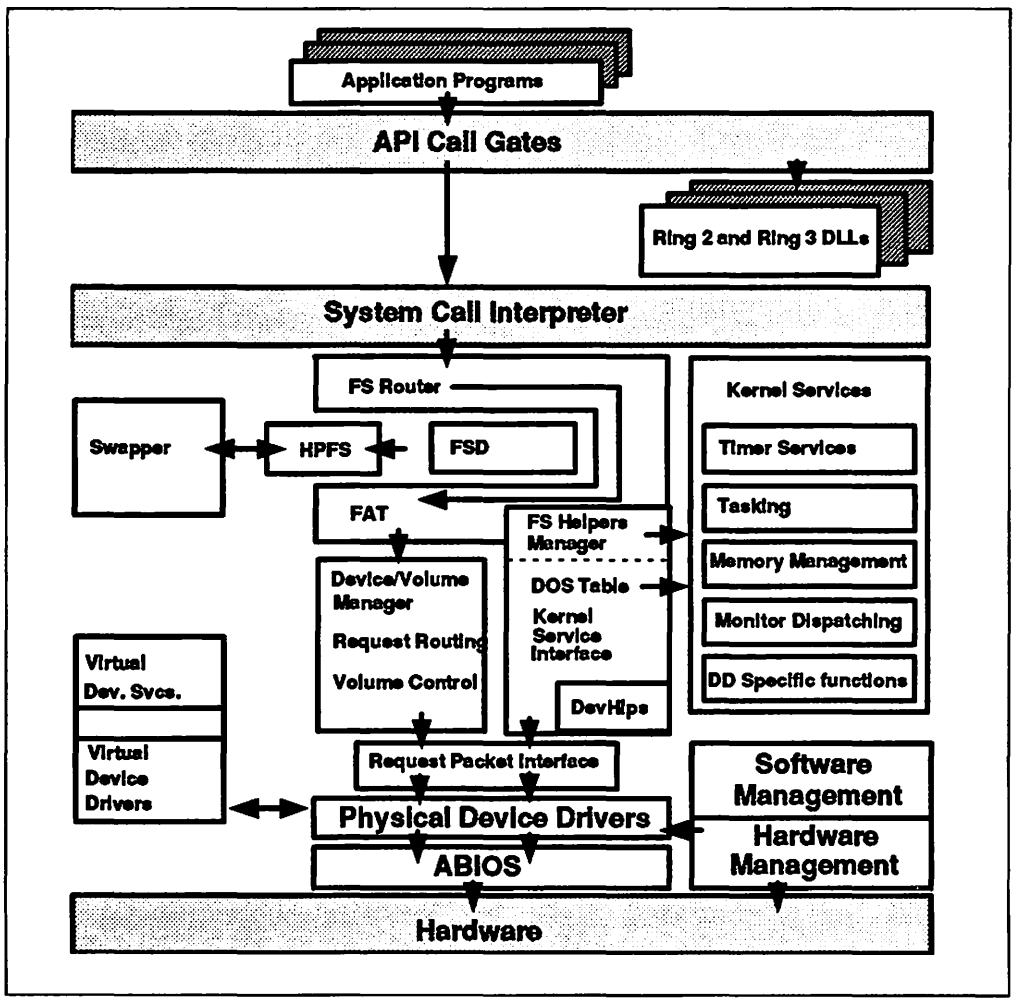


Figure 20. OS/2 Version 2.0 I/O Related Components

The enhancements in the I/O supervisor result in:

- Improved overall system I/O throughput capability by:
  - Using new protocol and data structures designed to optimize the submission of I/O requests
  - Allowing I/O command chaining by permitting file systems to submit a list of I/O operations to the device driver rather than submitting them one at a time
  - Providing a high-performance mechanism for transferring data to and from buffers for which pages are physically discontiguous in real storage

- Providing reductions in the overhead of interrupt processing and I/O request completion

- An architected and effective set of interfaces for an Installable File System (IFS) to allow paging I/O requests to be managed by a File System Driver (FSD).

The remainder of this chapter will discuss device driver and file system changes made under OS/2 Version 2.0, and IBM and OEM hardware support.

6.1 Device Driver Support

While OS/2 Version 2.0 is a 32-bit operating system, its device drivers and Device Helper (DevHlp) functions remain 16-bit. OS/2 V1.3 device drivers are generally compatible with OS/2 V2.0. New in OS/2 V2.0 is a layered device driver architecture. Also new are virtual device drivers for use by programs running in virtual DOS machine.

6.1.1 Compatibility with OS/2 V1.3

OS/2 Version 2.0 generally provides compatibility with OS/2 V1.3 device drivers written to the published OS/2 V1.3 device driver interface. Device driver incompatibilities with OS/2 V2.0 are summarized here:

- Drivers that directly modify or utilize the contents of the Global Descriptor Table (GDT)
- The PhysToVirt DevHlp returns different results to what is expected
- Drivers that set up their own GDT call gate will not work
- Drivers that switch to real mode will fail
- Device drivers that were written to support a single DOS session can fail if accessed by more than one DOS session concurrently.

Although OS/2 V2.0 device drivers remain 16-bit, the 32-bit architecture of OS/2 V2.0 can affect performance characteristics:

- OS/2 V2.0 provides 32-bit demand-paging virtual memory management based on Intel 386 while OS/2 V1.3 provides 16-bit segment-swapping virtual memory based on Intel 286

  Therefore, I/O requests are no longer guaranteed to occur in a single physically contiguous memory range. For compatibility, OS/2 V2.0 detects an OS/2 V1.3 device driver and breaks application I/O requests into multiple I/O requests to the OS/2 V1.3 driver at physical memory boundaries rather than a single I/O request as before in OS/2 V1.3.

- OS/2 V2.0 supports more than 16 MB of physical memory

  For compatibility, when a device driver utilizes an OS/2 V1.3 compatible request to lock memory for DMA transfers, OS/2 V2.0 rearranges memory so that the requested physical memory address range is below 16 MB.

6.1.2 Virtual Device Drivers

OS/2 Version 2.0 makes use of two distinct types of device drivers to communicate between the operating system and hardware devices:

- **Physical device drivers** communicate directly with hardware devices

  They operate in protected mode, and are accessed by protected mode processes and by virtual device drivers.

- **Virtual device drivers** (VDD) do not communicate directly with hardware devices; instead, they provide a virtual device driver interface for DOS applications running in virtual DOS machines

  Some VDDs have an associated physical device driver to which they pass requests. VDDs are 32-bit device drivers and make it possible for devices to be shared between processes running in virtual DOS machines and OS/2 screen groups.

See *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment* for a complete discussion of OS/2 Version 2.0 virtual device drivers.

6.1.3 Device Helper Functions

OS/2 Version 2.0 still uses a 16-bit device driver model. In order to optimize existing device drivers for the 32-bit flat memory model, new **DevHlp()** functions have been added for memory management.

- **Locking specific address ranges**

  Since paging is used in OS/2 Version 2.0, segments no longer need to be physically contiguous in memory. Thus, when locking these segments in memory, the memory manager may need to "shuffle" pages to make them contiguous.

  This shuffling can greatly increase the amount of time it takes to perform the locking. Locking of entire segments should therefore be avoided by using the new *VMLock* **DevHlp()** function, which locks *only* the required range of addresses within a memory object.

- **Address context conversion without locking**

  In previous versions of OS/2, device drivers were required to lock the user's buffers to gain context-free addressability. Since locking degrades performance and reduces available pageable system memory, new **DevHlp()** functions have been added to convert these addresses without the need for locking.

- **Removal of frequent selector loads**

  Device drivers can reduce the amount of selector loading by using the new flat memory model **DevHlp()** functions to allocate memory. All of the memory allocated by the new *VMAlloc* **DevHlp()** function can be addressed via one flat selector available to device drivers at initialization time.

- **Scatter/gather DMA support**

  Scatter/gather DMA adapters (such as the IBM SCSI adapters described in Appendix B, "Micro Channel Architecture and SCSI") allow DMA operations on physically discontiguous pages of memory. This ability provides a significant performance advantage over contiguous DMA operations.

  By using the new *VMLock* **DevHlp()** function, a device driver can specify that pages should be locked without being physically contiguous in memory.

VMLock will then return the physical addresses of each page that was locked.

- **Address-limited devices**

  Address-limited devices (using 24-bit addressing) do not support memory above 16MB. The device drivers for these products will use a new bit on the *VMLock* and *VMAlloc* calls, which specifies that the memory must be locked or allocated below the 16MB line.

The following are some of the new **DevHlp()** functions implemented in OS/2 Version 2.0:

- ***VMAlloc*** - Allocate block of physical memory

  VMAlloc is used to allocate linear and/or physical address space in memory. These allocations can exceed 64KB in size and can be either fixed, movable or swappable memory. This call can be used to map non-system memory into the current process context.

- ***VMFree*** - Free memory allocated via *VMAlloc*

  Memory allocated via *VMAlloc* can be freed with the *VMFree* **DevHlp()** function. *VMFree* is also used to remove the mappings created by *VMGlobalToProcess* and *VMProcessToGlobal*.

- ***VMLock*** - Locks a range of memory within a segment

  VMLock is used to lock a linear address range into physical memory. If the lock is needed for scatter/gather DMA, then a list of physical page addresses is returned.

- ***VMUnlock*** - Unlocks a range of memory within a segment

  VMUnlock is the counterpart of the *VMLock* function. It is used to unlock memory previously locked via *VMLock*.

- ***VMProcessToGlobal*** - Map process address into global address space

  The *VMProcessToGlobal* **DevHlp()** function is used to convert an address that is in the context of the current process to an address in a global context. This allows context-free addressability to the memory objects of a process.

- ***VMGlobalToProcess*** - Map global address into process address space

  VMGlobalToProcess can be used to map a global context address into the address space of the current process. When used for video buffers, the calling process can specify if the memory should be under screen group control. This will cause the memory to be validated or invalidated at task switch time.

- ***VirtToLin*** - Converts a selector:offset address to a linear address

  VirtToLin will convert a selector:offset address into a flat 32-bit linear address.

- ***LinToGDTSelector*** - Convert a linear address to a virtual address

  LinToGDTSelector is used to convert a linear address to a virtual (selector:offset) address by mapping the given GDT selector to the memory region referred to by the given linear address and range.

- ***PhysToGDTSel*** - Maps a physical address to a GDT selector

  PhysToGDTSel converts a 32-bit physical address to a GDT selector:offset pair.

- ***FreeGDTSelector*** - Free selector allocated via *AllocateGDTSelector*

  FreeGDTSelector frees up a GDT selector allocated via the *AllocGDTSelector* **DevHlp()** function.

- ***PageListToGDTSelector*** - Maps given physical addresses to selector

  PageListToGDTSelector is used to map physical addresses to a GDT selector, setting the access byte of the descriptor to the requested type. The virtual memory needed to map the physical ranges described by the page list array must not exceed 64KB.

- ***GetDescInfo*** - Return information on the contents of the descriptor

  GetDescInfo is used to return the access byte, linear address, and size of a descriptor allocated via the *AllocGDTSelector* **DevHlp()** function.

- ***PageListToLin*** - Maps physical pages to a linear address

  PageListToLin is used to map physical memory pages described in an array of page list structures to a linear address.

- ***LinToPageList*** - Returns the physical pages mapped by a linear range

  LinToPageList is used to translate a linear address range to an array of page list structures that describe the physical pages mapped.

6.1.4 New Disk Device Driver

A new disk device driver interface has been defined for use by OS/2 V2.0 file systems. The request-passing mechanism employs a request list of prioritized commands which the device driver may reorder to optimize access. Read and write operations use scatter/gather descriptors (as used by SCSI adapters) allowing data transfer to and from discontiguous memory buffers. The interface is designed to be data-structure compatible with future versions of OS/2 in order to minimize work at the time of migration.

With this extended device driver support, both the standard OS/2 interface (using ABIOS read/write) and the new high performance interface (using the ABIOS Transfer SCB) allow both OS/2 file systems and the OS/2 kernel to access the disk.

The following is a summary of the device driver changes:

- The disk device driver records the information indicating the type of DASD (ESDI/ST506 or SCSI) and the level of caching support for each logical drive (LID)
- Support is provided for the new device command **1Dh - GetDeviceSupport**. The device driver will return addresses to two structures in the request packet:

  1. The *Driver Capabilities Structure (DCS)* indicates the specific features that are supported by the device driver

     Examples are disk mirroring, disk duplexing and whether the device supports more than 16MB of memory.

  2. The *Volume Characteristics Structure (VCS)* for the device identified by the unit code in the device command

     Examples of characteristics reported in this structure are read-only, removable media, average seek time, outboard caching supported, SCB protocol supported and read prefetch supported.

- Support is provided for the new device command **1Ch - ExecuteChain**
- Support is provided for the new device commands **1Eh, 1Fh, and 20h - Read, Write and WriteVerify**
- Support is provided for the new device command **21h - Prefetch**
- I/O requests are placed on the device queue based on a priority passed in the request list

  To optimize the I/O operation, all requests are sorted in logical block number sequence. If a specific flag is on in the request list indicating that the execution order is critical, then the list will not be sorted.

  The possible priorities are:

Priority	***Meaning***
00h	Prefetch requests
01h	Low priority request - (lazy-write)
02h	Read ahead, low priority pager I/O
04h	Background synchronous user I/O
08h	Foreground synchronous user I/O
10h	High priority pager I/O
80h	Urgent request - used by kernel in critical situations.

- Calling threads with **ExecuteChain()** requests are no longer blocked by the device driver strategy routine

  The strategy routine instead returns control to the caller after the I/O is initiated or queued.
- Upon I/O completion, the device driver calls the notification routine specified in the request list.

6.1.5 Layered Device Driver Architecture

An OS/2 Version 2.0 DASD or SCSI device driver is no longer one large module as with previous versions of OS/2 but a layered device driver. The higher layer that interfaces with the OS/2 file systems and the OS/2 kernel is the Device Manager. The lower layer that interfaces with the device adapter is the adapter device driver (ADD). In addition, a filter ADD can exist in the middle between the Device Manager and the ADD. A filter ADD can be used to provide special services such as data compression or data encryption.

The new Adapter Device Driver interface for an ADD module defines direct call commands that are issued by the Device Manager or filter ADDs down to the registered entry point of an ADD.

It is now possible to add new device support to OS/2 V2.0 merely by writing the device support portion of the device driver. The higher level routines, required to support disk and SCSI devices, are only written once.

6.1.6 Base Device Drivers

In order to be more flexible in providing support for OEM machines, OS/2 Version 2.0 introduces a new class of device driver, a base device driver. A base device driver is one which is needed during the loading of the operating system. In order to identify and initiate the loading of base device drivers a new CONFIG.SYS statement **BASEDEV** is introduced. The **BASEDEV** statement is used to load support for disks, diskettes and printer devices.

Only the name of the base device driver is included on the **BASEDEV** statement as base device drivers are load during an early stage of OS/2 initialization and there is insufficient support available at this point to process a path. Base device drivers are installed in the root directory of the partition from which OS/2 is loaded or alternatively in the \OS2 directory in that partition.

The **BASEDEV** statement makes it possible to specify the device drivers required for system initialization. In previous releases of OS/2 this information was included as part of the system loader and this restricted the range of hardware that these releases could support.

6.2 File System Considerations

The following components are new to OS/2 Version 2.0 or contain changes from previous versions:

- HPFS device driver
- FAT file system device driver
- UNDELETE command
- Volume manager
- Pager (swapper).

6.2.1 High Performance File System Changes

The following changes have been made to the High Performance File System (HPFS) driver under OS/2 Version 2.0:

- At initialization time, the level of support provided by the device driver is determined using the new device command *1DH - GetDeviceSupport*.
- The HPFS driver passes physical addresses for data pointers, in the appropriate request format, to the device driver.
- The HPFS driver now supports command chaining, calling the volume manager with a list of all contiguous sector requests required to fulfill an I/O request.

  This function is supported for all DASD types.
- The HPFS driver supports scatter/gather by passing physical pointers to each page in the data buffer (physically discontiguous) as part of the I/O request.

  This allows I/O controllers such as the IBM SCSI adapters which support the scatter/gather capability to perform the I/O in a single operation.
- The HPFS driver now supports disk caching in the IFS driver, rather than in the device driver.

- The HPFS driver is able to recognize devices which have outboard caches (non-system memory), and incorporate them into the total caching scheme.

  The HPFS file system under OS/2 Version 2.0 supports a maximum file size of 2GB. The maximum size for an HPFS volume is 512GB.

6.2.2 FAT File System Changes

Changes have also been made to the FAT file system driver under OS/2 Version 2.0, in order to provide improved performance and enhanced support for disk hardware devices:

- The FAT driver now supports command chaining

  The driver attempts to call the volume manager with a list of all contiguous sector requests required to fulfill an I/O request, thus allowing multiple page-in and page-out requests in a single logical operation.

- The FAT driver provides faster allocation of free space on the logical drive, using a bitmap to track free clusters on the disk.

Disk caching is now supported within the FAT driver, and has been removed from the device driver. A cache buffer is provided to support disk caching with the following features:

- Lazy writing
- Lazy reading on writes, that is, the ability to write to the cache and flush the cache to disk, but then to read the updated information from the cache rather than requiring a physical disk read operation
- Asynchronous read-ahead through a multi-purpose asynchronous read thread
- Large cache size (theoretical maximum of 64MB, although practical limitations will necessitate a smaller cache)
- The ability to dynamically enable and disable the cache in response to a user command
- Bad sectors are automatically bypassed on reads.

There are a number of advantages in performing caching in the FAT driver rather than the device driver; more operating system kernel services are available at this level, and intelligent read-ahead operations can more easily be performed. Lazy writing is also more easily implemented at the file system level than at the device driver level.

The FAT file system under OS/2 Version 2.0 supports a maximum file size of 2GB. The maximum supported size for a FAT volume is also 2GB.

6.2.3 Disk Volume Considerations

The maximum size for a disk volume under OS/2 Version 2.0 is 512GB using HPFS and 2GB using the FAT file system, with the following conditions:

1. OS/2 V2.0 requires that the bootable partition be within the 1023rd cylinder of the disk

   When a machine's BIOS reports on the characteristics of a disk, it returns three values; it determines the number of sectors per head, the number of heads per cylinder, and the number of cylinders. The IBM BIOS sets the disk geometry such that one cylinder equals one megabyte of disk storage. For

IBM disk drives, 1023 cylinders corresponds to one gigabyte. Other manufacturers may use different sector and head values causing the 1023 cylinder limit to be greater than or less than one gigabyte.

2. A FAT file system volume must not exceed the 1023rd cylinder of the disk drive.

   It may also not exceed 2GB even if, due to the manufacturer's disk geometry, the 1023rd cylinder is beyond 2GB.

Imagine a 10GB disk where the geometry of the disk has sector/head/cylinder values such that the 1023rd cylinder corresponds to 7.4GB, as is the case with some OEM SCSI disks. Using the FAT file system, the way to configure the volumes to maximize their size would be to have three volumes of 2GB and one volume of 1.4GB. In this case the 2.6GB of free space (beyond the 1023rd cylinder) would be unusable by OS/2 using the FAT file system. These volumes may exist in primary partitions or an extended partition, or both. In an extended partition, the volume would occupy one or more of the logical partitions defined in the extended partition.

Although an HPFS volume can take up the entire disk, for performance reasons the practical limit is much less than 512GB.

There are 24 drive letters available for hard disk drives (a: and b: are reserved for diskette drives), and currently the largest tested IBM SCSI hard disk available for the PS/2 is 400MB. Using 24 single SCSI disks chained on four SCSI adapters we would get a maximum of 24 x 400MB, or 9.6GB of online DASD.

6.2.4 UNDELETE Command

OS/2 Version 2.0 provides a facility to delay the permanent removal of files that have been deleted. By default, files that are deleted or erased are stored (as hidden system files) in the \DELETE subdirectory. The UNDELETE command gives the user the ability to recover such deleted files and restore them to their original path.

A new environment variable DELDIR is provided to specify the path used to store deleted files. A separate directory and maximum directory size must be specified for each logical disk. The absence of a DELDIR environment variable in the CONFIG.SYS or a DELDIR statement with no path specified will disable the facility.

The function *DosForceDelete()* has been added to allow files to be deleted permanently without first making them recoverable.

6.2.5 Volume Manager

A number of changes have been made to the volume manager, in order to provide enhanced support for SCSI adapters, and to support the new SCB architecture request list data structures.

During the ABIOS initialization, one SCSI adapter-type logical ID (LID) will be generated for each SCSI adapter in the system. These LIDs cannot be allocated by the device driver, so in order to secure access to these LIDs, the device manager must include them in its list of LIDs.

The volume manager also supports fast path processing for the new request list data structures. These structures need no longer be transformed into request

packets. However, volume verification must still be performed by the volume manager. This function is required since the system must support removable media.

If the device driver written for previous versions of OS/2 is used with the OS/2 Version 2.0 volume manager, it will give an "invalid device command" return code to the GetDeviceSupport() command from the file system. In this case, the file system must be structured in a compatible form for the device driver. The volume manager then functions as before with no changes.

6.2.6 Pager (Swapper)

The swapper builds its I/O requests in a format similar to the ExecuteChain() format, using the SCB architecture and 32-bit physical page frame addresses. It then calls the file system to perform the paging operations.

When memory is overcommitted, page-in operations will occur in conjunction with page-outs which will free memory frames for the requested pages. The swapper may build request chains that contain both page-in and page-out operations, which results in a significant reduction in start I/O and interrupt processing overhead, thereby providing improved paging performance.

Normally the device driver has the possibility to sort I/O requests in order to optimize physical disk access. However, since a paging I/O request may contain both page-out and page-in instructions that address the same physical page in memory (as described above), the execution order of the individual operations may be critical.

In such circumstances, a flag indicating a critical I/O request is set by the swapper when issuing the request. The device driver will then not sort the chain, but will execute the operations in the exact sequence they were requested. The pager/swapper, therefore, optimizes I/O requests wherever possible, while maintaining the integrity of critical requests.

OS/2 Version 2.0 also contains a new interface (new calls) for those who program their own file systems. The new calls provide a clear interface between the file system and the FSD for paging. All calls are guaranteed to be used only for paging operations, so that the underlying FSDs may be optimized in supporting them. This guarantee implies that no argument checking, address verification, or size control takes place.

6.3 Hardware Support Levels

OS/2 Version 2.0 has been designed to run on both IBM and non-IBM (OEM) personal computers which incorporate processors from the Intel 80386 or 80486 families. The specific OEM models and their devices that are supported is a matter of testing and verification. IBM in conjunction with OEMs have tested OS/2 V2.0 on many OEM models and verified correct operation for those models tested. This process of verification is ongoing and more models are being added to the list of personal computers, on which OS/2 V2.0 will run, all the time. The architecture and design of OS/2 V2.0 supports OEM models in the following ways:

- All Machine Readable Information (MRI), text messages, online help, etc. is generic wherever possible. This generality means end user system text does not contain IBM or OEM specific references.

- Program code is generic wherever possible. This generality means it is not hard coded to recognize and handle only IBM specific systems or conditions.
- The device driver architecture has been enhanced to allow OEMs and Independent Hardware Vendors (IHVs) to more easily replace and add device support through the layered device driver architecture.

In this section we discuss the factors that determine whether OS/2 V2.0 will run on a particular hardware configuration. We also look at which of the hardware features of the system OS/2 V2.0 exploits.

A design point of OS/2 V2.0 is that it will enable support for a feature if it can do so without compromising the integrity of the system. This design point means that OS/2 V2.0 will only use a feature if it can do so reliably.

The first topic covered is the usage of large main memory, that is, systems with more than 16MB of RAM.

6.3.1 Large Main Memory Support

The Intel 80386 and 80486 families of microprocessors are capable of addressing 64TB of virtual memory. At present, OS/2 Version 2.0 has the capability of supporting up to 4GB of virtual memory and 4GB of real memory. In order to support more than 16MB of memory however, certain requirements must be met.

Firstly, the OS/2 V2.0 physical memory manager needs to know how much memory is present. This is done during kernel initialization by calling the BIOS interrupt 15 with subfunction code x'88'. This call will return the total amount of memory present. If the BIOS supports interrupt 15 with subfunction code x'C7', "return memory map information", then this function will be used to return an array giving the layout of the different levels of memory; for example, system board memory and adapter memory available in the machine, together with address ranges for the different memory types. Should the function not be present, the interrupt will return with the carry flag set. Support for this function is not mandatory. Information obtained is used by OS/2 V2.0 to prioritize the areas of physical memory that are allocated first.

The second requirement for supporting more than 16MB is that the paging device fixed disk controller can support data transfers to or from memory locations above the 16MB line. This ability usually requires the availability of a DMA controller with a 32-bit addressing capability, either on the system board or on a busmaster SCSI adapter. To determine whether this DMA is available, the disk device driver level is queried. If it is a level 3 device driver, it is further interrogated to find out if the DMA that the disk controller uses has support for 32-bit addressing. If this second requirement is satisfied and the disk controller can handle 32-bit addresses larger than 16MB, then all of memory is used by OS/2 for programs and data. If it cannot, then the OS/2 page manager allocates all of the memory above 16MB as a paging cache to be used before paging to the disk. This substitution is a good use of the memory above 16MB, but not as good as using it directly.

A final requirement for supporting more than 16MB is that the machine's ABIOS is capable of handling requests with addresses above the 16MB line.

Presently in the IBM PS/2 range, only the Models 90 and 95 have support for more than 16MB of memory and when using 8MB SIMMs will support up to 64MB. Machines using 80386SX, 80386SL or 80386SLC processors are not able

to support more than 16MB as they only support 24-bit addressing external to the processor.

If OS/2 V2.0 detects the presence of a device that does not support 32-bit addressing, and if support for more than 16MB is enabled, it will ensure that buffers used by that device's driver are locked below the 16MB line during I/O operations. Level 3 device drivers are then given a permanent buffer below the 16MB line. For level 1 and 2 device drivers the OS/2 V2.0 virtual memory manager, when requested to lock a user buffer for that device, will move the buffer below the 16MB line.

The only device for which 32-bit addressing support is mandatory is the paging device. Pages are written to or read from their exact physical location in memory and are not moved before I/O takes place. Should paging be disabled by specifying **MEMMAN=NOSWAP** in the **CONFIG.SYS**, then more than 16MB can be supported even though there is no disk subsystem support for it.

When determining whether an OEM machine will support more than 16MB, the considerations are:

- Is the BIOS/ABIOS in the machine capable of supporting the memory size?
- Is the BIOS capable of telling OS/2 that more than 16MB is present in the machine?
- If paging is enabled, does the paging device support 32-bit addressing?
- If the paging device does support 32-bit addressing, does its device driver have the capability of responding correctly to OS/2 V2.0 with this information?

6.3.2 Microprocessor Support

As with software, there are different releases of the Intel 80386 and 80486 family processors. The different releases of the processors are known as stepping levels. Later stepping levels fix bugs that were found in earlier stepping levels of the processor. OS/2 V2.0 will take into account the different stepping levels of processors and handle any nuances. An example would be when OS/2 V2.0 is running on an 80386 at B1 stepping level, with a numeric coprocessor present; OS/2 V2.0 will disable the coprocessor and provide emulation for the coprocessor, as it would do if the coprocessor were not present. This is one of the ways in which OS/2 V2.0 will circumvent the known bugs in a 80386 at B1 stepping level. Bugs such as these were fixed in the 80386SX and 80486 processors. In the case of the 80386 processor, a stepping level of D0 or higher is preferred.

Some early IBM PS/2s might have B1 stepping level 80386 processors on their system boards and might qualify for a free engineering change (EC) to upgrade the system board. If you experience problems running OS/2 on an older system, contact your dealer or the IBM service organization in your country to have your system unit checked.

6.3.3 Disk and SCSI Device Drivers

The layered device driver architecture has made it possible to easily support IBM and non-IBM disk and SCSI controllers. At the device management driver level, OS/2 V2.0 supplies two drivers:

OS2DASD.DMD General-purpose support for disk drives

OS2SCSI.DMD General-purpose support for non-disk SCSI devices.

OS2DASD.DMD will always be loaded. OS2SCSI.DMD is only needed when a SCSI adapter is present.

OS/2 V2.0 provides a number of adapter device drivers (ADDs) which are specific to the adapter or adapters installed in the system. At installation time, OS/2 V2.0 can in most cases sense the adapters installed and will then load the necessary ADDs and include the necessary BASEDEV statements in the CONFIG.SYS. ADDs are currently available for all IBM disk devices and some selected non-IBM adapters. See the *OS/2 V2.0 Online Command Reference* and the *README file*, which comes with OS/2 V2.0, for details of the supported adapters. The *README file* should always be consulted when using a non-IBM SCSI adapter.

At general availability of OS/2 V2.0, support was available for IBM, Future Domain** and Adaptec** SCSI adapters.

Shipped with OS/2 V2.0 is a generic disk device driver IBMINT13.I13. When specific support is not provided for a particular adapter, it should be possible to use this driver.

6.3.4 Video Display Support

At general availability OS/2 Version 2.0 provides the most complete support for video systems which provide VGA resolution. This is the only resolution supported for "seamless" WIN-OS/2 sessions at this time and if using "seamless" WIN-OS/2 on a particular system, the Presentation Manager desktop must be configured to use this resolution even if the display hardware is capable of higher resolution output.

OS/2 V2.0 provides support for devices with lower resolutions than VGA, namely CGA and EGA, for all types of sessions except "seamless" WIN-OS/2. Use of CGA for Presentation Manager is not recommended.

Devices with resolutions above VGA (640x480, 16 colors) are known as Super VGA (SVGA). The OS/2 installation process will detect display systems capable of running in SVGA mode. However, because of the wide variety of display systems currently available that fall into this category, full support is not provided.

Support is provided for 8514 and XGA display systems and drivers for Presentation Manager and WIN-OS/2 full-screen sessions will be installed during system installation. All other SVGA display systems will be supported in VGA mode. Systems with 8514 and XGA display hardware can also run in VGA mode.

Support however is provided to allow DOS full-screen and WIN-OS/2 full-screen sessions to use the display system in SVGA mode even though Presentation Manager is using it as a VG.\ device. A base video device handler (VDH) is pro-

vided which permits switching the display system between VGA and an SVGA mode. The list of video modes currently supported is:

- Graphic

  1. 640x480 256 colors
  2. 800x600 16 colors
  3. 800x256 256 colors
  4. 1024x768 16 colors
  5. 1024x768 256 colors

- Text

  1. 132x25 characters
  2. 132X43 characters
  3. 132X44 characters

SVGA support is enabled by:

- Including the SVGA base VDH, which is installed along with the VGA VDH, by changing the SET VIO_VGA statement in the CONFIG.SYS to:

  ```
  SET VIO_VGA=DEVICE(BVHVGA,BVHSVGA)
  ```

- By running SVGA.EXE

  From a DOS full-screen command prompt issue the command:

  ```
  SVGA ON
  ```

  Executing this program will result in a file called SVGADATA.PMI being created in the OS2 directory. This file contains the information about the video modes supported by the video adapter present in the system.

  This file also serves a flag to say that SVGA mode is enabled in the system. The SVGA support is only active once the SVGA program has been run. SVGA VDHs will run in VGA mode if the SVGADATA.PMI file is not present.

- Install the SVGA virtual VDH instead of the VGA VDH.

  This substitution provides support for DOS full-screen sessions to run in SVGA mode. The VVGA.SYS virtual VDH is replaced by VSVGA.SYS by changing the DEVICE statement in the CONFIG.SYS to read:

  ```
  DEVICE=x:\OS2\MDOS\VSVGA.SYS

  where x = the drive letter of the system partition
  ```

- Install a Windows V3.0 driver for the SVGA adapter in the system to provide support for WIN-OS/2 full-screen sessions.

  If a Windows V3.0 driver is available for the adapter, this should work under OS/2 V2.0. Update the "display.drv=" line in the SYSTEM.INI file in the \OS2\MDOS\WINOS2 directory to point to the new driver. The DEVICE statement for the video driver in the CONFIG.SYS also needs to be changed.

To disable SVGA support run the SVGA.EXE program with the OFF parameter:

```
SVGA OFF
```

Changing DOS settings may rectify problems experienced when running with SVGA enabled:

- If corruption occurs when switching to the WIN-OS/2 full-screen session, turn **VIDEO_SWITCH_NOTIFICATION** ON.

- If difficulty is experienced with animated graphics, turn **VIDEO_RETRACE_EMULATION** OFF.

Some manufacturers of SVGA hardware are working to develop OS/2 V2.0 video device drivers that will support Presentation Manager. It is advisable to check with the hardware supplier about the availability of drivers for a particular video subsystem. Information will also be posted on OS/2 bulletin boards and these can be checked for the latest information.

6.3.5 AT Bus Serial Port Support

COM.SYS will support COM3 and COM4 ports on AT bus machines. This is in addition to the published support for COM1 and COM2. In order to use COM3 and/or COM4 on AT bus machines it is necessary to include parameters on the DEVICE=COM.SYS command in the CONFIG.SYS. The format of the statement is:

```
DEVICE=COM.SYS (N,XXX,I)

where N   is the COM port number        ( 3 or 4 )
      XXX is the I/O port address       ( 3E8, 2E8, etc )
      I   is the IRQ (interrupt) level  ( from 1 to 15 )
```

The I/O port address will be found in the specifications for the adapter card. The port address must be unique for each COM port. The port address for COM1 is 3F8 and for COM2 it is 2F8.

The IRQ level must be chosen in such a way that it does not conflict with another adapter card in the system. COM1 uses IRQ 4 and COM2 uses IRQ 3. It is possible that the IRQ level of the COM3 or COM4 adapter cards is not settable and will default to either IRQ 3 or IRQ 4. In this case it is still possible to have up to four COM ports installed. However, if two or more ports use the same IRQ level, it is only possible to have one of those ports active at any one time. If the IRQ level is settable on the adapter card, choose an IRQ level not used in the system. Levels that are usually available include 5, 10 and 15.

6.3.6 Pointing Device Support

OS/2 Version 2.0 provides support for some of the popular pointing devices such the IBM PS/2 mouse and several mouse types which attach to the serial port. Some pointing devices are supplied with their own device drivers. An example of this is the Hewlett-Packard** (HP) mouse. Support for these devices can only be activated after installation of OS/2 V2.0 is completed. The OS/2 V2.0 installation will have to be carried out using the keyboard. During the installation, install support for the IBM PS/2 style pointing device.

After the system has been restarted, support for the pointing device can be installed. This installation will usually involve copying one or more device drivers to the *\OS2* directory and updating the CONFIG.SYS. In the case of the HP mouse and assuming OS/2 is installed on the C: partition, the steps involved are:

1. Copy the EXBIOS.SYS and HILMOU.SYS to the C:\OS2 directory from the diskette supplied with the mouse
2. Update the CONFIG.SYS as follows:

```
add-----> DEVICE=C:\OS2\EXBIOS.SYS
add-----> DEVICE=C:\OS2\HILMOU.SYS
modify--> DEVICE=C:\OS2\MOUSE.SYS TYPE=HILMOU$
```

6.3.7 When OS/2 Version 2.0 Will Not Run

IBM has been working with a number of OEMs of IBM-compatible PCs to ensure that OS/2 Version 2.0 will work on as wide a range of hardware as possible.

At general availability of OS/2 V2.0, a list of OEM machines that have been tested and support OS/2 V2.0 will be published. The updating of this list will be an ongoing process. Even if your machine appears on this list, you will need to check if there are any special considerations for installing OS/2 V2.0 on your machine. The dealer from whom you purchased the machine will be able to provide you with details supplied by the manufacturer and any special device drivers you may need.

Also to be considered is the support required for any adapters installed in the machine, which are not normally standard for the machine. This consideration applies to IBM PS/2s as well. For example, it is common to replace the disk controller in older machines with a newer SCSI controller. If this is an IBM controller, it is not a problem. However if a controller from an OEM is used, then a different device driver might be required.

If OS/2 V2.0 fails to run on a particular machine, these are some of the things that can be checked:

1. Check the documentation shipped with OS/2 V2.0 such as the online *README File*, which is found in the *Information* folder, for any special installation considerations which apply.
2. If any OS/2 bulletin boards are available, access them for any revised installation instructions or device driver requirements for the machine's configuration.
3. Check that the microprocessor on the system board is from the 80386 or 80486 families of processors.
4. If it is an 80386 processor, check that it has a stepping level of D0 or later.
5. Check that at least 4MB of memory is installed and available.
6. Check that there is sufficient free space in the disk partition containing SWAPPER.DAT to allow for growth of this file.
7. Check that the BIOS/ABOIS installed is the latest available for the machine.
8. Check if the disk system needs a special device driver and if there any special instructions for installing it.
9. Check for any other device driver requirements.
10. If installing on an IBM PS/2 Model 57, 90 or 95 update the system partition using the latest available reference diskette.

6.4 Summary

The I/O supervisor components of OS/2 Version 2.0 are designed to make greater use of advanced I/O devices than previous versions of OS/2. Particular emphasis has been given to the optimization of disk access through the use of the High Performance File System and SCSI devices.

New to OS/2 V2.0 is the Layered Device Driver architecture which enables the device drivers to be developed more easily to support OEM hardware. This interface expedites the development of new DASD and SCSI device support by OEM's by reducing device driver code and complexity. This architecture allows OS/2 to better support a broad range of OEM platforms and devices.

Device drivers are provided with new device help commands to enable drivers to more efficiently use the 32-bit flat memory model and the paging facility for virtual memory management.

A new disk device driver has been defined for use by the High Performance File System, which optimizes access to SCSI devices by allowing request prioritization and making use of the command chaining capabilities of the SCSI adapters. This optimization allows more efficient use of the physical devices. For paging operations where the sequence of read and write operations may be critical, the device driver allows the request optimization to be disabled.

Chapter 7. Boot Manager

OS/2 Version 2.0 provides the capability to support multiple operating systems on the same machine, by providing an operating system independent boot utility known as **Boot Manager**.

For example, OS/2 Version 2.0, DOS and AIX may be installed on the same physical machine, using separate fixed disk partitions and/or disks as supported by Boot Manager. When starting the machine, a user is presented with a panel allowing selection of the required operating system. The partition which contains the desired operating system is then activated, and that operating system is loaded and started.

7.1 Boot Manager Architecture

The following is a brief description of the Boot Manager architecture provided by OS/2 Version 2.0, and is provided in order to facilitate planning for Boot Manager support. Readers who are primarily interested in using Boot Manager rather than understanding its implementation may wish to skip this section and proceed to 7.2, "Partitioning the Fixed Disk" on page 89.

7.1.1 Partitions

Each physical disk drive in a machine is divided into **partitions**. A partition defines the area of the disk that belongs to a particular operating system. Each disk can be divided into multiple **primary partitions** and/or one **extended partition**. Each operating system formats its partition in its own way; this format may or may not be supported by other operating systems.

Only one primary partition on each physical disk drive is accessible at any one time; other primary partitions on that drive are hidden. Boot Manager allows up to four primary partitions on each physical disk drive.

Note that the first physical disk drive in the machine *must* have a primary partition.

An extended partition can be created in place of one of the four primary partitions when logical drives are desired. All logical drives in an extended partition are accessible at any time, regardless of which primary partition is active, provided that the format of the logical drives is supported by the currently active operating system.

7.1.2 Logical Drives

Operating systems typically access partitions through **logical drives**, each of which is assigned a logical drive letter. A primary partition contains only one logical drive, but an extended partition may be subdivided into one or more logical drives. An operating system only assigns logical drive letters to those partitions with a recognizable format, that is, supported by the operating system.

The assignment of drive letters is done by the operating system's volume manager when the operating system is loaded. The first physical disk drive is searched for a recognizable primary partition. This partition is assigned the logical drive letter "C" and the next physical disk is searched. After logical

drives in primary partitions have been assigned for all physical drives, drive letters are assigned to the recognizable logical drives in extended partitions.

A different operating system may reside on each of the logical drives in an extended partition (with certain limitations; see 7.7, "Operating System Restrictions" on page 102). Usually, DOS 5.0 and OS/2 V1.3 will only be installed on a primary partition on the first physical disk. An exception could be with DOS, for example, where there are no recognizable primary partitions on the first physical disk. In this case DOS (V4.1 or later) may recognize the primary partition(s) on the first physical drive to be formatted with the HPFS or Boot Manager, and treat them as unusable to DOS. In these situations, the first drive letter C: may be assigned to a primary partition on the next physical disk.

Logical drives in extended partitions are shareable; any data installed on these logical drives can be used by an operating system running from any other logical drive in the system, provided the partition formats are compatible. Although 26 logical drive letters are available (A: through Z:), A: and B: are typically reserved for diskette media. Under DOS or OS/2, up to 24 logical drives may be created using the logical drive support for extended partitions.

7.1.3 Logical Drive Boot Names

A user may assign a **boot name** to a logical drive, to simplify its identification. A boot name is a unique name which identifies the logical drive for booting, and is independent of the logical drive letter assigned to the drive by the currently active operating system. The boot name is only recognized by the Boot Manager.

Boot names are useful in the event of a hardware reconfiguration where additional fixed disks are added to the machine. Boot names assigned to logical drives on the original fixed disk will not change and will still identify the same area on the disk, unlike the logical drive letters which may change due to the reconfiguration.

7.1.4 Multi-Boot Block

The Boot Manager architecture distinguishes between *system-independent* and *system-dependent* components as part of the startup process of an operating system. The system-independent components are used to connect the POST code sequence executed on a PS/2 or compatible machine to a system selection sequence supplied as part of Boot Manager, which then chains to the operating system-dependent initialization sequence. The system-independent components are:

- The **Master Boot Record** (**MBR**) located in the first sector of the physical disk drive.
- A **Multi-Boot Block** (**MBB**) which resides in a primary partition on the physical drive, outside of the logical disks accessible by operating systems.

When Boot Manager is enabled as a *startable primary partition*, the MBR boots it first, like traditional MBR environments. The MBB manages the remainder of the boot process.

The layers of system independent and system dependent code on the physical disk drive are shown in Figure 21 on page 87.

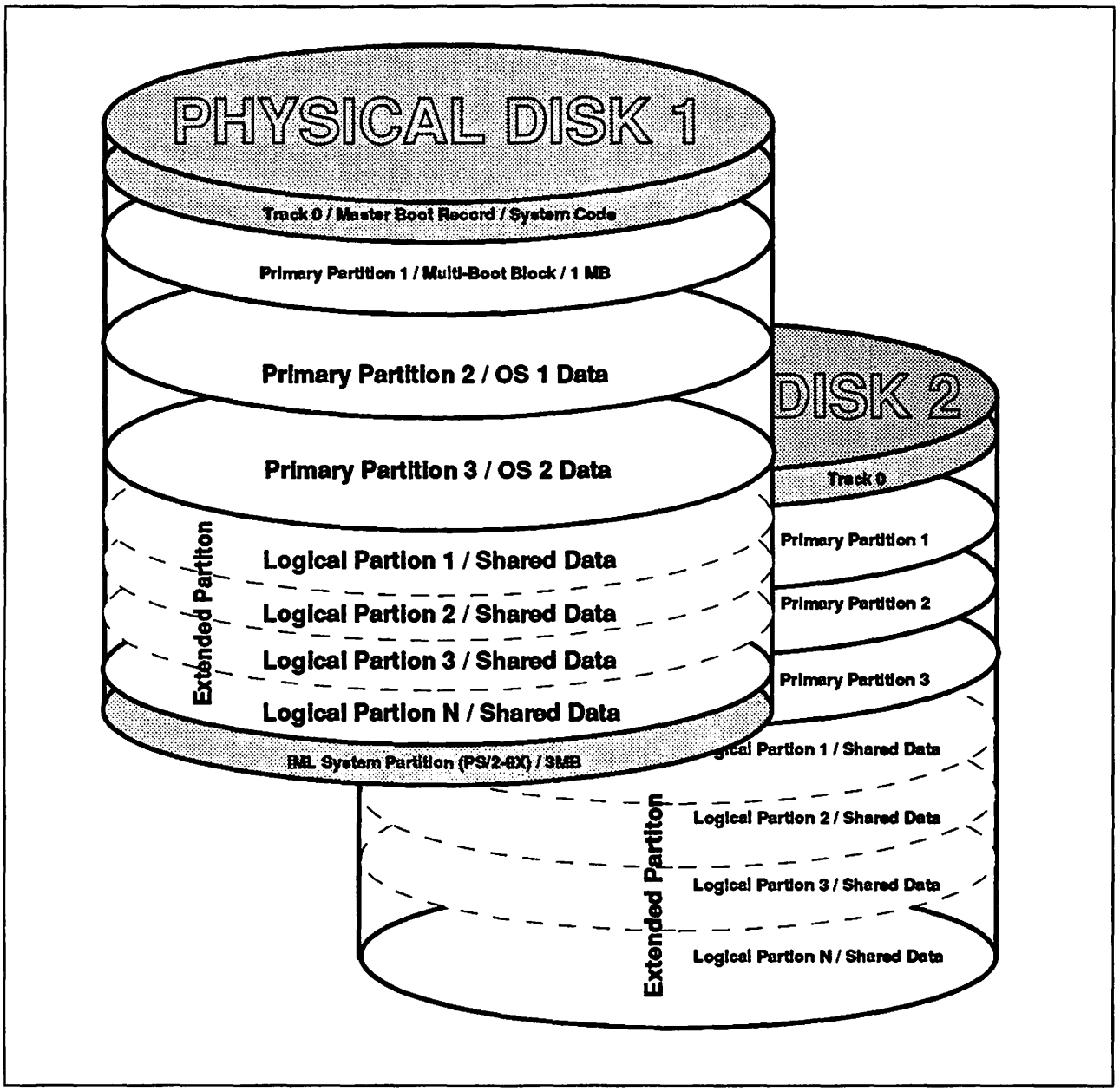


Figure 21. Hard Disk Layout

The MBB is installed in a 1MB primary partition which must be created at the request of the user, by installing Boot Manager. This space is allocated on request by OS/2 Version 2.0's FDISK/FDISKPM utilities and can be created at any location on the disk after track zero. It uses one of the four available primary partitions on the first physical disk.

The MBB contains code that:

- Displays the logical drive selection menu.
- Allows the user to select the logical drive from which to start the system.
- Gives control to the boot record of the selected logical drive.
- Manages a timer tracking the time available for logical drive selection before using a default logical drive.

- Contains a data area for the names of the designated default, fallback, and selectable logical drives, the time-out value for the selection menu and some reserved data space.
- Contains a data area reserved for boot error messages.
- Contains a data area describing locations and status of logical drives by boot name.

The operating system dependent code resides within the logical drives assigned to each operating system. Upon selection of a specific system to be started, this system specific code is executed, loading the chosen operating system.

The logical drives are allocated using the FDISK/FDISKPM utilities provided with OS/2 Version 2.0. These tools update the MBR to indicate which areas on the physical disk have been defined as logical drives containing the operating systems. Boot Manager logical drives can be defined on any ST-506, ESDI or SCSI interface fixed disk drive which is accessed through INT 13h. This precludes logical drives from being created on SCSI devices which are accessed through INT 4Bh; for example, logical drives may not be created on removable media devices.

Once one or more physical drives are set up and logical drives created, the specific operating system's formatting utility is used to supply the operating system boot record within the logical drive. It is important to note that the OS/2 Dual Boot function operates with the system-dependent boot code. Boot Manager neither enhances nor disables the Dual Boot function, since Boot Manager does not affect the contents of any logical drive. Therefore, Dual Boot will work in conjunction with and independent of Boot Manager.

No operating system may store its own information in the MBB or MBR. Any such information *must* be held within the logical disks owned by that operating system. Operating systems which do not obey this rule will not function correctly in the Boot Manager environment nor in any other "multi-boot" environment, and may compromise disk integrity. Operating systems which are known to obey this rule, and operate correctly in the Boot Manager environment, include:

- DOS Version 3.2 or above, 4.x and 5.0
- OS/2 Version 1.x
- OS/2 Version 2.0
- AIX.

7.1.5 Migration from Other Operating Systems

The Boot Manager architecture requires repartitioning of fixed disks already set up under any operating systems other than OS/2 Version 2.0. Although this may seem troublesome, it must be remembered that Boot Manager is only of use when multiple partitions are allocated, so it is likely that repartitioning would be necessary to support multiple operating systems.

Migration from previous versions of OS/2 to OS/2 Version 2.0 with Boot Manager is eased by installation support that allows migration to the new Boot Manager scheme with minimal user impact, other than the possible repartitioning discussed above. A Boot Manager installation on a system that had previous versions of OS/2 or DOS installed attempts to minimize the impact by providing as much automatic support during installation time as possible.

Note

There seems to be the possibility of a conflict between OS/2 V2.0's Boot Manager and other *third party disk organizers and boot utilities*. In order to avoid any trouble which could come up after installation of OS/2 V2.0 in these situations, it is highly recommended that a *low-level format* be performed before installing OS/2 V2.0. On a PS/2, such a *low-level format* can be performed by booting from the appropriate *reference diskette* and executing the *Advanced Diagnostics (CTRL-A)*. Once Boot Manager is installed, future migration steps will not require such an operation.

7.1.6 Performance Impacts

The path length added to the overall system startup time path consists solely of the instructions contained in the MBB. The OS/2 Version 2.0 MBB is approximately 16KB in size, with about 50% being instructions and the remaining 50% being data and space for the boot table and error messages. Tests have shown no measurable increase in overall startup time, counting the time from POST exit to display of the selection menu, plus the time from the selection being entered to the appearance of the first OS/2 or DOS operating system message on the screen. Based on these tests, the impact of Boot Manager during boot time is considered negligible.

The MBB is loaded into memory during startup, and is later overwritten by the operating system boot record loaded from the logical drive from which the system is IPLed. Thus, no memory impact is realized when using Boot Manager. Memory requirements for the various operating systems participating in a Boot Manager environment do not change.

7.2 Partitioning the Fixed Disk

Boot Manager can be installed, the machine's fixed disks partitioned, and logical disks created at the beginning of OS/2 Version 2.0 installation. It is possible to install Boot Manager after the operating system has been installed by use of OS/2 V2.0's full-screen FDISK command. See 7.2.1, "Boot Manager Installation" on page 90 for further information.

The names of the logical drives presented at the Boot Manager startup menu are stored in the MBB for primary partitions and in the **Extended Boot Record (EBR)** for logical drives in extended partitions. For SCSI drives that may be moved freely between different workstations, this approach allows Boot Manager to identify that drive continuously via the same logical name, even though the internal drive letter assignment may have changed.

Operating systems such as OS/2 and DOS are installed on the logical drives, and the user may then switch back and forth between operating systems via the selection menu displayed at IPL time.

OS/2 Version 2.0 provides two programs to create and maintain partitions on fixed disks. These programs provide substantially similar function and differ primarily in their presentation support requirements:

- **FDISK** operates in character mode in a full-screen session, and supports command line parameters and options, but does not support pointer device (mouse) input. FDISK is provided for use during the initial installation of

OS/2 Version 2.0, before the Presentation Manager components are loaded, and for occasions when it is not possible to load Presentation Manager or a command line parameter-driven interface is required.

- **FDISKPM** operates on the Presentation Manager desktop and supports pointer devices, but does not support command line parameters and options.

The normal expectation is that FDISKPM will be utilized for partition maintenance, unless there is some reason that Presentation Manager cannot be loaded.

An FDISK utility was supplied with OS/2 Version 1.3. However, its only function was to completely erase any existing partition information and as such it was only intended for use by the installation process, had no meaningful user interface and was not documented. The primary difference between FDISKPM supplied with OS/2 Version 1.3 and the FDISK/FDISKPM supplied with OS/2 Version 2.0 is the addition of a facility for the creation and management of multiple primary partitions together with additional facilities for management of logical drives, as well as support for managing multiple operating systems installed on the logical drives.

7.2.1 Boot Manager Installation

Since the only way that a physical drive may be partitioned for Boot Manager is to utilize OS/2 Version 2.0, it must be the first operating system installed.

> **Note**
>
> OS/2 Version 2.0 will not specifically prompt for the installation of Boot Manager as part of the normal installation process. To ensure that Boot Manager is installed, the user should elect *not* to accept the default disk partitioning when prompted by the installation procedure. The user should take the option to modify the default partitioning if Boot Manager is required to be installed.

Any disk partitioning required during installation is provided through the full-screen FDISK utility. To ensure that Boot Manager support is installed, "Install Boot Manager..." must be chosen and remain startable. If users wish to install an operating system on any one of the partitions available, they must make this partition *installable*. The FDISK utility provides advisory warning messages to remind users of this requirement.

Note that after installing OS/2 V1.3 or DOS 5.0 on one of the primary partitions, this partition will automatically become *startable*. To reactivate Boot Manager, the user has to use the FDISK command in order to make Boot Manager *startable* again.

7.2.2 FDISKPM Program

The following description applies to the FDISKPM program. The facilities provided by the full-screen interface of the FDISK program are generally the same, though they are accessed slightly differently. Since there are such strong similarities between FDISK and FDISKPM, only the FDISKPM program will be described in this section. The command line interface for FDISK is described in 7.2.3, "FDISK Program" on page 94.

FDISKPM is used to create or delete logical drives in primary or extended partitions on a fixed disk. With FDISKPM, a user can use Boot Manager to set up fixed disks on the system. The FDISKPM interface is shown in Figure 22 on page 91 and Figure 23 on page 91.

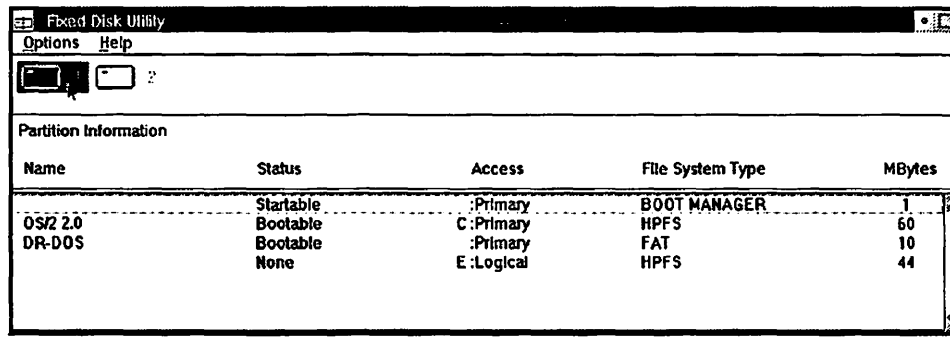


Figure 22. FDISKPM Showing Disk One (of a Two Disk System)

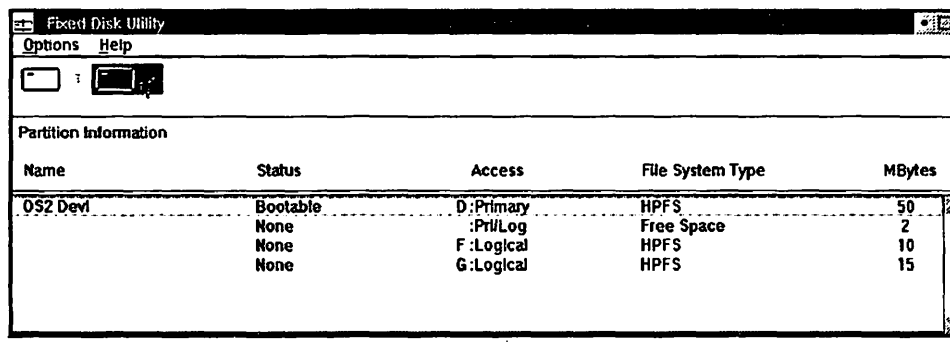


Figure 23. FDISKPM Showing Disk Two (of a Two Disk System)

The FDISKPM window has five columns containing specific information about the partitions that exist on the hard disks in the system. Each hard disk is represented by an icon at the top of the FDISKPM window. When a hard disk is selected, its partition information is displayed in the window. Partitions are either primary partitions or logical drives within an extended partition. Any free space (space within the hard disk that is available for more partitions) is also displayed in the window.

This information includes:

- **Name**: Displays the name that has been assigned to any primary partition or logical drive to be displayed on the Boot Manager menu. This name is specified using the *Add to Boot Manager* menu choice and can be changed by using the Change Partition Name choice.
- **Status**: Shows the partition status, which may be one of the following.
  - **Installable**: the partition is used as the target for installing another operating system.
  - **Bootable**: the partition is displayed on the Boot Manager menu when the system is restarted.
  - **Startable**: the system will restarts directly to this partition. This option is available for primary partitions only. Remember, one of the primary partitions must be set as startable for the system to restart successfully.

- **Access**: specifies if a partition is accessible. The letter in the column indicates that the partition is accessible. This column also indicates if the partition is a primary partition or a logical drive within the extended partition.
- **File System Type**: indicates the type of file system on the partition. Any partitions that have not been formatted will be displayed as unformatted. Any area on the hard disk not assigned to a partition will be displayed as free space.
- **MBytes**: indicates the size in megabytes of the partition or free space.

The user may select choices on the options pull-down to:

- Install the Boot Manager partition.
- Create a primary partition or logical drive.
- Add a partition to the Boot Manager menu.
- Change the partition name.
- Assign the accessibility of primary partitions.
- Specify startup values such as a default partition, startup selection time, or mode for the Boot Manager menu.
- Remove a partition from the Boot Manager menu.
- Delete a primary partition or logical drive.
- Set a primary partition as installable.
- Specify a primary partition as being startable.
- Exit FDISKPM.

Note that a hard disk and partition must be selected before these operations can be performed.

Setting a Partition Installable

The FDISKPM *Set Installable* menu option is used to set the selected partition as the target for continuing the OS/2 installation, or as a target to install another operating system. When a partition is set installable, the installable indicator will be displayed in the Status column.

Creating and Deleting a Partition

The FDISKPM *Create* menu option is used to create logical drives in primary or extended partitions on the selected physical drive. In addition, the user may do the following:

- Create multiple primary partitions (up to four for each physical drive).
- Place the partitions in relation to the beginning or end of each free area on the physical drive.
- Assign extended partition space as each logical drive is created. This provides flexibility for later assignment of free space. For example, if free space exists between a primary and extended partition, it may subsequently be allocated to another primary partition or for expanding the extended partition.

The FDISKPM *Delete* menu option is used to delete primary or extended partitions on the selected physical drive.

Warning

All data in such a partition is lost after **saving changes** on exiting FDISK.

Making a partition Bootable

The *Add to Boot Manager menu* option is used to name the partition and add it to the Boot Manager menu at startup. When *Bootable* appears in the *Status* column, the partition will be displayed. To remove the partition so that it is not displayed on the Boot Manager menu, select *Remove from Boot Manager menu.*

Assign Primary Partition

The *Assign x: partition* option is used to specify the accessibility of primary partitions. Because only one primary partition for each hard disk can be active (accessible) at a time, other primary partitions are effectively hidden (inaccessible). This feature exists for primary partitions only. All logical drives are accessible.

Making a partition Startable

Use *Make startable* to make a primary partition that you have highlighted on the FDISKPM window the one that will be activated each time you start the system. To have Boot Manager support active and have the Boot Manager menu displayed when the system is started, the Boot Manager partition must be set as startable. To deactivate Boot Manager support so that the Boot Manager menu is not displayed, set another primary partition as startable, that is, the partition that you want started when the system is turned on. Only one primary partition is startable at a time.

Partition Names

Partition names are important especially in the event of a hardware reconfiguration such as adding other physical disks to the system. The names assigned to partitions will not change and still identify the same area on the disk, unlike the drive letters which can change if your system is reconfigured. Names are up to eight characters in length and are case sensitive. Use the *Change partition name* to rename any bootable partitions.

Setting Startup Values

The FDISKPM *Set startup values* is used to specify the startup values for the partitions on the system. The values are for the following:

- A default partition to be started automatically without being selected from the Boot Manager menu.
- A startup selection time allowing you to specify the amount of time you want the menu to be displayed before the default partition is started automatically.
- A mode for the menu allowing you to indicate how much information about the partitions you want displayed on it.

7.2.3 FDISK Program

The full-screen version of FDISK is used during installation of the operating system. It provides users with the same functions as the FDISKPM version. The full-screen version looks and acts in much the same way as FDISKPM, but does not support a mouse. The FDISK full-screen interface is shown in Figure 24.

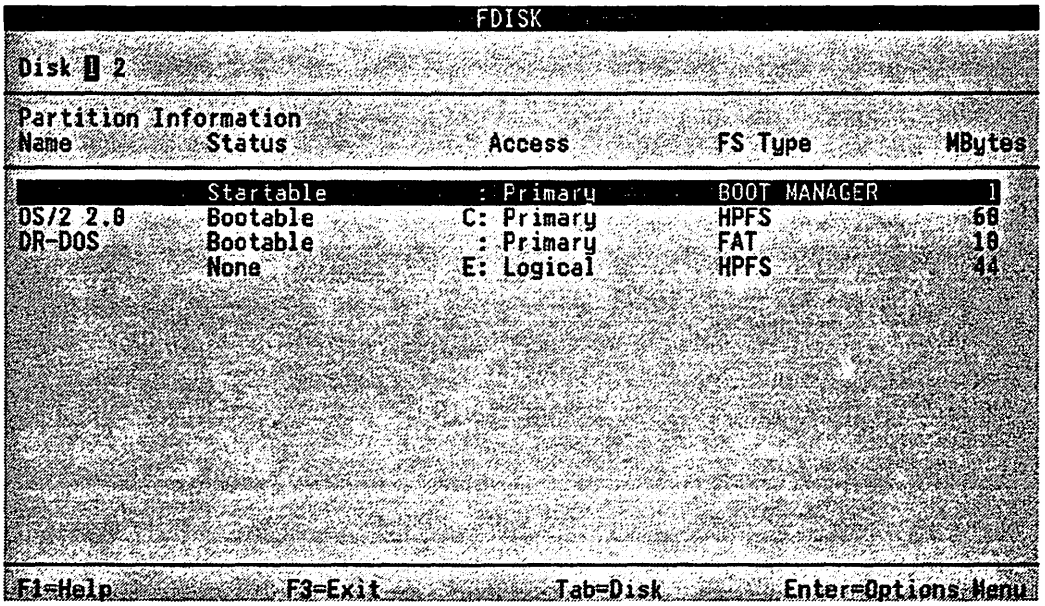


Figure 24. FDISK Utility (in Full-Screen Mode)

In order to enable the initial installation environment where Presentation Manager facilities are not available, the FDISK program provides a command line interface with similar capabilities to those provided by the FDISKPM program. When combined with the setboot command line utility, the FDISK command line interface also allows modification of logical drive environments and changes to Boot Manager values via batch files or remote command line interfaces such as DCAF, for use in unattended environments.

The following syntax shows how to use the FDISK command:

```
<Drive:><path> FDISK parameter:value </option:value>
```

The ***PARAMETERS:VALUE*** specified in the FDISK command may be one of the following:

/QUERY Displays a list of all partitions and free space on the hard disks of the system. To display a list of all partitions and free space on the hard disks of the system, type:
FDISK /QUERY

/CREATE:name Creates a primary partition or logical drive in an extended partition. You can specify an optional name for the created partition. You must specify the type of partition being created by using the /VTYPE:n option where n = 1 for a primary partition and 2 for a logical drive in an extended partition. To create a logical drive in an extended partition on disk 1 with the name OS2FAT, type:
FDISK /CREATE:OS2FAT /VTYPE:2 /DISK:1

/DELETE Deletes a logical drive or primary partition. This parameter must be used with one or more options. You can use /DELETE:all to delete all logical drives on a disk. If you use /DELETE:all, however, you must specify the disk using the /DISK option. To delete a logical drive with the name OS2FAT, type:
FDISK /DELETE /NAME:OS2FAT

/SETNAME:name
Specifies names for primary partitions or logical drives and makes them bootable from the Boot Manager. If the name is left blank, the boot name is removed and the partition will not be bootable from the Boot Manager menu. To specify the name DOS4.0 of a primary partition, type:
FDISK /SETNAME:DOS4.0

/SETACCESS Sets a primary DOS partition as accessible. Once a primary DOS partition has been set as accessible, all other primary DOS partitions on the same drive are inaccessible. If there are two primary DOS partitions on a hard disk, specifying this parameter with no options makes the previously inaccessible partition accessible and changes the previously accessible partition to inaccessible. To specify a primary DOS partition as accessible type:
FDISK /SETACCESS

/STARTABLE Specifies a partition as startable. To specify a partition as startable, type:
FDISK /STARTABLE

/FILE:filename Processes all FDISK commands in the specified file allowing the batching of FDISK commands. You must have commas separating the arguments of each command in the file and the commands are processed just once. For example, the file MYFILE contains the following commands:

```
/query
/create:OS2,/vtype:1,/disk:1,/name:01000030,/size:20
/startable,/name:OS2
/query,/name:OS2
```

The processing of these commands is as follows:

Command 1: displays a list of all partitions and unused space on all hard disks.
Command 2: creates a primary partition on disk 1 with a size of 20MB in the free space alias 01000030 and assigns it a boot name of OS2.
Command 3: sets the partition named OS2 to be the startable partition.
Command 4: displays the partition information of the partition named OS2. To process FDISK commands in the file MYFILE, type:
FDISK /FILE:myfile

OPTIONS limit the actions of the FDISK command and parameters. The valid options and their associated values are:

/NAME:name Indicates the name of a partition. A name can be up to eight characters in length and is case-sensitive. The /NAME option can be used with all FDISK parameters except /FILE.

Note: During a QUERY operation, a temporary name is assigned to every partition and free space that does not have a boot name assigned. This name is not set as the partition name, but is only used as a temporary identifier for the user. Because they will not have a visual representation of the FDISK screen, these temporary names can be used in place of real names for the NAME option. To delete a partition named dos, type:
FDISK /DELETE /NAME:dos

/DISK:n Specifies the number of the hard disk that you want to work with using the FDISK command and parameters. The /DISK option can be used with all FDISK parameters except /FILE. To display all partitions on drive 2, type:
FDISK /QUERY /DISK:2

/FSTYPE:x Specifies the file system type of the partition. The type x=DOS, FAT, IFS, Free, Hnn, or other. "Hnn" is the hex value (nn) of the partition system ID value.

The /FSTYPE option can be used with all FDISK parameters except /FILE and /SETACCESS. To display a partition with a FAT file system, type:
FDISK /QUERY /FSTYPE:FAT

/START:m Specifies the partition starting location. Specify m=t or m=b, where t=top of the partition and b=bottom of the partition. The /START option can be used with all FDISK parameters except /FILE. To create a primary partition starting at the top of the partition, type:
FDISK /CREATE /START:t

/SIZE:m Specifies the size of the partition where m is the size in MB. The /SIZE option can be used with all FDISK parameters except /FILE. To create a primary partition with a size of 8MB, type:
FDISK /CREATE /SIZE:8

/VTYPE:n Specifies the type of the partition. The value of n can be as follows:

0 Space is not usable
1 Primary partition (not shared)
2 Logical drive (shared in an extended partition)
3 Free space that can be used to create a primary or extended partition.

The /VTYPE option can be used with all FDISK parameters except /FILE, /SETACCESS, and /STARTABLE.
To display unusable space on a disk, type:
FDISK /QUERY /VTYPE:0
To specify a primary partition to be displayed, type:
FDISK /QUERY /VTYPE:1
To specify a logical drive in an extended partition to be displayed, type:
FDISK /QUERY /VTYPE:2
To display free space that can be used to create a primary or extended partition, type:
FDISK /QUERY /VTYPE:3

/BOOTABLE:s Indicates the bootable status of partitions; s=0 for partitions that are not bootable; s=1 for partitions that are bootable. The /BOOTABLE option can be used with all FDISK parameters except /FILE. To display all partitions that are bootable from the Boot Manager menu, type:
FDISK /QUERY /BOOTABLE:1

/BOOTMGR Specifies an action for the Boot Manager partition. To create the Boot Manager partition, type:
FDISK /CREATE /BOOTMGR

7.3 SETBOOT Utility

The SETBOOT utility provides a command line interface which enables the user to configure the parameters associated with the Boot Manager environment.

Invoking the SETBOOT utility is achieved using the following command:

```
<Drive:><path> SETBOOT parameter <:value>
```

where *parameter:value* is one of the following:

/IBD:X Restarts the system from the specified logical drive, *X*, without going through the Boot Manager menu. The parameter *X* is the drive letter of a startable partition. If Boot Manager is not present then only drive C can be specified. For example, to reboot the system immediately from drive *D* without going through Boot Manager type:
SETBOOT /IBD:D

/T:x Sets the time-out value of the timer for the Boot Manager selection menu. The specified value of *x* may be 0 to 999 or NO. A specified value of nnn is the time in seconds that the Boot Manager selection menu will remain displayed before automatically starting the default logical drive. A specified value of 0 seconds bypasses the Boot Manager selection menu entirely, booting the default system without any intervention. When the time-out value is NO, the timer is disabled, leaving the Boot Manager selection menu displayed until the user makes a selection.

/M:m Sets the mode for the Boot Manager selection menu:

- *m* = n sets the mode to display only the boot names of the logical drives marked bootable. This is the default mode.
- *m* = a (advanced mode) sets the mode to display additional information such as physical drive, file system type, etc.

/Q Queries the current Boot Manager environment. Returns the default volume boot names, time-out value, mode, and unattended operation logical drive assignments.

/B Performs an shutdown and then reboots the system, booting to the system set to the marked index value at the time of boot.

Note: This is not the complete system shutdown which can be performed via the &wps's desktop context menu. All it does is basically flushing all file system buffers and close all currently open files. PM Applications will not receive any message, they will just be cut off, where ever they were left.

/X:x This is an index which indicates the logical drive to Boot Manager from which the system is to be started:

- A value of 0 sets Boot Manager to attended mode operation and provides for a default system selection.
- A value of 1 to 3 puts Boot Manager in an unattended operation mode, in which case all attended mode functions are bypassed, including the selection menu. This is provided as a mechanism that unattended operation can use to implement a fallback boot sequence. For example, if an operating system fails to access a program to set the index, the subsequent boot will attempt to start the next fallback system (probably a more trusted system) and so on.

Boot Manager will start the system from the logical drive corresponding to the index and then, before exiting, will decrement the value by 1 so long as the value is greater than 1. If the value is not set by SETBOOT before the next boot, Boot Manager will start the system from a different logical drive.

/N:name Sets the partition or logical drive specified by the boot name and its corresponding index value as the operating system to be started. Up to four pair-combinations can be given, one for each index value, 0 to 3. A boot name is case-sensitive, and if it contains blanks, the /N:name pair must be enclosed in quotation marks.

The following shows the values for N:

N = 0 assigns the specified boot name as the default operating system.
N = 1 to 3 specifies the boot name to be started when the corresponding index is chosen to start.

To specify the logical drive with the boot name MYSYSTEM as the default operating system to be started type:
SETBOOT /0:MYSYSTEM

SETBOOT will provide a return code and the requested information. The information is output to STDOUT and can be piped and redirected. Return codes are 0 for successful operation and 1 for unsuccessful operation.

7.4 Selecting an Operating System

The selection of the logical drive from which the system is to be started is performed at IPL time. The user is presented with a panel containing a list of selectable logical drives. The user may explicitly select a logical drive, or may allow the selection timer to expire and have Boot Manager start the system from the default logical drive without further intervention.

If the time-out value was set to zero using the SETBOOT utility or FDISK/FDISKPM, the selection menu is circumvented and Boot Manager immediately starts the system from the default logical drive.

If the time-out value is other than zero, the selection menu is displayed as shown in Figure 25 on page 99.

If no default logical drive was specified, then the last system booted becomes the default.

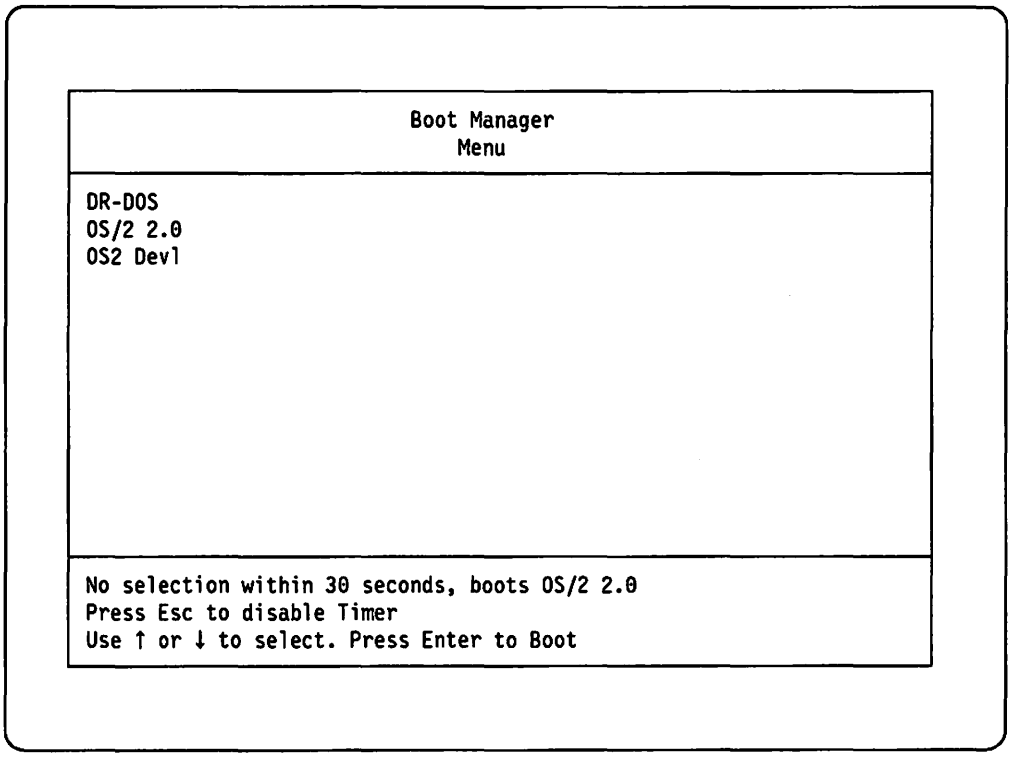


Figure 25. Boot Manager Selection Menu

The up/down cursor keys are used to select a particular logical drive. By default, the cursor is located at the current default logical drive. If more logical drives are configured in a workstation environment than lines are available on the screen to display them, a scroll bar will appear in the selection screen indicating to the user whether more logical drives are available for display.

The selection menu displays the current time-out value, and the boot name of the current default logical drive. In order to allow the user to suspend or resume the timer while the selection menu is displayed, the Esc key may be used in toggle mode. The current status of the timer is displayed (enabled/disabled) and the Esc key will toggle a status switch.

The selection menu shown in Figure 25 is the standard mode normally seen at power-on or IPL time. Only the boot names of selectable logical drivers are displayed; no additional information about the type of logical drive and physical location is shown in order to keep the standard user interface simple and straightforward.

However, in order to accommodate advanced users and system administrators who wish to see more details about the available logical drives, an advanced mode menu is available. This advanced mode menu can be activated via an option in FDISK, FDISKPM or via the SETBOOT utility.

The advanced mode selection menu shows not only the boot names of the logical drives, but will also indicate the physical disk upon which the logical drive resides, the type of partition (primary/extended), the file system for which the logical drive was formatted, and the accessibility of the logical drive. The advanced mode selection menu is shown in Figure 26 on page 100.

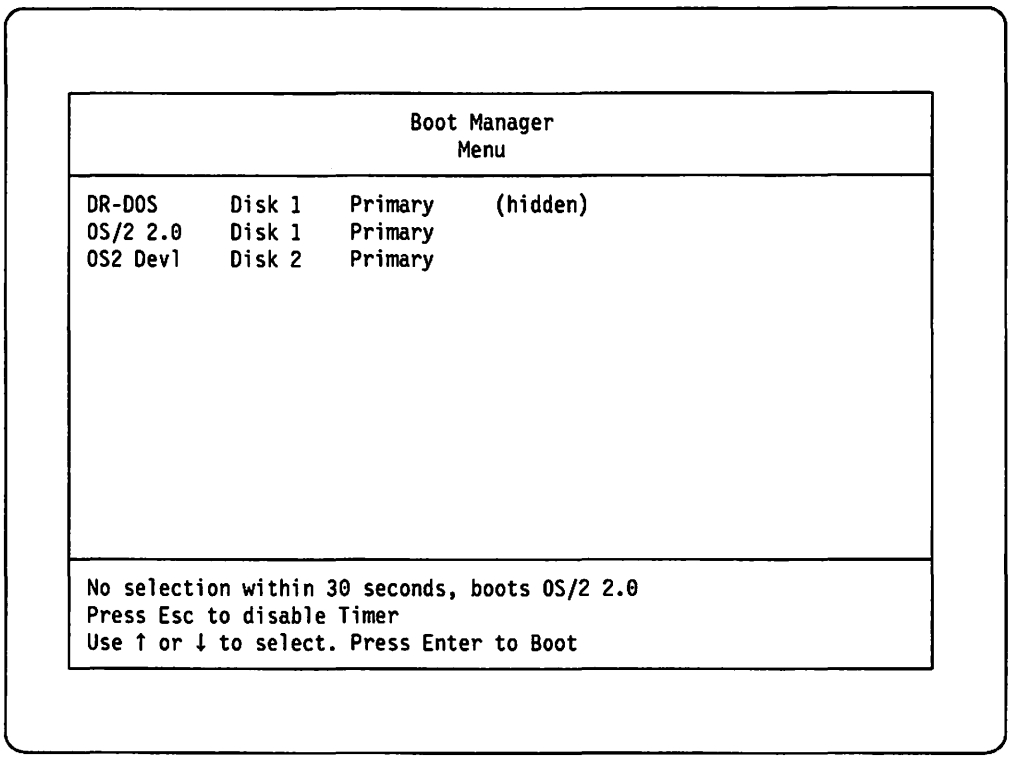


Figure 26. Boot Manager Selection Menu - Advanced Mode

Note that the third column displays the partition type for a logical drive boot name. *Logical* indicates that the drive resides in an extended partition.

The fifth column displays the accessibility of the logical drive. *Hidden* means the system will not have access to this volume; however, if a hidden boot name is selected as the logical drive from which to start the system, Boot Manager will "unhide" it. Boot Manager will automatically hide any other primaries on that physical drive, since only one primary partition may be active at any time.

Note: Only primary partitions can be marked as hidden, extended logical drives (partitions) are always accessible.

7.5 Sharing Partitions between Operating Systems

Boot Manager allows for both primary and extended partitions on a single physical drive. Only one primary partition may be active at a time. However, the extended partition will always be active, regardless of which primary partition is active. Logical drives in an extended partition may therefore be accessed by multiple operating systems started from primary partitions or from logical drives in the extended partition. This access is dependent on the operating system supporting the format of the extended partition.

Table 2 on page 101 illustrates some of the dependencies between operating systems and extended partitions.

Table 2. Partition Format Accessibility					
Format	**DOS 3.3**	**DOS 4.0**	**DOS 5.0**	**OS/2**	**AIX**
FAT-16	Yes	Yes	Yes	Yes	Yes
BIG-FAT	No	Yes	Yes	Yes	No
HPFS	No	No	No	Yes	No

Note that it is not possible for an operating system loaded from a primary partition to access any other primary partition on that hardfile; only the logical drive on the operating system's own primary partition and those on extended partitions may be accessed.

Where it is desirable to have a common set of application or data files which may be accessed by different operating systems, a workable strategy is to locate *only* those files which are specific to the operating system on the appropriate primary partition, while locating common application and data files on logical drives in extended partitions. This implies that the primary partitions should be created to be comfortably large enough to accommodate the desired configuration of the operating system concerned, but no larger.

Since the amount of disk space required to configure any operating system to meet particular needs will vary widely with those needs, readers should consult the installation and configuration instructions for the operating system concerned, in order to determine a reasonable partition size.

When setting up a complex set of operating systems and extended partitions, readers should carefully consider the implications of logical drive letters that each operating system will automatically assign to all logical drives visible to the operating system. This especially applies to applications which reference other applications or data located on different logical drives. The logical drive letters may not be consistent between operating systems; for example, a logical partition formatted for HPFS will be completely invisible to DOS 3.3, but visible though not accessible to DOS 4.0 (CSD UR31300). This may result in different logical drive letters being assigned to other logical drives, depending upon which operating system is currently active in the machine, which in turn holds implications for the configuration of applications and the design of batch files.

7.6 AIX Considerations

It is possible to install AIX, OS/2 and DOS in different partitions on the same disk and use Boot Manager to switch between the different operating systems. There are some things, however, to be aware of:

1. When AIX installs it creates two partitions on the disk. The first partition which is 1MB in size contains the MBR for AIX. The second partition contains several AIX minidisks. The AIX partitions cannot be created using either the DOS or OS/2 FDISK programs, but are created by the AIX install process.
2. The order that AIX, OS/2 or DOS are installed is not important. If installing AIX first, remember to leave sufficient free space available in which to create partitions for OS/2 or DOS. If installing OS/2 or DOS first, when running FDISK, do not create partitions across the entire disk. Also only allocate the Boot Manager partition and one other, thus leaving two available for AIX.

3. The OS/2 V2.0 FDISK and FDISKPM programs will recognize the AIX partitions. Use FDISK or FDISKPM to make the smaller AIX partition bootable. If AIX is installed second, it will be necessary to shutdown AIX and to boot from the OS/2 install diskettes, escape out of the install process and run FDISK from the full-screen command prompt to make Boot Manager *startable* and the smaller AIX partition *bootable*.
4. It is possible to install AIX on the second disk of a two-disk system. However AIX will still create a partition with its MBR on the first disk. This partition should be marked startable. On the second disk it also creates a partition to contain the MBR as well as the partition for the AIX minidisks.

7.7 Operating System Restrictions

Several restrictions apply to the manner in which various versions of operating systems may be installed under Boot Manager. These include:

- Only DOS Versions 3.2 and above are supported by Boot Manager.
- All supported versions of DOS *must* be installed on a primary partition on the first drive and therefore, end up working on the C: drive only.
- DOS Versions 3.2 and 3.3 *must* be installed in a partition that falls wholly within the first 32MB of a fixed disk.
- OS/2 Version 1.3 and earlier versions may only be installed on a primary partition on the first drive and therefore, end up working on the C: drive only.

Notes:

1. An operating system may NOT store its own information in the Multi-Boot Block or in the Master Boot Record on the physical drive. Any such information *must* be held within the logical disks owned by that operating system. Operating systems which do not obey this rule may not function correctly in the Boot Manager environment, and will compromise disk integrity.
2. Other operating systems may leave their partition *startable* following installation. The user would have to use their FDISK program or boot the OS/2 V2.0 installation diskette to mark the Boot Manager partition as *startable* again to enable the Boot Manager menu at startup. See also 7.2.1, "Boot Manager Installation" on page 90.

7.8 Summary

Boot Manager provides the capability to install and run multiple operating systems concurrently in the same machine. The user may select the desired operating system at system power-on or at IPL-time, using a selection menu provided by Boot Manager.

Operating systems reside on logical drives in both primary and extended fixed disk partitions. Partitions are created and manipulated using the FDISK, FDISKPM, and SETBOOT utilities provided with OS/2 Version 2.0.

Operating systems may share applications and data files, provided that the disk formats supported by each operating system are compatible. In order to be shared, such files must reside on logical drives in extended fixed disk partitions, or on physically separate disks.

Chapter 8. National Language Considerations

OS/2 Version 2.0 provides support for additional national languages over and above those supported by OS/2 Version 1.3. The discussion in this chapter will concentrate on the new languages supported, and will not cover the languages that were already supported in OS/2 Version 1.3.

8.1 Single-Byte Languages

New single-byte codepage support is added in OS/2 Version 2.0 for the following countries:

- Iceland
- Latin (Group 2) countries:
  - Czechoslovakia
  - Hungary
  - Poland
  - Yugoslavia
- Turkey.

Czechoslovakia uses two common national languages: Czech and Slovak. Support is provided for both these languages by providing two keyboard layouts: a primary keyboard layout for Czech/Czech, and a secondary keyboard for Czech/Slovak. This is similar to the method used to support countries such as the United Kingdom and France where more than one keyboard layout is in common use.

8.1.1 Iceland

OS/2 Version 2.0 provides the following support for Iceland:

- Country code: 354
- Keyboard layout: 197
- Primary codepage: 850
- Secondary codepage: 861.

8.1.2 Czechoslovakia

OS/2 Version 2.0 provides the following support for Czechoslovakia:

- Country code: 42
- Keyboard layout: 243 (Czech) or 245 (Slovak)
- Primary codepage: 852
- Secondary codepage: 850.

8.1.3 Hungary

OS/2 Version 2.0 provides the following support for Hungary:

- Country code: 36
- Keyboard layout: 208
- Primary codepage: 852
- Secondary codepage: 850.

8.1.4 Poland

OS/2 Version 2.0 provides the following support for Poland:

- Country code: 48
- Keyboard layout: 214
- Primary codepage: 852
- Secondary codepage: 850.

8.1.5 Yugoslavia

OS/2 Version 2.0 provides the following support for Yugoslavia:

- Country code: 38
- Keyboard layout: 234
- Primary codepage: 852
- Secondary codepage: 850.

8.1.6 Turkey

OS/2 Version 2.0 provides the following support for Turkey:

- Country code: 90
- Keyboard layout: 179
- Primary codepage: 857
- Secondary codepage: 850.

8.2 Double-Byte Languages

OS/2 Version 2.0 provides additional support for double-byte character sets, with four new SAA codepages supported. These are:

- 942 - Japanese
- 944 - Korean
- 946 - People's Republic of China
- 948 - Taiwanese.

This support is in addition to the previously supported codepages 932, 934, 936, and 938.

Double-byte codepages may *only* be used on hardware which supports the use of double-byte character sets, such as the IBM PS/55 family, and with the Asian version of OS/2.

8.3 Bidirectional Languages

Support is provided in OS/2 Version 2.0 for national languages such as Hebrew and Arabic, which require video display output and printer output to be presented for reading from left-to-right or right-to-left, and for the nesting of phrases in either direction. This bidirectional support is used in conjunction with the Hebrew and Arabic codepages, and uses a bidirectional video device handler (VDH) and a bidirectional printer monitor. This VDH is chained to the device-specific VDH and handles all the bidirectional aspects of the video output.

This support is *only* available for full-screen screen groups.

The bidirectional language support provided in OS/2 Version 2.0 will support those countries with country codes 785 (Arabic) and 972 (Hebrew), and which use

either codepages 864 (Arabic) or 862 (Hebrew) as the primary codepage. These countries may also use either codepage 437 or 850 as a secondary codepage.

A number of function calls are provided to allow querying and setting of bidirectional screen group attributes and the manipulation of character strings from within applications. This enables the development of bidirectional aware applications.

8.3.1 Installation

The OS/2 Version 2.0 installation program allows installation of bidirectional support for Arabic and Hebrew countries in a similar manner to other supported countries. The following configuration commands are affected:

- COUNTRY
- CODEPAGE
- DEVINFO
- RUN
- SET.

Note that the bidirectional support will not become active at installation time; it requires an IPL of the system in order to load the correct drivers and the device monitors.

8.3.1.1 Arabic

The following commands are used in CONFIG.SYS to support those countries using the Arabic language:

```
COUNTRY=785,C:\OS2\SYSTEM\COUNTRY.SYS
CODEPAGE=864,437
DEVINFO=KBD,AR,C:\OS2\KEYBOARD.DCP
RUN=C:\OS2\SYSTEM\BDPRTM.EXE
```

For each SET VIO_xxx=DEVICE(BVHxxx), the BDBVH driver is appended after the name of the device-specific video handler. For example when installing support for a VGA display, the SET=VIO_VGA line will be:

```
SET VIO_VGA=DEVICE(BVHVGA,BDBVH)
```

Please refer to *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775 for details of using the DEVINFO statement to prepare a printer which supports the Arabic codepage, to use the codepage and any specific fonts associated with the codepage.

8.3.1.2 Hebrew

The following commands are used in CONFIG.SYS to support those countries using the Hebrew language:

```
COUNTRY=972,C:\OS2\SYSTEM\COUNTRY.SYS
CODEPAGE=862,437      (or CODEPAGE=862,850)
RUN=C:\OS2\SYSTEM\BDPRTM.EXE
```

For each SET VIO_xxx=DEVICE(BVHxxx), the BDBVH driver is appended after the name of the device specific video handler. For example when installing support for a VGA display, the SET=VIO_VGA line will be:

```
SET VIO_VGA=DEVICE(BVHVGA,BDBVH)
```

Please refer to *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775 for details of using the DEVINFO statement to prepare a printer which supports the

Hebrew codepage, to use the codepage and any specific fonts associated with the codepage.

8.3.2 Programming Interface

The following 16-bit functions are provided for application developers to use the bidirectional language support provided in OS/2 Version 2.0:

- **NlsQueryBidiAtt()** returns the current bidirectional attributes that are in effect for the current screen group.
- **NlsSetBidiAtt()** sets specific values in the bidirectional control blocks (per screen group).
- **NlsShapeBidiString()** shapes or reshapes an Arabic string of specified length.
- **NlsEditshape()** reshapes an Arabic character (typically in an input or editing sequence) based on the two preceding characters, or reshapes the current and surrounding characters after an editing function has been carried out.
- **NlsInverseString()** inverts the sequence of characters in a specific string.
- **NlsConvertBidiNumerics()** converts numerics in a string to Arabic character codes.
- **NlsConvertBidiString()** converts a Bidi string conforming to a certain set of bidirectional attributes, to a form which conforms to a different set of bidirectional attributes.
- **NlsSetBidiPrint()** associates a given set of Bidi attributes with a particular print file.

8.3.3 Bidirectional User Interface

The following keyboard key combinations are supported in a full-screen screen group in a bidirectional support enabled system:

- **National-Language Layer** (Alt+Right Shift) switches the keyboard to its national language mode. Typed keys produce NL characters.
- **Latin Layer** (Alt+Left Shift) switches the keyboard to its Latin language mode. Typed keys produce Latin characters.
- **Screen Reverse** (Alt+Newline) switches the orientation of the screen. Screen maybe either in Left-to-Right or Right-to-Left orientation.
- **Field Reverse** (Alt+NumLock) temporarily switches the direction of the cursor within a line or field.
- **PUSH** (Shift+NumLock) temporarily switches the direction of the cursor within a line or field. Characters are PUSHed (as in a pocket calculator) in the direction which is opposite to the screen orientation.
- **End Push** (Shift+NumPad /) ends PUSH mode.
- **AutoPush** (Alt+NumPad /) Toggle key that enables automatic activation and deactivation of PUSH mode, based on the keys that are typed on the keyboard.
- **Bidi Status** (Alt+ScrollLock) activates a pop-up window that displays the bidirectional settings for the current screen group. The user is able to change these settings by manipulating fields in the pop-up window.

The following are Arabic-specific keyboard functions.

ITSC Technical Bulletin Evaluation

Technical Bulletin Title: _____

Technical Bulletin Form Number: _____

This is an evaluation form to assess the quality of ITSC publications. Your feedback will help maintain the high quality of ITSC standards. Please fill out this questionnaire and send it to the address on the back of this page. No postage stamp is required if mailed in the U.S. Elsewhere, you may choose to have your IBM Marketing Representative forward your reply to the address listed on the reverse side of this form.

Date publication was ordered (MM/DD/YY) ___/___/___

Date publication was received (MM/DD/YY) ___/___/___

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Technical Bulletin organization	___	Grammar/punctuation/spelling	___
Accuracy of the information	___	Ease of reading and understanding	___
Relevance of the information	___	Ease of finding information	___
Completeness of the information	___	Lack of redundant information	___
Value of illustrations	___	Overall satisfaction	___

Please answer the following questions:

a) Was the level of detail of the information adequate? Yes___ No___

b) Did you find information duplicated that was available in other IBM publications? Yes___ No___

If yes, please name the publication:

_____

_____

c) Was the bulletin published in time for your needs? Yes___ No___

d) Did this bulletin meet your needs? Yes___ No___

If no, please explain:

_____

_____

Comments/Suggestions:

Thank you for your feedback.

Your name, company name, and address (optional):

_____

_____

_____

_____

ITSC Technical Bulletin Evaluation Form

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:
IBM International Technical Support Center
Department H52, Building 930
P.O. Box 950
Poughkeepsie, New York 12602
U.S.A.

ATTN: Quality Coordinator

Fold | Fold

IBM®

PRINTED IN THE U.S.A.

- **AutoShape/Base** (Alt+NumPad 4) toggles the keyboard between automatically (context dependent) shaped characters and base shapes.
- **Shape Isolated** (Alt+NumPad 2) typed characters are in isolated shape.
- **Shape Initial** (Alt+NumPad 1) typed characters are in initial shape.
- **Shape Middle** (Alt+NumPad 7) typed characters are in middle shape.
- **Shape Final** (Alt+NumPad 8) typed characters are in final shape.

8.4 Message Files

Previous versions of OS/2 supported only a single codepage for message files used by the system and by applications within the system. This created a problem where a process used a codepage other than the system's primary codepage, since messages appeared using the primary codepage rather than the codepage specific to that process. OS/2 Version 2.0 resolves this problem by allowing multiple codepages in a single message file, and allowing a message to be retrieved by a process using the codepage currently used by that process. This level of support has resulted in changes in the following areas:

- The message file structure is modified to include codepage and language identification.
- The MKMSGF utility is enhanced to create multiple codepage message files.
- The MSGBIND utility is enhanced to allow the binding of messages for multiple codepages to a single application program.
- The **DosGetMessage()** function call in MSG.DLL is enhanced to retrieve messages based on the current codepage of a process.
- A new **DosQueryMessageCp()** function is added to MSG.DLL to retrieve the list of codepage and language identifiers present in the application's message file.

Using this support, an application may query the current codepage for its parent process using the **DosQueryCp()** function, ensure that codepage is supported by the application's message file by issuing a **DosQueryMessageCp()** call, and then issue **DosGetMessage()** calls specifying the appropriate codepage for the message to be retrieved. For most applications, it is recommended that the multinational codepage 850 be used where support for multiple national languages is required.

8.5 Information Presentation Facility

The following table shows the language files used by the Information Presentation Facility (IPF), in order to provide national language support.

Table 3. IPF NLS Language Files		
ID	**Language**	**NLS File**
CHT	Chinese	IPFCHT.NLS
DAN	Danish	IPFDAN.NLS
DEU	German	IPFDEU.NLS
ENG	English UK	IPFENG.NLS
ENU	English US	IPFENU.NLS
ESP	Spanish	IPFESP.NLS
FIN	Finnish	IPFFIN.NLS
FRA	French	IPFFRA.NLS
FRC	Canadian French	IPFFRC.NLS
ITA	Italian	IPFITA.NLS
JPN	Japanese	IPFJPN.NLS
KOR	Korean	IPFKOR.NLS
NLD	Dutch	IPFNLD.NLS
NOR	Norwegian	IPFNOR.NLS
PTG	Portuguese	IPFPTG.NLS
SVE	Swedish	IPFSVE.NLS
UND	User Defined	IPFUND.NLS

These files are installed as part of the *IBM Developer's Toolkit for OS/2 2.0*, and are used during execution of the IPF compiler. The compiler requires a symbol file APSYMBOL.APS to be available. When using DBCS languages, the default symbol file must be overwritten with a symbol file appropriate for the language being used. See *IBM OS/2 Version 2.0 Information Presentation Reference* for details.

8.6 Supported Countries

The supported countries and codepages are shown in Table 4.

Table 4. NLS Country Codes and Codepages

Country	Country Code	Codepages Supported (primary/secondary)
Arabic	785	864 / 437 or 850
Asian English	099	437 / 850
Australia	061	437 / 850
Belgium	032	437 / 850
Canadian English	001	437 / 850
Canadian French	002	863 / 850
Czechoslovakia	042	852 / 850
Denmark	045	865 / 850
Finland	358	437 / 850
France	033	437 / 850
Germany	049	437 / 850
Hebrew	972	862 / 437 or 850
Hungary	036	852 / 850
Iceland	354	850 / 861
Italy	039	437 / 850
Japan	081	932 or 942 / 437 or 850
Korea	082	934 or 944 / 437 or 850
Latin America	003	437 / 850
Netherlands	031	437 / 850
Norway	047	865 / 850
People's Republic of China	086	936 or 946 / 437 or 850
Poland	048	852 / 850
Portugal	351	860 / 850
Spain	034	437 / 850
Sweden	046	437 / 850
Switzerland	041	437 / 850
Taiwan (Traditional Chinese)	088	938 or 948 / 437 or 850
Turkey	090	857 / 850
United Kingdom	044	437 / 850
United States	001	437 / 850
Yugoslavia	038	852 / 850

8.7 Summary

OS/2 Version 2.0 has national language support for twenty-two languages:

- Arabic
- Canadian French
- Czech (Czech)
- Czech (Slovak)
- Danish
- Dutch
- Finnish
- French
- German
- Hebrew
- Hungarian
- Icelandic
- Italian
- Norwegian
- Polish
- Portuguese
- Spanish
- Swedish
- Turkish
- United Kingdom English
- United States/Universal English
- Yugoslavian.

OS/2 Version 2.0 provides support for all countries supported by OS/2 Version 1.3, with additional single-byte and double-byte national languages supported. Where a number of the newly supported countries use more than one common language, these languages are supported by OS/2 Version 2.0 as part of the national language support for that country.

OS/2 Version 2.0 provides support for message files which use multiple codepages. This allows an application to select messages from the file using the current codepage for the application's parent process, thereby enabling more comprehensive national language support by applications.

OS/2 Version 2.0 also provides support for bidirectional languages such as Arabic and Hebrew, in conjunction with the national language codepages for these languages, and keyboard and printer monitors implemented by the operating system. In addition, OS/2 Version 2.0 provides enabling support for those countries using bidirectional alphabets such as Arabic and Hebrew.

Appendix A. Intel 80386 Architecture

The Intel 80386 is a powerful 32-bit microprocessor and is the first hardware platform on which OS/2 Version 2.0 has been implemented. The 80386 incorporates multitasking support, sophisticated memory management, pipelined architecture, address translation caching, and a high-speed bus interface, all combined within the processor chip. While the 80386 represents a significant improvement over previous generations of Intel microprocessors, it retains software compatibility with older 16-bit microprocessors such as the 8086 and 80286 families.

The capacity of the 80386 processor is significant. Some figures are presented below, in comparison with 80286 processors:

- 4, 5, or 6 million instructions per second (corresponding to clock speeds of 16, 20, and 25 MHz).
- 4 gigabyte physical address space, compared with the maximum of 16 megabytes available on the 80286.

  Note: The 80386SX is an exception as it still is limited to a maximum of 16MB of physical memory. Internally the 80386SX is a full 32-bit processor but externally it has only a 16-bit data bus and a 24-bit address bus.
- 64 terabyte virtual address space, compared with the 1 gigabyte available on the 80286.
- Ability to handle memory objects from 1 byte to 4 gigabytes in size, compared to segments of 16 bytes to 64 kilobytes on the 80286.
- Paged memory management using 4 kilobyte pages, compared to the 80286 which offered only segmented memory management.

This chapter provides an overview of the 80386 processor architecture, in order to serve as a base for understanding the changes made in OS/2 Version 2.0. More detailed information about the 80386 can be found in:

- *Intel 80386 Hardware Reference Manual* (ISBN 1-55512-069-5)
- *Intel 80386SX Hardware Reference Manual* (ISBN 1-55512-105-5)
- *Intel 386 DX Programmer's Guide* (ISBN 1-55512-082-2)
- *IBM PS/2 Model 80 Technical Reference* (IBM P/N 84X1508).

A.1 Physical Characteristics

The 80386 processor consists of six dedicated units:

- Bus Interface Unit
- Code Prefetch Unit
- Instruction Decode Unit
- Execution Unit
- Segmentation Unit
- Paging Unit.

These individual units are connected by 32-bit buses and operate in parallel to provide a six-stage pipelined execution of instructions. This implies that up to six different instructions may be held concurrently within the chip, at different stages of execution. To further improve performance, the 80386 uses on-chip

caching and implements sophisticated memory management and bit manipulation (such as a 64-bit barrel shifter) in hardware.

The 80386 chip contains eight 32-bit general registers. To provide compatibility with the 8086 and 80286 processors, the 80386 provides the capability to use the lower-order 16 bits of these registers to represent the 16-bit registers used in these preceding processors. This is illustrated in Figure 27.

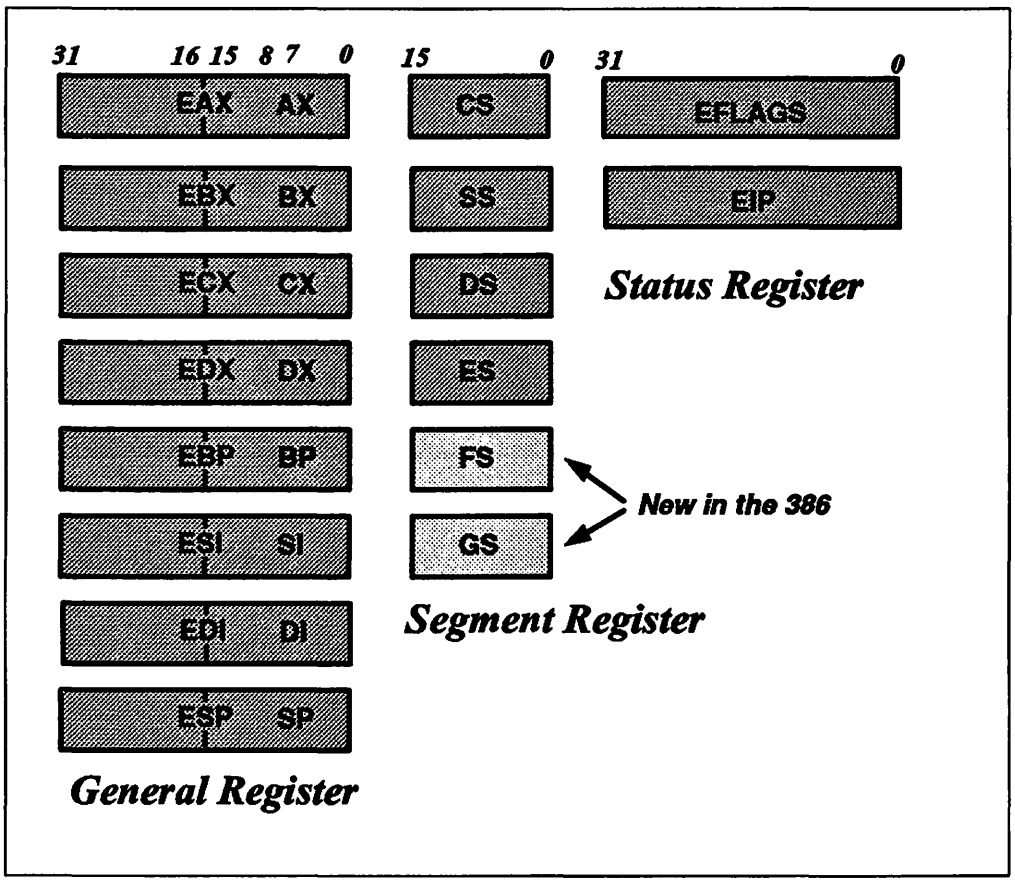


Figure 27. 80386 General, Segment, and Status Registers

Programs running in virtual 8086 mode may utilize the full register set of the 80386 (all 32-bit registers including the new FS, GS, debug, control, and test registers). The programs can also use instructions with 32-bit operands by overriding the operand size by including an operand size prefix on the instruction.

The 80386 also provides six 16-bit segment registers, which are used to contain segment selectors, thus providing support for the same segmented memory model used in the 80286 processor. Note that the FS and GS segment registers are new in the 80386.

The instruction pointer (EIP) register and the flags (EFLAGS) register are both 32-bit registers.

Applications written for the 80286 run unmodified on the 80386. This is because the 80286 instructions, addresses, limits, segment types, etc., are a subset of those available in 80386, and run in 16-bit mode automatically. The 80386 handles this very simply; if the upper word (16 bits) of a memory reference is zero, then that reference must be an 80286 reference.

The registers described above are available to application programmers, either directly using assembly language or indirectly through the use of higher-level programming languages. The 80386 processor provides a number of additional registers which are available for use by the operating system. These registers are protected and are not accessible by application programs:

- The 80386 provides four registers, which contain pointers to data structures used to implement the segmented memory model:
  - GDTR (global descriptor table register)
  - LDTR (local descriptor table register)
  - IDTR (interrupt descriptor table register)
  - TR (task register).
- Four control registers (CR0 to CR3) are used to hold pointers to data structures used by the paged memory model and for status information. These registers are new in the 80386 processor, since previous processors did not support the paged memory model.
- Eight debug registers (DR0 to DR7) are provided to aid in real-time system and application debugging. These registers are also new in the 80386.
- Two test registers (TR6 and TR7) are provided to allow for verification of the integrity of the translation lookaside buffer hardware used by the processor's paging subsystem.

A.2 Memory Addressing

The 80386 processor, like its predecessor the 80286, can operate in two addressing modes: **real mode** or **protected mode**. The memory addressing schemes used in each of these modes are described in the following sections.

A.2.1 Real Mode

When the 80386 is powered up or re-initialized via a hardware reset, the processor is set into real mode. In real mode, the 80386 effectively operates as a 16-bit processor. While in real mode the 80386 is emulating an 8086 processor. Program addresses correspond directly to physical memory addresses. Memory is addressed using the segmented memory model only (paging is not supported), and the system's physical address space is limited to 1MB of real memory. Virtual memory is not supported in real mode.

While running in real mode, the Intel 80386 does not implement any memory protection scheme. Real mode is not suitable for running multiple applications concurrently. When more than one program is loaded at a time, there is the possibility of one program accessing another program's memory. Also there is no protection for the operating system code, and application programs can overwrite operating system code and data.

Segment registers are used to supply the base address for each type of memory segment (DS - data segment, CS - code segment, SS - stack segment and ES - extra segment). Figure 28 on page 114 shows how a segment is addressed in real mode.

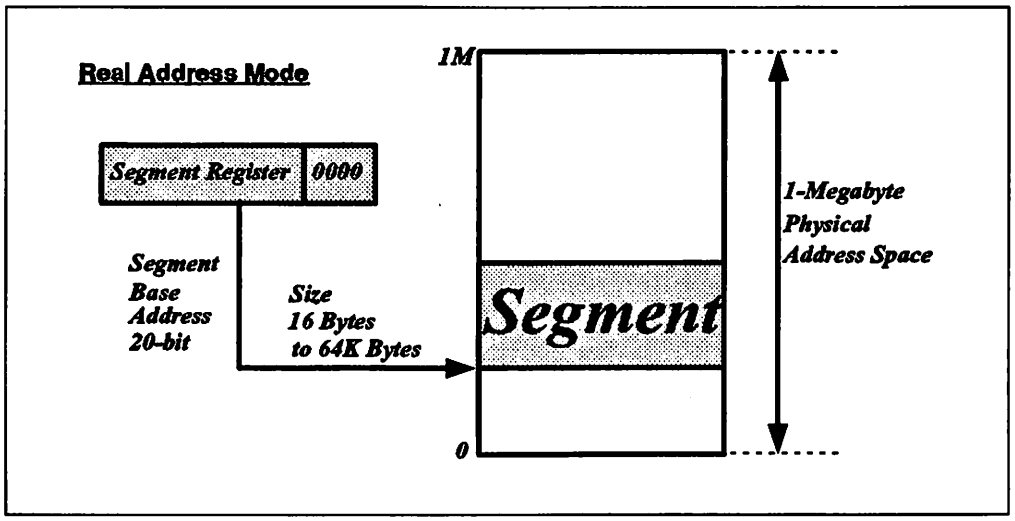


Figure 28. Real Mode Addressing

Each memory reference consists of a 16-bit **segment address** and a 16-bit **offset**. The processor automatically adds four binary zeros to the segment selector value (equivalent to multiplying by 16) to obtain a segment base address in memory. Thus, a segment may start on any 16-byte boundary within the 1MB physical address space.

The required memory location within the segment is determined by adding the offset to the segment base address. Since the offset is 16 bits in length, the maximum offset (and therefore the maximum size of a segment) is 64KB.

A.2.2 Protected Mode (Segmented Memory Model)

When the 80386 is switched to protected mode by a software command, the full 32-bit capabilities of the processor are enabled, and the system's physical address space is increased to 4GB. Since virtual memory support is enabled in protected mode, the virtual address space visible to an application increases to a theoretical maximum of 64 terabytes.

Note

In OS/2 Version 2.0, the process address space is limited to 512MB in order to reserve memory for operating system use and to retain full compatibility with applications written for previous versions of OS/2, which used 16-bit addressing.

Each process occupies a separate logical address space, and the 80386 provides full memory protection between the address spaces of different processes, thereby preventing an application from inadvertently accessing and/or corrupting memory used by another application. Note, however, that under OS/2 Version 2.0, multiple threads may be created within a single process, and dispatched independently by the operating system. These threads share a common address space, and it is therefore the responsibility of the application developer to ensure correct behavior of and synchronization between multiple threads within a single process.

In protected mode, the 16-bit segment registers are used in conjunction with a 32-bit offset to give a 48-bit selector:offset pointer. The segment registers no

longer contain the segment base address; rather, they contain an index into a **descriptor table**. The entries in the descriptor table each point to the start of a segment in physical memory.

Of the 16 bits which make up the segment selector, two bits are used to specify the privilege level of the segment, and one bit is used to select between the **global descriptor table** (GDT) and a **local descriptor table** (LDT). The GDT is used by the operating system or privileged software to maintain control over all segments within the system. A unique LDT is maintained for each process and used to control only the memory segments used by that process. In this way, each process is prevented from accessing the memory used by another process. The remaining 13 bits are used as the index into the appropriate descriptor table. A logical address which consists of a selector and offset is converted into a 32-bit linear address by extracting the segment base address from the descriptor table by using the selector as the index, then adding the offset to the base address. With paging disabled, the linear address equates the physical memory address. This address translation operation is shown in Figure 29. The virtual address space, which can consist of up to 16,383 segments of 4GB each, is thus mapped to the physical 4GB address space. The descriptor also contains access information for the segment along with the segment size. The access information and segment size is used by the processor to implement memory protection.

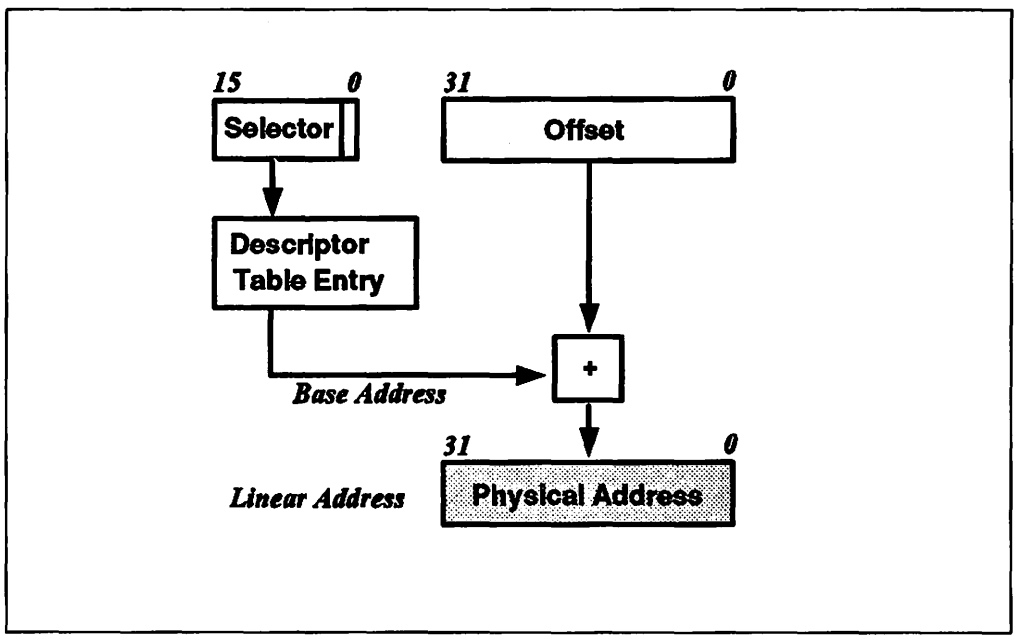


Figure 29. Protected Mode Addressing - without Paging

The maximum allowable value for the offset, and thus the maximum size of a segment, is defined by two things. Each entry in a descriptor table contains a 20-bit limit field. These 20 bits allow a maximum segment size of 1MB, using the byte as the unit of size.

However, the descriptor table entry also contains a **granularity bit**, which specifies that either the byte or the *page* may be used as the unit of size in the limit field. When using page granularity, the 20 bits in the limit field represent a multiple of 4KB, allowing a segment size of 4KB to 4GB.

A.2.3 Protected Mode (Flat Memory Model)

The 80386 is able to address up to 4GB in a single segment. This is a large address space. It may, therefore, be desirable *not* to use the segmented memory model, but simply to map the entire system memory as a single linear address range. While the 80386 does not have a mode bit for disabling segmentation, the same effect can be achieved by mapping the stack, code, and data spaces to the same range of linear addresses. When this is done, the 32-bit offsets used by 80386 memory references can cover the entire linear address space.

OS/2 Version 2.0 uses this technique to implement a flat addressing model. The operating system internally creates a single code segment and a single data segment, with the base address of each segment selector set to zero, and a segment size of 4GB. This segment selector is loaded into the CS, DS, ES, FS, GS, and SS registers; hence these registers all point to the same memory range. The selectors are allocated within the GDT. Offsets within the segment are actually offsets within the 4GB global address space, and are therefore equivalent to linear memory addresses.

The advantage of using such a technique is that it greatly simplifies memory management within an application, since the application developer no longer need be concerned with the internal implementation of data structures as segments with a defined maximum size. The use of a flat memory model also facilitates migration of the operating system and application code to other hardware platforms, since the code is not explicitly designed around the segmented memory model. Application performance is also improved since there is no longer the need for continually changing the contents of segment registers. All address references are near references as with a 32-bit offset, you can access the entire address space.

A.3 Paging

In addition to the segmented memory management offered on the earlier 80286 processors, the 80386 provides a paged memory model. This is an optional function of the 80386, and there are no direct performance implications of an operating system choosing not to use paged memory. However, the paged memory model provides significant performance benefits when running large applications which make extensive use of virtual memory.

Under previous versions of OS/2, the smallest unit of memory (for memory management purposes) was the segment, since the operating system was designed to execute on the 80286 processor and use the segmented memory model. With the 80286, segments may vary in size between 16 bytes and 64KB; there is therefore a danger of having a large amount of free memory which is fragmented into small, discontiguous units.

Previous versions of OS/2 managed this by moving segments within real storage to create a larger free space, and by swapping unused segments to disk until they are required. This entailed a high degree of overhead for the operating system. With an 80386 processor, however, segments may be up to 4GB in size, and the overhead could potentially result in an unacceptable performance impact, particularly for applications with very large segments. Also there is the problem of having sufficient physical memory to load a complete segment, when the segment is very large.

In order to avoid this situation, the 80386 processor provides a paged memory model, implemented in hardware through a dedicated paging unit included on the processor chip. A page is a 4KB unit of contiguous memory, and replaces the segment as the unit of granularity for memory management, including swapping to and from disk. Note that paging is available in protected mode, in conjunction with both the segmented and flat memory models.

Using the paged memory model, an application makes a memory reference in the normal way, using either the segmented memory model or the flat memory model. The segmentation unit in the processor automatically resolves the reference into a 32-bit linear address. However, this linear address does not represent a physical address, but is made up as follows:

- The high-order 10 bits of the field are used as an index into a page directory table. The entry in this table in turn refers to the base address of a page table.
- The next 10 bits of the field are used as an index into the page table referred to by the page directory entry. The entry in the page table provides the physical base address of a 4KB page.
- The lower-order 12 bits of the field are used as an offset within the page referred to by the page table entry.

OS/2 Version 2.0 maintains a single page directory for the entire system, but a separate set of page directory entries for each process present in the system. The physical address of the page directory is held in the CR3 control register. This is known as the page directory base register (PDBR) in OS/2 Version 2.0. When a task switch takes place, the page directory entries, for the process being scheduled, are copied into the page directory.

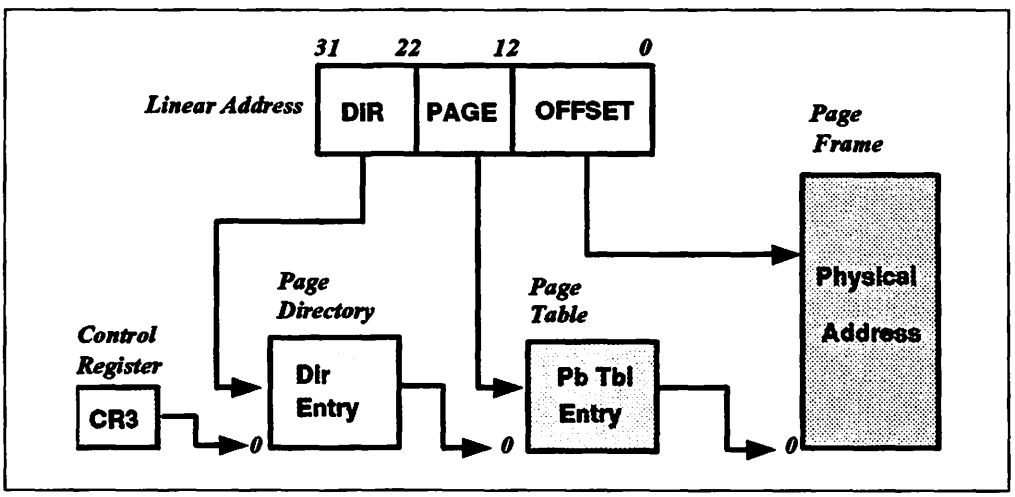


Figure 30. Protected Mode Addressing - with Paging

Both the page directory and page tables contain 32-bit page specifiers. The page directory and page tables are themselves contained within single pages, and may therefore contain a maximum of 1024 entries. Each page directory can hence access up to 4GB of storage, which is the maximum physical address space of the 80386.

Pages may be shared between processes by defining them in the page tables of more than one process. Note that this is done at the page table level rather than the page directory level, in order to share *only* the individual pages required.

In order to further reduce the overhead involved in looking up page references, the 80386 also provides a hardware-based address caching mechanism for paging information. This is known as the translation lookaside buffer (TLB). The TLB contains the physical addresses for the 32 most recently used pages, and therefore allows very fast access to these pages since it is no longer necessary to read entries from the page directory and a page table, which are held in system memory. Use of the TLB is handled entirely within the paging unit, and is not visible to software. The operating system must, however, ensure that the buffer is flushed whenever the PDBR is updated or an entry in either the page directory or a page table is changed. This ensures that the TLB contents are kept in step with the operating system maintained tables.

A.4 Task Switching

A significant function of the 80386 processor is its ability to support a multitasking environment. While much of the multitasking support in the 80386 is similar to that provided in the previous 80286 family, multitasking will be discussed here for those who may not be familiar with its implementation.

In a multitasking system, when one task is suspended and control is passed to another, the processor state must be stored so that when the suspended task regains control, it may resume normal operation where it left off.

The 80386 processor architecture defines a special type of memory structure known as a Task State Segment (TSS). The processor uses a specific fixed format to store task-related control information in the TSS, in order to provide high-performance task-switching operations with complete isolation between tasks. A separate TSS is maintained for each task in the system. Each TSS contains:

- General registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI)
- Segment registers (ES, CS, SS, DS, FS, and GS)
- Flags register (EFLAGS)
- Instruction pointer (EIP)
- Selector for the TSS of the previous task
- Selector for the task's LDT (static)
- Logical addresses of the stacks for privilege levels 0, 1, and 2
- The T-bit (debug trap bit)
- Base address for the I/O permission bit map.

A special segment descriptor is used for each TSS, and appears only in the system's Global Descriptor Table (GDT), since TSSs are not available to applications. The Task Register always contains a pointer to the TSS for the current task.

Task switching may occur as the result of either an interrupt or of executing an instruction that explicitly transfers control. A task switch may be achieved in one of four ways:

- The current task executes a JMP or CALL to a TSS descriptor
- The current task executes a JMP or CALL to a task gate (a special type of segment descriptor)

- An interrupt or exception indexes a task gate in the IDT (Interrupt Descriptor Table)
- The current task executes an IRET instruction with the NT (next task) flag set. The selector for the previous task is always stored in the current TSS, thus providing the means to return control to the previous task.

During the task switch operation, the processor saves the contents of the current registers in the TSS of the current task. The selector of the next TSS is then loaded into the Task Register. This selector references an entry in the GDT, which contains the physical address of the TSS. The values in the TSS are then loaded into the processor's registers, and control information is loaded into the segment registers from the GDT and the process's LDT. The processor is then ready to continue execution of the new task.

To create a new task, the operating system initializes a TSS to the appropriate initial values. The operating system then determines when to start the task, and accomplishes this by simply switching from the current task to the new one.

OS/2 V2.0 only makes minimal use of the TSS mechanism. The use of the flat memory model and the way in which OS/2 V2.0 implements paging makes a large part of the data stored in the TSS redundant. Consequently OS/2 V2.0 implements its own task switching model which optimizes switches between threads in the same or different processes. Also allocating a TSS for each thread in the system would use a large amount of storage. OS/2 V2.0 uses a single TSS for effecting transitions between the different privilege levels, at which tasks present in the system run. Privilege levels are described in the A.5.3, "Privilege Levels" on page 120.

A.5 Protection

The 80386 processor implements five different types of protection for tasks executing within the system. These are:

- Type checking
- Limit checking
- Privilege levels
- Restriction of procedure entry points
- Restriction of instruction set.

Each instruction and memory reference is checked by the hardware to ensure compliance with the protection rules prior to execution. For memory references, checking is performed during the address translation process, and is applied to both segmented and paged memory models. Protection parameters are stored in the segment descriptors or in the Page Directory and Page Table entries.

A.5.1 Type Checking

With each descriptor, there is a *type* field which is used to distinguish between the different descriptor formats. This field also specifies the intended use of a segment. For example, the allowable types for data segments are:

- Read-Only
- Read/Write

and for code segments:

- Execute-Only
- Execute/Read.

The type field, therefore, ensures that segments are only used in ways for which they are intended. For example:

- The code segment (CS) register may only be loaded with the selector of an executable segment.
- Selectors of executable segments that are not defined as readable cannot be loaded into a data segment (DS, ES, FS, and GS) registers.
- No instruction may write into an executable segment.
- No instruction may write into a data segment unless that segment is defined as *Read/Write*.
- No instruction may read an executable segment unless that segment is defined as *Execute/Read*.

So far this discussion applies to segment level checking. When paging is enabled, there is also checking done at the page level. Pages may be of two types:

1. Read-only access
2. Read/write access.

Segment protection is evaluated first, then page protection is checked. It is possible to have a large segment which is both readable and writable and to have pages within it which are only readable. At the page level, the execute-only attribute does not exist.

A.5.2 Limit Checking

The limit field in each segment descriptor is used by the processor to prevent a program addressing memory outside the segment through the use of an overly large offset value. During address translation, the offset value specified in the memory reference is compared with the limit field, and an exception is generated if the offset is larger than the limit for that segment. The limit field in general prevents errors in one program from corrupting other programs' code or data areas.

A.5.3 Privilege Levels

The 80386 implements a four-level protection mechanism. Level 0 is the most privileged, and level 3 is the least privileged. The privilege level is assigned on a segment basis, and therefore applies to both code and data. The four levels may be visualized as concentric "rings," with the most privileged segments in the center.

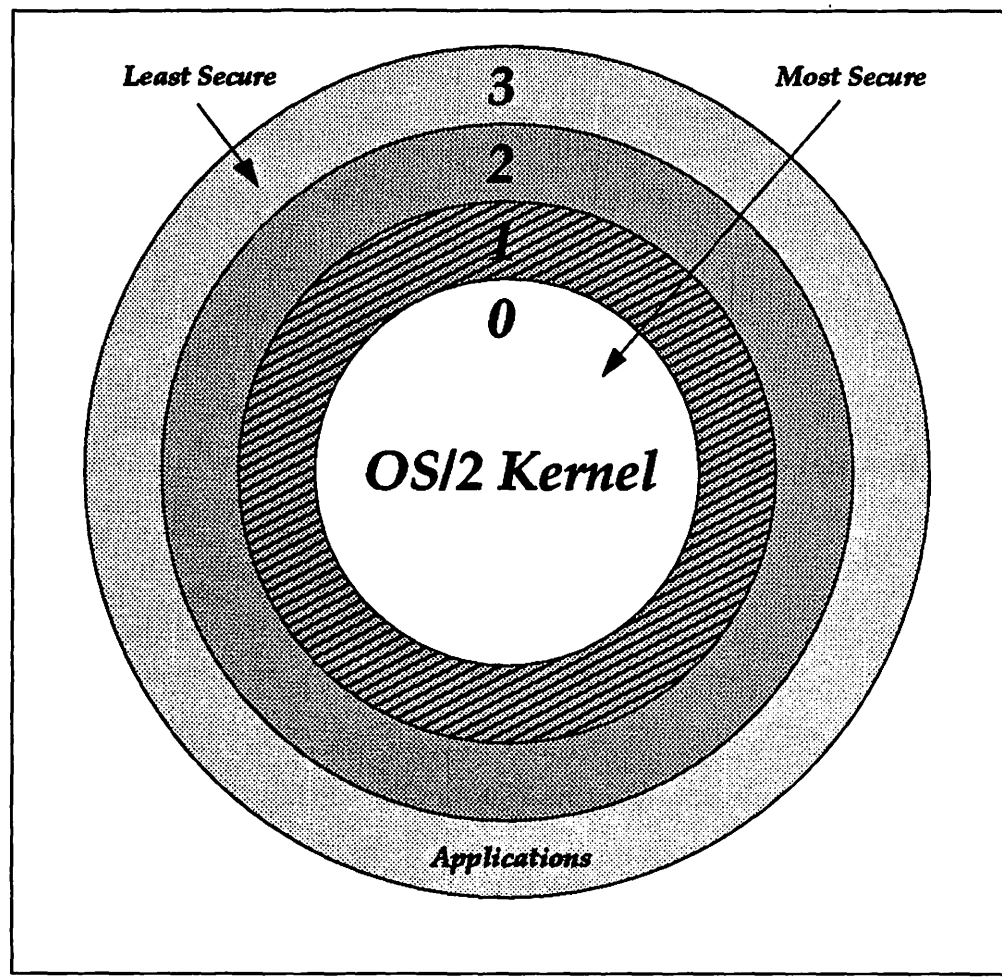


Figure 31. 80386 Ring-Oriented Privilege Scheme

All code and data segments in the system are assigned a privilege, which is stored in the segment descriptor. At any one moment, a task executes only on one of the four rings:

Ring 0 This is the most privileged type of segment, and code executing in this ring may use all protected mode processor instructions. This ring is used by those routines in an operating system which are essential for resource allocation and control. This part of the system is often referred to as the kernel or nucleus.

Ring 1 This is the second most privileged ring and is normally used for the remainder of the operating system routines and for the input/output support routines. Note that ring 1 is *not* used by OS/2 Version 2.0.

Ring 2 This ring is typically used as the application services level. It should be used for routines that do not belong to the operating system, but should still be protected from user code. Communications support and database management programs are good examples.

Ring 3 This is the least privileged ring and is typically assigned to user application code and data.

A task executing in one ring cannot access data in a more privileged ring (for example, ring 3 cannot access data at ring 1), nor can it invoke a procedure in a less privileged ring (for example, ring 1 cannot invoke ring 3). Thus, both access to data and transfer of control are restricted in appropriate ways. The processor

interprets the protection parameters and automatically performs all the checking necessary to implement this protection.

Although at the segment level there are these four levels of privilege, at the page level there are only two privilege levels:

1. Supervisor level, for the operating system, privileged programs, such as device drivers, and system data including page tables
2. User level for application code and data.

The privilege levels used for segmentation are mapped into the page level privilege levels. Tasks running in ring 0, 1 or 2 are all assumed to at supervisor privilege level.

A.5.4 Restriction of Procedure Entry Points

To achieve transfer of control between procedures on different privilege levels, a special descriptor type called a **gate** is provided. Programs wishing to transfer control call the gate by specifying the segment base address of the gate, rather than transferring control directly to the required procedure. Under OS/2 Version 2.0, where a single segment is used to implement the flat memory model, the gate is called by specifying its offset within the process address space.

The four types of gates are CALL gates, TASK gates, INTR (Interrupt) gates, and TRAP gates. The routine invoked when the gate is called simply redirects control to a new address which contains the privileged routine to be executed.

From the program's point of view, this is no different from transferring control to another code segment, since the calling instruction simply regards the gate as another procedure. However, it effectively isolates the calling procedure from the called procedure, and since only the entry point address of the gate is supplied in the calling instruction, the calling procedure has no access to any point other than the defined entry point of the called procedure.

Calls are verified to ensure that they satisfy two conditions:

1. The call must enter the called procedure at the beginning of that procedure; this is normally ensured by the gate descriptor itself, which supplies the necessary offset to the entry point.
2. The privilege level of the called procedure must be the same as that of the gate descriptor.

A.5.5 Reserved Instructions

Certain processor instructions are reserved for execution only by the operating system, and may therefore execute only at privilege level zero. Such instructions include HLT (Halt Processor), LGDT (Load GDT), and LTR (Load Task Register).

In addition, some I/O instructions are restricted:

1. The IOPL field in the EFLAGS register defines whether or not the current task has the right to use I/O-related instructions
2. The I/O Permission Bit Map in the TSS determines whether the current task may use ports in the I/O address space.

A.6 Interrupts

When the processor is running in protected mode, interrupts are not vectored from the base of memory. Instead, each interrupt has a code which is used as an index into an **Interrupt Descriptor Table** (IDT), the base address of which is contained in the Interrupt Descriptor Table Register. There may be up to 256 interrupt and exception codes, generated by devices or by software.

At system initialization, the IDT is loaded into memory by the operating system, and its location is stored in the IDT register. Each descriptor in the IDT specifies the address of the interrupt handler routine, which will service interrupts with that code.

There are three types of gate descriptors in the IDT:

- Interrupt gate descriptors
- Trap gate descriptors
- Task gate descriptors.

For interrupt and trap gates, the descriptor in the IDT contains the selector of the gate, and therefore points indirectly to a procedure that will execute within the current task, since the selector within the gate procedure points directly to an executable segment descriptor in the GDT or the current LDT. This takes place exactly as if the 80386 were calling a procedure within the current application.

For the task gate, however, the selector within the gate points to a TSS descriptor in the GDT. Invoking the task gate, therefore, causes a task switch to occur. There are certain advantages to the use of a task gate, since it allows a program to pass control to a higher privilege level, and the application may therefore invoke operating system routines to process interrupts and exceptions. In addition, the new task may be given its own LDT to prevent it from accessing memory used by the current task, and the TSS of the current task is automatically saved.

However, there are also performance implications in using task switching. Interrupt handling through task switching requires approximately 15 microseconds on a 20 MHz 80386, while switching to a procedure within the current task takes about 3.6 microseconds. But, the advantages of having the operating system manage exceptions (smaller application code, greater portability, standard exception handling) usually outweigh the slight performance penalty.

A.7 Input/Output Processing

I/O addressing on the 80386 may be performed either by issuing specific I/O instructions to the I/O address space, or issuing general-purpose memory manipulation instructions to memory-mapped I/O.

The I/O address space is separate from the linear physical memory and the I/O instructions do not go through the segmentation or paging hardware. The I/O address space is 64KB in size. It may be mapped in various ways, for instance: 64KB of individually addressable 8-bit ports, 32KB of 16-bit ports, 16KB of 32-bit ports, or any combination of the above up to the maximum allowed 64KB. The processor can transfer 32 bits of data at a time to a device located in the I/O address space, using the IN, OUT, INS, and OUTS commands.

The I/O address space has two protection mechanisms:

1. The **I/O privilege Level** (IOPL) field in the EFLAGS register controls access to the I/O instructions.

   The IN, INS, OUT, OUTS, CLI, and STI instructions are only allowed to execute if the CPL (Current Privilege Level in the CS descriptor for the active task) is less than or equal to the value of the IOPL field.

   Only system code (privilege level 0) can change the IOPL value.

2. The **I/O permission bit map** in the active TSS controls access to individual ports in the I/O address space.

   There is one bit for each 8-bit port in the I/O address space, which means that the I/O permission bit map could be up to 64 kilobits (8KB). If a task references an I/O port and the corresponding bit is on, the processor signals a general protection exception. The exception can then be handled by the system software to initiate an exception handling procedure within the current task, or to initiate a new task, which will redirect the I/O.

   By changing bits in the I/O permission bit map of different tasks' TSSs, an operating system can allocate ports to tasks and avoid having two tasks use the same port concurrently.

A.8 Virtual 8086 Mode

The 80386 processor supports concurrent execution of one or more 8086 programs within the protected mode environment. There is no longer a need for the processor to switch back to real mode in order to simulate an 8086 machine.

An 8086 program runs in protected mode as part of a **virtual 8086 task**. Virtual 8086 tasks are able to take advantage of the 80386 hardware support for multitasking, offered in protected mode. Virtual 8086 tasks may execute concurrently with one another and with other protected mode tasks in the system.

The purpose of the virtual 8086 task is to form a **virtual machine** for running programs written for the 8086 processor. A complete virtual machine consists of the 80386 processor support, plus additional support from operating system software:

- The hardware provides a set of virtual registers (implemented through the TSS), a virtual memory space (the first 1MB of the 32-bit linear address space) and directly executes all instructions that deal with these registers and with this address space.

  While running in virtual-8086 mode the processor does not treat the contents of the segment as an index into a descriptor table, but the linear address is formed in exactly in the same way as it is done by the 8086 processor. With paging, enabled the address is then mapped to a physical address by the address translation hardware. When paging is not being used, the linear address is the physical address. This means that when there is a requirement for multiple virtual-8086 tasks, paging must enabled. Figure 32 on page 125 shows the way in which the memory used by virtual 8086 machines is mapped into the system's physical address space.

- The operating system software controls the external interfaces of the virtual machine (I/O, interrupts, and exceptions). In the case of I/O, the operating system can choose either to emulate I/O instructions or to let the hardware execute them directly.

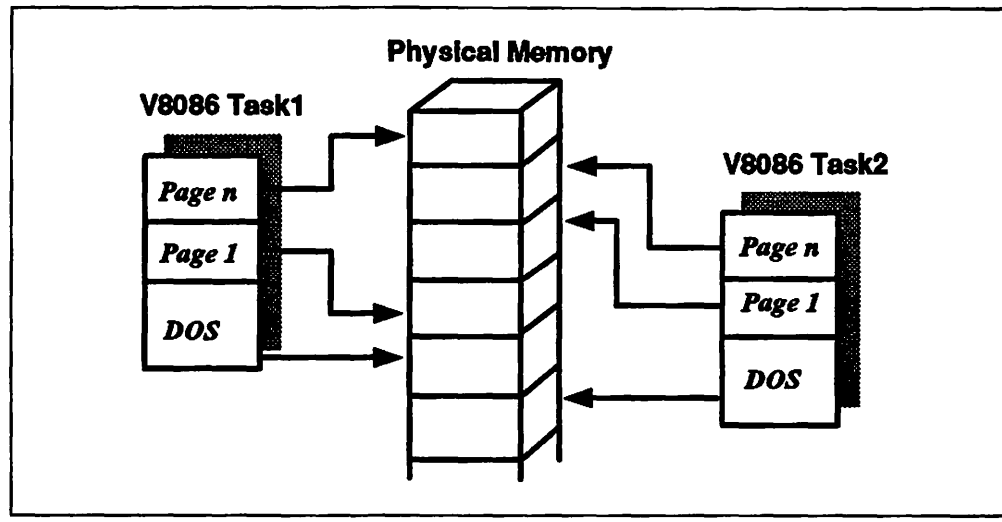


Figure 32. Virtual 8086 Environment - Memory Management

Virtual 8086 tasks execute at privilege level 3 (lowest) and are subject to all of the protection checks defined in protected mode, thereby preventing an ill-behaved application from accessing and potentially corrupting memory used by other tasks in the system.

All I/O is normally handled through the I/O permission map in the 80386 TSS for both virtual 8086 applications and other protected mode applications. This means that any call to I/O services generates an exception which is trapped by the 80386 and may then be handled by the operating system. Any unauthorized calls may be trapped within the operating system, thus preventing an ill-behaved application from "hanging" the system.

In addition, the 80386 paging hardware allows virtual 8086 tasks to share segments.

A.9 Numeric Coprocessor Utilization

The 80386 processor may operate in conjunction with, and utilize the features of either the Intel 80287 or 80387 numeric coprocessors. When the system is initialized, the presence of a numeric coprocessor, and its type if present, is checked by the 80386. If an 80287 coprocessor is detected, the 80386 automatically converts all memory transfers to 16-bit format. If an 80387 is detected, it is used in 32-bit mode, thereby utilizing the full potential of both the 80386 and 80387.

Note

IBM does not support or recommend the use of 80287 numeric coprocessors in 80386-based systems. For a list of supported numeric coprocessors for each system unit, readers should refer to the appropriate IBM Product Announcement for that system unit.

A.10 Multi-Processing

The 80386 supports the Intel 80287 and 80387 numeric coprocessors. Support is also provided within the instruction set for multiple 80386 processors within the same system, sharing memory and other resources. This support is provided through the LOCK prefix instruction. When specified in conjunction with another instruction, the LOCK prefix instruction ensures that the locking processor has exclusive use of the requested resource.

Only a few 80386 instructions can be used with the LOCK prefix instruction. It is typically used to prefix instructions like BTC (Bit Test and Complement) where it locks the area of memory defined by the destination operand for as many cycles as necessary to update the entire operand.

In several instances, the processor itself automatically locks activities on the data bus. For example, when acknowledging interrupts, switching tasks, loading descriptors from the LDT to the segment selector, and updating the page table ACCESS and DIRTY bits, the required memory pages are locked since these are highly critical operations.

The 80386 includes on-chip memory caching to improve performance. The processor must therefore allow for the case where data in shared memory is modified and where that data is currently recorded in a cache on another processor. In such situations, the 80386 employs an interprocessor interrupt to let other processors know when such a change has been made.

This is normally done by using one of the physical address pins on the chip, and having the receiving processor implement a task switch when it receives this signal. The task switch clears the system registers, reloads the new descriptors and invalidates the memory cache in the processor.

Note that by changing the function of one of the addressing pins, however, the physical addressing capability of the processor is reduced to 2GB.

A.11 The Intel 80486 Processor

Some of the most recent PS/2 machines use the latest Intel microprocessor, the 80486. The 80486 offers more processing power and also some functional extensions over the 80386. The 80486 microprocessor subsystem has the following characteristics:

- 32-bit addressing
- 32-bit data interface
- Extensive instruction set, including string I/O
- Hardware fixed-point multiply and divide
- Three operating modes:
  - Real address mode
  - Protected virtual address mode
  - Virtual 8086 mode
- 4GB physical address space
- Eight general-purpose 32-bit registers
- 64TB virtual address space

- Internal 8KB, set-associative instruction cache with controller
- Internal 80387 numeric coprocessor.

  Note: The 80486SX is an exception, since it does not provide a built-in 80387 numeric coprocessor

The 80486 microprocessor is compatible with the 80386 in the following areas:

- Real address mode
- Protected virtual address mode
- Virtual 8086 mode
- 80386 paging mechanism
- All published 80386 instructions
- All published 80387 instructions.

The major differences between the 80386 and 80486 processors are in the execution speed of instructions, and the fact that the 80486 has six new processor instructions to control the operation of the internal 8KB instruction cache.

In addition, the 80486 processor performs certain operations in a different manner. For example, flushing the transaction lookaside buffer in an 80386 processor may only be done for the entire TLB in a single operation. The 80486 provides a facility for selectively flushing the TLB.

Appendix B. Micro Channel Architecture and SCSI

This section of the document explains the basic principles of the IBM Micro Channel architecture and its usage. The principles of the Small Computer Systems Interface (SCSI) protocol for I/O device communication are also covered in this section of the document. This information is presented in order to provide the reader with a better understanding of the way that this technology is utilized by OS/2 Version 2.0.

B.1 Micro Channel Architecture

In "conventional" personal computer architectures such as the original IBM PC and PC AT, the system bus formed the data path, by which hardware components exchanged information with one another. The bus operated under the direct control of the system's main processor, and could handle only a single task at any time.

The Micro Channel architecture defines a set of specifications for a high-speed data "highway" connecting the system processor or processors, memory, I/O devices, and hardware adapters. The main feature that distinguishes the Micro Channel from the older PC bus is that the system processor does not have exclusive control of it but other processors and intelligent adapter cards, which are connected to the channel, can take charge of it and initiate data transfers across it. The mechanism, by which control of the Micro Channel is shared amongst the competing subsystems, is known as *arbitration*. The Micro Channel consists of a number of buses, controlled by the bus arbitration unit, which operate independently of the system's main processor. The main components of the Micro Channel are:

- **Arbitration Bus**

  The arbitration bus and its associated signals are used by the bus arbitration unit to prioritize and resolve up to 16 concurrent requests by intelligent devices (known as *masters*) for control of the channel. The system processor has the lowest priority level, leaving 15 levels available for other processors and intelligent devices in the system. Although the system processor is defined to have the lowest priority level, it always "owns" the channel whenever the channel is not being used by another device. The arbitration unit resolves contending requests, and selects one device as the temporary owner of the channel; this device is then known as the **controlling master**. The controlling master may then perform a single data transfer or if the channel had been requested in "burst mode", multiple data transfers. For "burst transfers", the controlling master owns the channel until the transfer is complete or another arbitration participant requests ownership of the channel. In such cases the controlling master must relinquish ownership of the channel within 7.8 microseconds.

  The arbitration controller has a *fairness* algorithm built into it to ensure that subsystems with high priority do not monopolize the channel at the expense of lower priority subsystems. There is also a feature known as *pre-emption*, which allows a subsystem with an urgent requirement to request and be given control of the channel even though another subsystem is currently using it.

- **Address Bus**

  The address bus and its associated signals are used by the controlling master to select a *slave* to be the source or target for a data transfer. The Micro Channel addressing consists of two separate address spaces:

  - The **I/O address space** consists of 64KB (65535) I/O addresses. 16 lines of the address bus plus the state of certain associated control signals signifies the I/O port to be read or written.
  - The **memory address space** may be as large as 4GB (4,294,967,296 bytes) for a machine capable of 32-bit addressing. Only PS/2 Models 90 and 95 are currently capable of 32-bit addressing. All other system units in the PS/2 range use 24-bit addressing, which allows addresses up to 16MB (16,777,216 bytes).

- **Data Bus**

  The data bus is used to transfer 8, 16, 24, or 32 bits of data between two masters or between a master and a slave. The associated control signals indicate the width and direction (read or write) of the transfer.

The Micro Channel architecture defines the physical properties of the circuits and all the timings and signal sequences of signals on these circuits.

The Micro Channel architecture is *extensible* in that it has been possible to include new features in the architecture while maintaining *compatibility* with existing devices. All Micro Channel devices are expected to support certain basic functions. Mechanisms are provided which allow a device that supports an optional feature to communicate its capability to a partner device, with which it is exchanging data. If both devices support the feature then it is enabled for the data transfer operation, if appropriate. If one device supports a feature and the other device does not, they may still communicate with one another. Obviously that feature cannot be used during the data transfer operation.

B.2 Micro Channel Participants

There are two basic types of devices, which may exist on and communicate via the Micro Channel:

- A *master* is an intelligent device, which may contend for control of the channel.
- A *slave* is a unintelligent device, which merely acts as the source or target of a data transfer, in conjunction with a master.

Figure 33 on page 131 shows the interaction between masters and slaves over the Micro Channel. The master devices are of three kinds:

- The **system master** is the processor provided with the system hardware, and is thus also known as the system processor. The system master assumes ownership of the channel when no other master has arbitrated for and won control of it. The system master is, therefore, also known as the **default master**.
- The **DMA controller** is typically provided on the system board. It supports multiple independent DMA channels, each allowing the attachment of a DMA slave. Each DMA slave is allocated its own dedicated channel. The DMA controller manages the transfer of data between a DMA slave and a memory slave, and supports burst mode data transfer if the DMA slave requests it. The DMA controller does not arbitrate for the channel, but requires the DMA

slave to do the arbitration. Once this is done, the data transfer is completed independently of the system processor.

- The **bus master** is an intelligent device or adapter on the Micro Channel and is typically an advanced adapter, which functions as a subsystem within the system. It arbitrates for the channel and manages the data transfer to or from an I/O slave or a memory slave. The bus master is described further in B.2.1, "Bus Master Adapters."

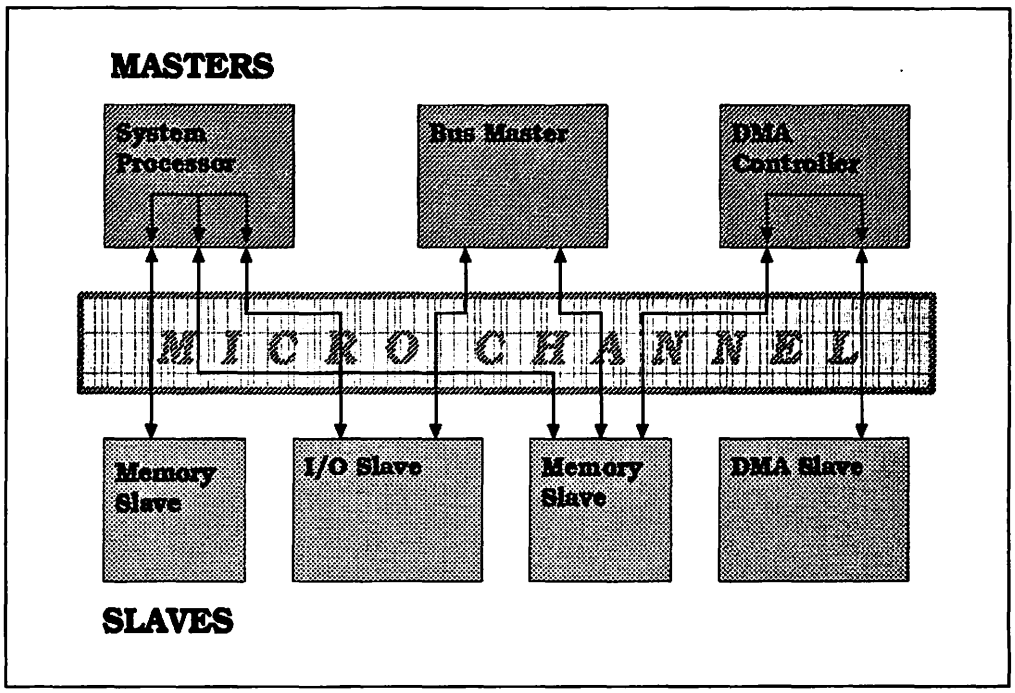


Figure 33. Micro Channel Participants and Data Transfer Paths

Slave devices on the Micro Channel may be of three kinds:

- The **I/O slave** is selected via its address in the I/O address space. The system processor or a bus master is required to actually perform the data transfer.
- The **memory slave** is selected via its address in the memory address space. Any master may perform the data transfer. The memory may reside on the system board or on an adapter on the Micro Channel. It can also be non-system memory (such as memory-mapped I/O) on an adapter, used for communication of information between the system and the adapter.
- The **DMA I/O slave** contends for control of the Micro Channel on behalf of the DMA controller, and is thus an exception to the normal rule that slaves do not contend for control of the Micro Channel. It relies on the DMA controller to be the controlling master and manage the data transfer between the DMA slave and memory slave.

B.2.1 Bus Master Adapters

The use of bus master adapters avoids the need for the system processor to become involved in data transfers between adapters in the system. For example, a bus master LAN adapter may interact with the disk subsystem to service I/O requests on a LAN server, while the system processor continues to process other work.

Bus master adapters provide advantages over other "intelligent" adapters such as DMA adapters, since a bus master is typically directly involved in the data transfer (as either source or destination) and thus requires only a single operation to transfer the data, while a DMA adapter requires two (one read and one write). In addition, the DMA controller used in current IBM PS/2 system units is only capable of 24-bit addressing, while bus master adapters, such as the SCSI adapters, utilize the full 32-bits available on the address bus on the PS/2 models 90 and 95.

A bus master may randomly address memory and memory-mapped I/O devices. This makes it ideal for paged environments and enables the bus master to execute chains of work, addressing different memory areas and devices without support from the system processor. The bus master adapter is regarded as a subsystem and can work completely asynchronously to the system processor and other bus masters. Where necessary a bus master may communicate directly with another bus master.

B.2.2 DMA Adapters

The DMA adapter frees the processor from having to move data between an adapter and system memory. The DMA slave adapter arbitrates for control of the MCA. Once control is given, the DMA controller uses the DMA channel allocated to the DMA slave adapter to transfer the data.

The Micro Channel supports up to 15 DMA units, which may use the burst mode data transfer capability of the Micro Channel. The DMA controller implemented in current PS/2 machines supports transfers of up to 64KB segments of contiguous data and is able to address 16MB of memory. In the case of systems that have more than 16MB of memory, DMA transfers can only take place to memory locations below the 16MB line.

The major limitation is that the data must be contiguous. For a system using paged memory (with 4KB pages) this introduces an overhead through the need to reserve contiguous blocks of memory. This makes the system DMA controller best suited for records shorter than or equal to the page size.

B.2.3 Simple Adapters

Simple adapters are totally dependent on support from the main processor or another master. This approach to adapter design was common in single task systems, when the processor was dedicated to one adapter function at a time. In such cases, the most economic approach is to use the minimum of logic on the adapter card, and depend on software support. Adapters, which use these techniques, are:

- **Polled adapters**

  With polled adapters, the processor must periodically request information, or poll, the adapter registers for status of the attached device. This can consume much of the processor's power.

- **Interrupt per character adapters**

  With these adapters, the processor is free to perform other duties until an interrupt is received from the adapter. The overhead of servicing the interrupt is several hundred instructions to save and restore the environment.

  Devices with high data transfer rates (for example, fast communication adapters) cause high interrupt rates and the overhead could then use up

much of the available processor power. This is even more of an issue in a multitasking environment where the situation is more complex and the overhead for interrupt processing is typically higher.

- **Memory mapped adapters**

  This type of adapter shares a segment of memory (on the adapter) and if combined with the use of interrupts, can be much more efficient than the interrupt per character type. However, the adapter is still dependent on the processor to move data between the adapter's storage and a dynamically allocated buffer in system memory. With the arbitration overhead and large amounts of data (for example, bitmaps for a graphics adapter), this may be a burden on the processor.

  The biggest disadvantage with this type of adapter is that there is a limited number of fixed assignments for shared I/O memory. This limits the number of possible configurations and designs.

B.3 Data Transfer Modes

The attainable speed of data transfer between two participants on the IBM Micro Channel is affected by a number of factors:

- Time required to gain control over the channel (arbitration)
- Time required for each data transfer
- Capability to perform burst mode transfers
- The width of the data path (8, 16, 32, or more bits)
- Type of transfer cycle used (basic or matched-memory cycle).

The arbitration cycle required to gain control over the Micro Channel takes a minimum of 300 ns. After that time, the device can use the channel until the transfer is completed or a pre-empt signal is raised by another device. During the time a device controls the channel, it can move data on the channel by placing an address on the address bus and data on the data bus. The capacities of different devices on the Micro Channel, in their current PS/2 implementation, are shown in Table 5.

Table 5. Data and Address Bus Widths for Micro Channel Participants

Participant	Address Bus Width	Data Bus Width
System Processor	24-bit or 32-bit	16-bit or 32-bit
DMA Controller	24-bit	16-bit
Bus Master	24-bit or 32-bit	16-bit, 32-bit or 64-bit
DMA Slave	16-bit	8-bit or 16-bit
I/O Slave	16-bit	8-bit, 16-bit or 32-bit
Memory Slave	24-bit or 32-bit	8-bit, 16-bit or 32-bit

B.3.1 Basic Data Transfer Mode

The basic transfer cycle on the Micro Channel is a minimum of 200 ns (100 ns for the address and 100 ns for the data which results in five million basic transfer cycles per second for a device running in burst mode. As shown in Figure 34 on page 134, a data transfer operation is done in two steps. First the address for the transfer (either I/O adapter or memory location) is selected, then up to four bytes of data is moved across the data buffer.

Depending of the width of the data path (8, 16, or 32 bits) the instantaneous data transfer rate on the channel would be 5, 10, or 20MB per second.

The *matched-memory* extension is a modification to the basic data transfer mode, which can improve the data transfer capabilities between the system master and channel-attached memory. When supported, it allows the basic transfer cycle of 200 ns to be reduced.

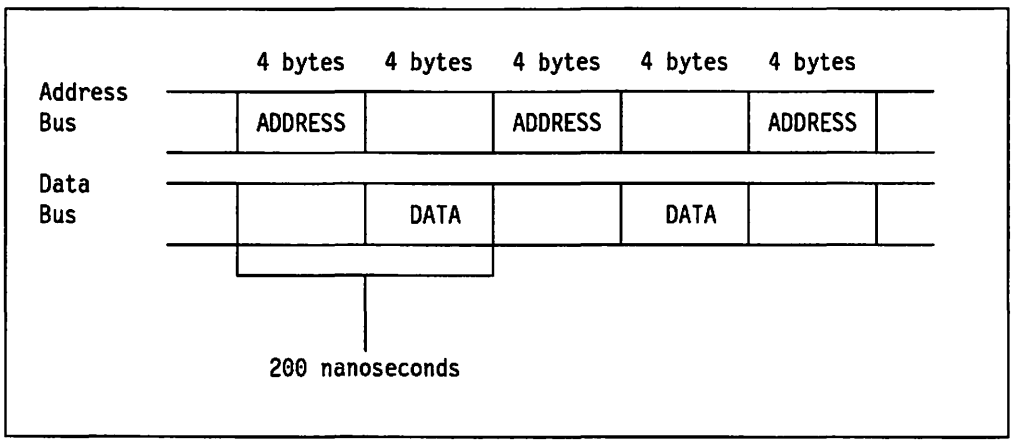


Figure 34. Basic Data Transfer Mode

The DMA controller on the system board requires two basic transfer cycles to move either 8 bits or 16 bits of data. It moves the data from the originator to a buffer in the DMA controller and then to the target device or memory location. Because two cycles are used per 8 or 16 bits of data, the data transfer rate for DMA controllers is 2.5MB or 5MB per second.

B.3.2 Streaming Data Mode

For blocks of sequential data transferred over the Micro Channel, it should not be necessary to specify the address information more than once. Both the source and destination devices should update the address for each cycle by the size of the transferred data. This technique is supported by the Micro Channel and is known as **streaming data mode** (or streaming data procedure). Using streaming data mode with 32-bit transfer, the effective transfer rate is 40MB per second. The usage of the address and data buses during a data transfer using streaming data procedure is shown in Figure 35.

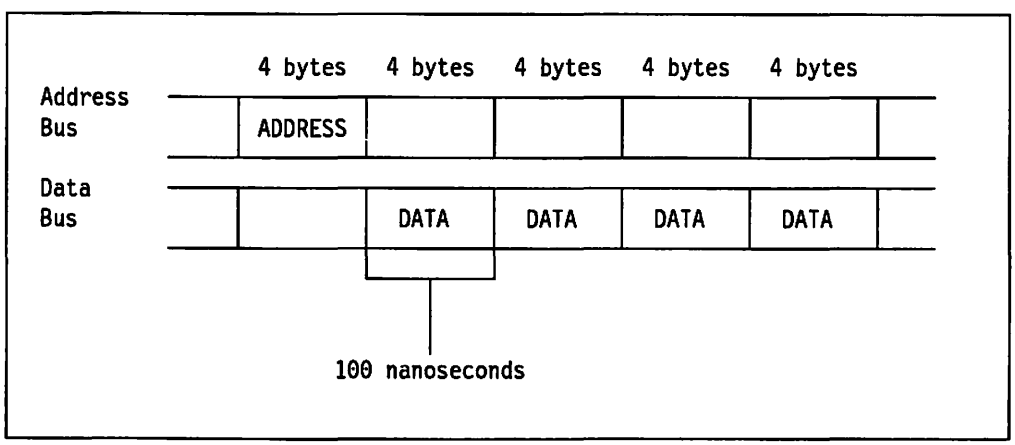


Figure 35. Streaming Data Mode

B.3.3 Multiplexed Streaming Data Mode

When the Micro Channel is running in streaming data mode, the 32 address lines are only used during the first cycle of the transfer. These address lines are therefore available for transfer of an additional four bytes during each following cycle. This mode is called **multiplexed streaming data mode** and gives an effective width of 64 bits (8 bytes) for each cycle. The resulting effective data rate is 80MB per second. This is shown in Figure 36.

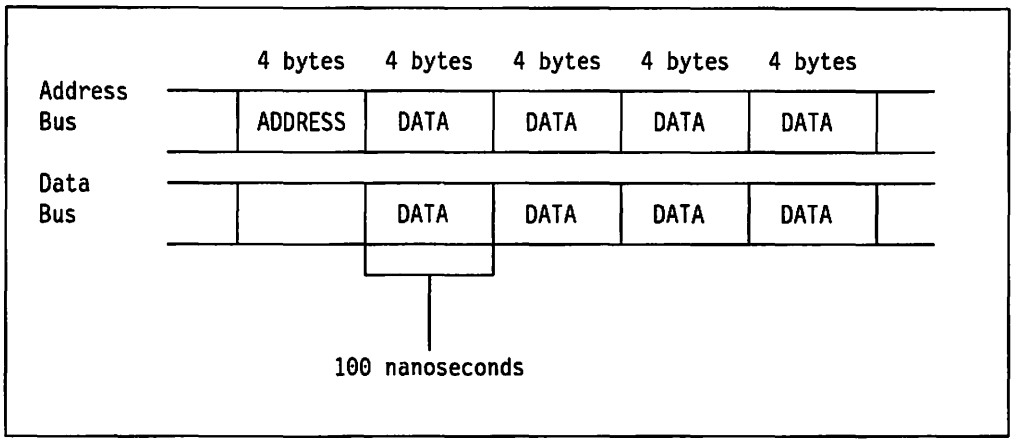


Figure 36. Multiplexed Streaming Data Mode

IBM has also disclosed that upcoming generations of Micro Channel systems may implement a faster basic transfer cycle of 100 ns rather than the current 200 ns. With the current cycle the Micro Channel is able to transfer *sequential blocks* of data with transfer rates of 20, 40, and 80MB per second. Systems implementing the faster transfer cycle would be able to reach transfer speeds of up to 160MB per second. These rates are essential for advanced function bus masters, which must move large blocks of sequential data.

B.4 Data Integrity and Exception Handling

There are two things that can go wrong while transferring data across the channel:

1. Data can be corrupted
2. Execution of an instruction can fail.

We therefore need mechanisms to report the occurrence of an error.

B.4.1 Parity Checking

When data is stored in memory or on any external device, the integrity of the data is typically ensured by having some sort of checking information stored with the data. Using this information, lost or incorrect bits may be detected and in certain cases corrected (on advanced memory systems and I/O adapters). The buses that comprise the Micro Channel are normally regarded as extremely safe conduits of information and are therefore not usually checked. While the chance of an error on the buses is very small, a missing or extra bit could still be induced by:

- Electromagnetic interference
- Power distribution disturbances (poor decoupling of logic spikes)
- Adapter cards not following timing rules properly.

To improve the integrity of the buses, the Micro Channel architecture supports address parity and data parity as an option for devices that are able to handle this function.

Odd parity is used and is implemented by using formerly reserved lines as new address and data parity lines. To indicate that parity is used/checked two new control lines are defined: *APAREN* (Address Parity Enabled) and *DPAREN* (Data Parity Enabled).

If a master addresses a slave with APAREN enabled, the slave will answer *only* if the parity is correct. If not, the bus master suspends the operation and sets an internal error flag.

During write operations from a bus master to a slave device, the slave is responsible for the parity checking if the DPAREN line is active. The slave should then indicate bad parity by activating the CHCK (Channel Check) signal. If a bus master detects a parity error during a read operation with the DPAREN line activated (by the slave), it is expected to suspend the operation and set an internal error flag.

Address parity and data parity are both optional and can be used independently of each other. Devices that use the parity checking and devices that do not are permitted to coexist in the same Micro Channel system.

B.4.2 Synchronous Exception Signaling

The Micro Channel architecture provides for exception signaling using the CHCK signal. Initial use of the CHCK signal in the PS/2 line of products was asynchronous. With the new advanced uses of the Micro Channel and bus master adapters, which are able to execute multiple I/O-operations in burst mode, synchronous exception handling is needed in order to correctly associate an exception with the command that caused it.

The synchronous exception support within the Micro Channel architecture provides for activation of the CHCK signal in the same bus cycle that caused the exception. This simplifies the exception handling and provides increased system data integrity.

B.5 IBM SCSI Implementation

In this section we give a brief description of the Small Computer Systems Interface (SCSI) and describe the currently available IBM PS/2 SCSI adapters. We also look at the support in OS/2 V2.0 for the IBM SCSI adapters.

B.5.1 What is SCSI?

SCSI is a standard interface bus, through which computers may communicate with devices such as fixed disks, CD-ROMs, printers, plotters, scanners, etc. The standard is fully described in *ANSI standard X3.131-1986*.

SCSI is a bus level interface, as compared to interfaces such as ESDI and ST-506, which are device level interfaces. With a bus level interface all the circuitry required to control the device is built into the device itself. This is the reason it is possible to attach different device types to a SCSI bus.

Up to eight SCSI devices can be attached to a single SCSI bus. The first of these devices is normally the SCSI attachment adapter, which would be installed in the PS/2 system unit and act as a bridge between the system unit bus (the Micro Channel) and the SCSI bus. A further seven SCSI devices can then be connected by means of the SCSI bus cable. Each of these physical devices can support up to eight logical devices. Each device attached to the SCSI bus has a device ID that is in the range 0 - 7. The attachment feature is usually given the ID of 7 making it the highest priority device on the bus. Figure 37 shows this diagrammatically.

SCSI supports features such as arbitration and disconnect/reconnect allowing several devices to operate concurrently and to share the bus. Data transfer across the SCSI can either be asynchronous or synchronous. Asynchronous data transfer requires that each byte sent across the bus be acknowledged before the next byte is transferred. When using synchronous mode, each byte of data must still be acknowledged but multiple bytes may be sent before any acknowledgements have been received.

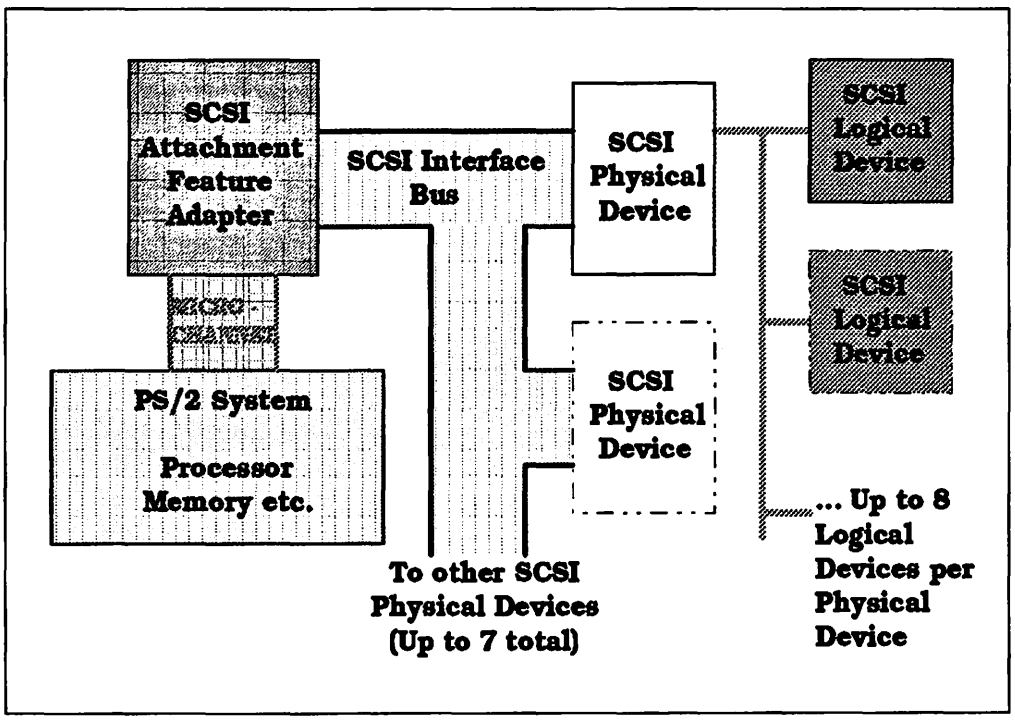


Figure 37. SCSI Subsystem Block Diagram

The cable connecting the SCSI devices may be up to 20 feet (6 meters) in length, and carries 8-bit parallel data with a transfer rate of 5MB per second.

B.5.2 IBM SCSI Adapters

IBM currently has available two SCSI adapters, which can be used in Micro Channel based PS/2s:

- **The IBM 16-bit SCSI adapter**

  This bus master adapter has a:

  - 16-bit data capability
  - 32-bit address capability.

- **The IBM 32-bit SCSI adapter with cache**

This is also a bus master adapter, which features:

- 32-bit data capability
- 32-bit address capability
- 512KB on-board cache.

Up to four adapters may be installed in a PS/2 system at the same time. The two IBM adapters support the following common features:

- The attachment of mixed SCSI device types that support the SCSI Common Command Set (CCS).
- Up to seven physical SCSI devices. Each physical device can support eight logical devices, giving a total of 56 devices per adapter (SCSI bus).
- Overlapped command processing for up to 15 devices.
- A data transfer rate of up to 5MB per second from the adapter to a device on the SCSI bus.
- A data transfer rate of up to 16.6MB per second from the adapter to the system (8.3MB per second for the 16-bit version).

SCSI devices can be attached to the IBM SCSI adapters both internally and externally. The adapters have two interface connectors. A connector on the top edge of the card allows the attachment of devices mounted internally in the system unit. There is a second connector on the end of the card, to which the external connection cable attaches.

B.5.3 Adapter Components

The different parts of the adapters are illustrated in Figure 38 on page 139, and are described below:

- **Local microprocessor, RAM and ROM**

  An Intel 80188 microprocessor controls the operation of the adapter. It translates commands received from the system into a series of operations to implement the desired results. The processor also manages the intelligent buffer, controls data transfers to and from the system, controls the SCSI bus, and handles error detection/recovery.

  The program code that controls the processor is stored in a local read-only memory (ROM) module. There is also a Random Access Memory (RAM) module available for storage of work information.

- **System interface controls**

  The system interface controls provide command and interrupt registers that allow the processor to receive commands from the system and to interrupt the system when a command is complete.

- **SCSI bus control**

  The SCSI bus control chip is an electronic circuit module used by the processor to control the SCSI bus. It is used to transmit commands, receive status information and transfer data between the adapter card and attached SCSI devices. It also provides the following functions:

  - SCSI bus arbitration
  - SCSI device selection/reselection
  - SCSI phase change detection
  - SCSI bus parity generation/checking
  - Diagnostics and self-testing capabilities.

- **Intelligent buffer**

  This buffer is a 512KB parity-checked RAM module, and is only available on the 32-bit high-performance version of the SCSI adapter. This buffer has two basic functions:

  1. It serves as a prefetch cache and keeps frequently used or prefetched SCSI device data available for immediate transfer to the system without the normal device-dependent physical access delay.
  2. It buffers the SCSI device data until it can be transferred to the system RAM via DMA. This allows the SCSI bus data transfer to be asynchronous to the Micro Channel data transfer, permitting interleaving of transfer operations, and providing enhanced performance.

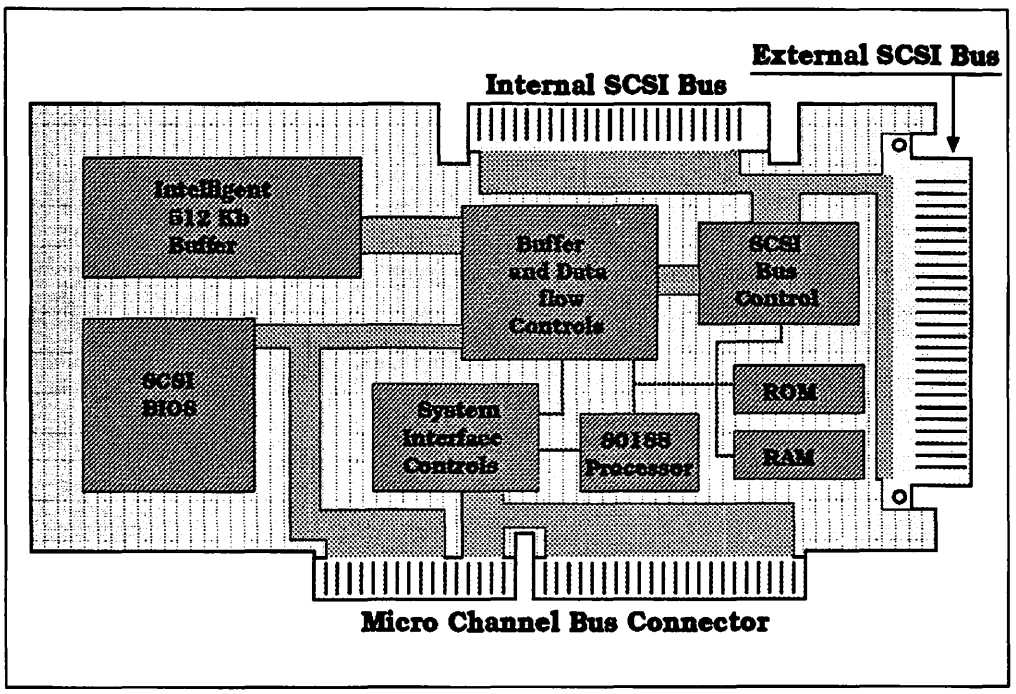


Figure 38. Adapter Component Block Diagram

- **Buffer and data flow controls**

  This circuit module manages the flow of parallel data between the SCSI bus, the intelligent buffer, and the Micro Channel. The circuit is controlled by the processor and can perform all transfers simultaneously. It supports 32-bit to 16-bit conversion, burst mode DMA, odd byte/word transfers and parity generation/checking of the buffer data.

- **SCSI BIOS**

  The SCSI BIOS consists of a set of ROM modules, which contain the basic input/output support code for the adapter. It provides both the compatibility (CBIOS) and advanced (ABIOS) support for any attached SCSI fixed disk. The BIOS ROM is completely independent from the processor's ROM on the adapter. BIOS is discussed further in B.5.4, "SCSI BIOS" on page 140.

B.5.4 SCSI BIOS

As mentioned above, the SCSI adapters contain ROM modules with the compatibility BIOS (CBIOS), the advanced BIOS (ABIOS), and the power-on self test (POST) routines for the SCSI subsystem. Each of these BIOS functions is explained in the following sections.

B.5.4.1 Compatibility BIOS (CBIOS)

The CBIOS interface supports SCSI device and adapter functions for the single tasking environment, as implemented by PC DOS. The SCSI CBIOS intercepts and replaces parts of the **Interrupt 13h (fixed disk)** and **Interrupt 15h (system services)** functions. See Figure 39 for a functional overview of the BIOS interface. It also adds the new **Interrupt 4Bh (advanced services)** support to provide a generic interface to SCSI devices other than disk drives. This new interface uses the carry flag and the AH register to report status information.

The CBIOS interface makes it possible for programs in a DOS environment to use both SCSI and other types of disks via INT 13h, with the help of new device drivers and INT 4Bh services. The CBIOS interface also supports other types of SCSI devices.

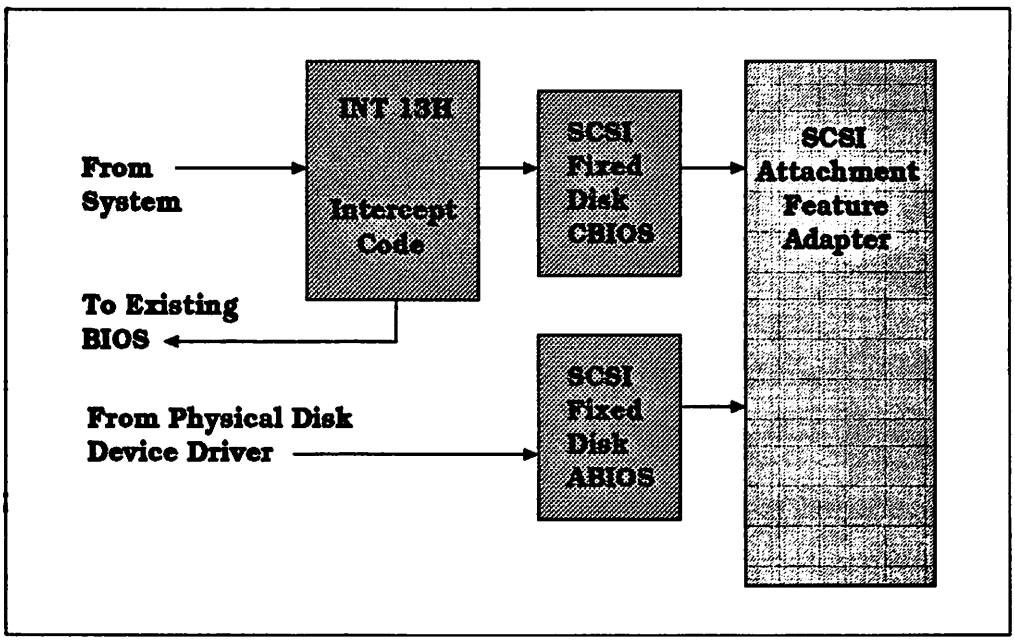


Figure 39. SCSI BIOS Interface Block Diagram

B.5.4.2 Advanced BIOS (ABIOS)

The ABIOS interface shown in Figure 39 provides new and enhanced **device ID 02h (fixed disk)** support. ABIOS used by operating systems such as OS/2 V2.0, which use the protected mode of the Intel microprocessors. The enhanced functions enable support of both SCSI and other types of disk without an upgrade of operating system software. The new functions enable a more effective use of SCSI disk devices by operating systems through new and enhanced device drivers. The new ABIOS functions are:

- **0Dh - Enable Intelligent Buffer**
- **0Eh - Disable Intelligent Buffer**

These two functions enable or disable the intelligent buffer capability for all subsequent commands to this device. If the function is not supported, an error code is returned.

- **0Fh - Return Intelligent Buffer Status**

  Returns the status of the intelligent buffer. Status may be *enabled*, *disabled* or *not supported*.

- **10h - Set DMA Pacing Factor**

  This function programs the adapter with the supplied pacing value. The value specifies what percentage (25% to 100%) of the DMA transfer bandwidth the adapter is allowed to use. All devices on the adapter are affected.

- **11h - Return DMA Pacing Factor**

  This function returns the current pacing factor for the adapter.

- **12h - Transfer SCB.**

  This function programs the adapter to process a Subsystem Control Block (SCB). SCBs are described further in B.6, "Subsystem Control Block Architecture" on page 142.

B.5.4.3 Power-On Self Test (POST)

The POST routines will automatically configure the SCSI subsystem at power-on and system reset. The SCSI subsystem supports the coexistence of other types of fixed-disk adapters, and if more than one SCSI adapter is installed in a system, the BIOS ROM modules on the extra adapters will be disabled by the POST routines.

POST will also issue the *Inquiry command* to all 56 possible combinations of physical unit number/logical unit number (PUN/LUN), until 15 logical devices (logical device IDs 0 to 14) are assigned. The SCSI adapter is logical device 15.

B.5.5 Support for Generic SCSI Functions

The SCSI ABIOS provides support for two new device types:

- Device ID 17h - SCSI Adapter support

  Functions for this device ID control the SCSI adapter, and some of them also affect all devices attached to the adapter.

- Device ID 18h - SCSI Peripheral Type support.

  Functions for this device ID control the devices attached to the SCSI adapter. Some functions affect all devices attached to the adapter, and others are used to control a specific device via its logical ID.

These device types are intended for access to SCSI devices other than disk drives. Examples of these generic types are CD-ROMs, tape devices and communication devices, all of which may theoretically use the SCSI bus, provided the device is designed to do so.

The logical IDs for a particular peripheral type are allocated/deallocated to specific SCSI devices on demand via the allocate and deallocate function. Through the allocate function the device driver specifies a SCSI peripheral type, a removable media indicator and a relative unit number. If the device exists and is unallocated, ABIOS assigns that device to the logical ID. The controlling program can then use the logical ID to make requests through the SCSI adapter.

B.6 Subsystem Control Block Architecture

The bus master in Micro Channel-based systems is capable of transferring data on the system channel without intervention from the system controller. Typically the system processor issues a request to the bus master. The bus master then takes charge and controls all further operations required to complete the request. The operations might be:

- Issuing control commands to physical devices attached to the bus master
- Data transfers to or from the physical device
- Data transfer across the Micro Channel to or from system memory
- Handling exception conditions
- Notifying the system processor of completion of the operation.

To utilize the power of the bus master we need an effective method of communicating control instructions to it.

The **Subsystem Control Block (SCB)** architecture defines logical protocols and services needed to transfer commands, data, and status information between the system processor and a bus master adapter and between two bus master adapters. The term *subsystem* is used since the advanced function adapters are much more than simple interfaces to peripheral devices. These adapters typically have some *intelligence* built into them in the form of local processing power and can be regarded as "systems within a system".

The SCB architecture provides a powerful means of communication between device drivers running in the system processor and I/O processors that are capable of operation independently from the system processor. The architecture defines a control block structure for use between functional entities in a base system processor and functional entities in advanced function adapters. Entities are defined as either *client* or *server*. *Client* entities make requests of *server* entities. The architecture also defines the way that control blocks are passed between entities in the system processor and entities in the feature adapters, or between entities in two adapters (peer-to-peer communication).

The SCB architecture sees the system unit and each adapter logically structured into three levels:

- The *physical level* handles the transfer of the control information and data across the Micro Channel. It accesses either the MCA-defined I/O address space or memory address space.
- The *delivery level* is an interface between the physical level and the processing level. It uses the services provided by the physical level to transfer requests and replies between processing level entities.
- At the *processing level* we have the client and server entities. The client builds and sends requests to the server and the client receives replies in response.

We will now discuss the SCB architecture and the manner, in which it is implemented in the IBM SCSI adapters.

B.6.1 I/O Port Definitions

To control an advanced function bus master adapter, the SCB architecture defines a set of I/O ports. These ports are assigned as a sequence of register addresses in the I/O address space. Since multiple adapters of the same or different types may be used in the system, the base address for the I/O space of each adapter must be defined during system setup. The IBM SCSI adapters support eight address ranges. However, ABIOS supports only four. The ports used by the IBM SCSI adapters are shown indexed from the base address in Figure 40. The IBM SCSI adapter implements a subset of the ports defined in the SCB architecture.

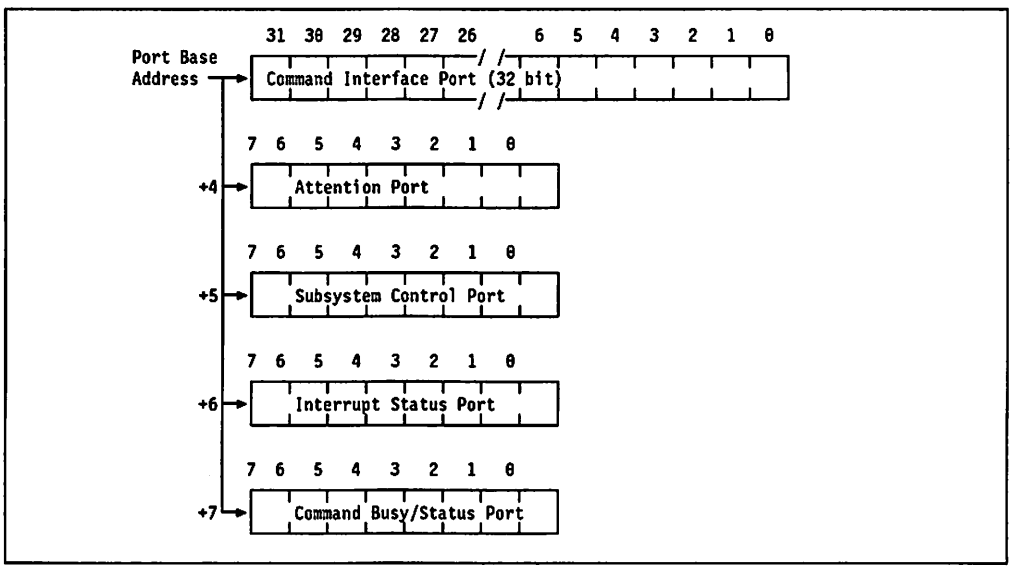


Figure 40. IBM SCSI Adapter I/O Ports

The following are general descriptions of the I/O space control ports:

- **Request Ports**

  There are four types of request ports associated with sending requests to a device. The first of these is the **Command Interface Port (CIP)**, which consists of four 8-bit read/write registers used to transfer either a 32-bit immediate command or a Subsystem Control Block address from the system to the adapter. The immediate commands are typically device-directed and control-oriented.

  The second is the **Attention Port (AP)**. This is an 8-bit read/write register used by the system to request an adapter operation. The register contains two pieces of information. The high-order four bits define the operation request code and the low-order four bits define the logical device that should be selected for the operation.

  The third, the **Interrupt Status Port (ISP)**, is an 8-bit read-only register used by the subsystem adapter to return command completion information, if allowed by the interrupt-enable bit in the Subsystem Control Port. The adapter stores the logical device ID in the four low-order bits and an interrupt ID in the high-order bits. A hardware interrupt is then generated.

  The fourth port is the **Command Busy/Status Port (CB/SP)**. It is an 8-bit read-only register that is used by the adapter to serialize access to the shared logic of the control block delivery service. It contains the following indicators:

- **Busy** - indicates that the adapter is busy (using the shared logic). Commands submitted while the busy indicator is on are ignored by the feature adapter.
- **Interrupt Valid** - indicates that the content of the ISP (Interrupt Status Port) is valid, and that the feature adapter has requested an interrupt on behalf of one of its entities. This indicator will be set even if interrupts are not allowed by the system and will be reset by the EOI (End Of Interrupt) request. This allows an adapter to operate without presenting hardware interrupts to the system.
- **Reject** - indicates that the feature adapter has rejected a request. A *Reset Reject* request is needed to clear the reject signal and allow the adapter to resume accepting requests.
- **Status** - three bit indicator giving the reason for rejection.

- **Subsystem Control Port**

  The Subsystem Control Port (SCP) is an 8-bit read/write register used for direct hardware control of the subsystem adapter. This type of control cannot typically be handled by requests to the adapter management via immediate commands through the Command Interface Port (CIP). The SCP contains the following control indicators:

  - **Enable interrupt** - indicates that interrupts to the system unit should be enabled or disabled for all devices attached to the adapter.
  - **Enable DMA** - indicates that DMA operations should be enabled or disabled for the subsystem.
  - **Reset reject** - indicates that a reset of the reject state for the subsystem should be performed (see *Reject* under the description of Command Busy/Status Port above).
  - **Hardware reset** - indicates that a controlled reset of the adapter and attached devices should be performed.

B.6.2 Delivery Service Structure

The SCB architecture defines a structure that supports the physical delivery of control information between multiple entities (requesting or serving functions) executing in a base system processor and one or more peer entities executing in an advanced function adapters. Some entities may act as clients (requesters) while others act as servers. The entities build the control block structure, and the *control block delivery service* is used by one entity in either a system unit or feature adapter to communicate control block information to another entity.

Based upon the information within a control block, data may also be communicated between the two entities. The communication of data is referred to as *data delivery*. The delivery of data is separate from the delivery of command and control information and the individual entities are responsible for requesting the transfer of the data.

In the simplest case, the control block delivery service may be viewed as supporting communication between client entities located in the system unit and server entities located in an adapter.

B.6.3 Delivery Service Facilities

The SCB architecture defines two forms of control block delivery, known as **locate mode** and **move mode**. As the IBM SCSI adapters use locate mode delivery only this mode will be described. The following section provide an overview of the services, functions and protocols defined for locate mode and a brief description of their underlying control structures.

B.6.3.1 Locate Mode

Locate mode is generally used for traditional I/O protocols where there is a single system unit that requests a feature adapter to perform work on its behalf. The format of the control structure in locate mode is relatively fixed. The structure allows various command, status, and indirect list control blocks, connected with pointers, to represent a request from a client to a server. Figure 41 shows an overview of the control block and data delivery support.

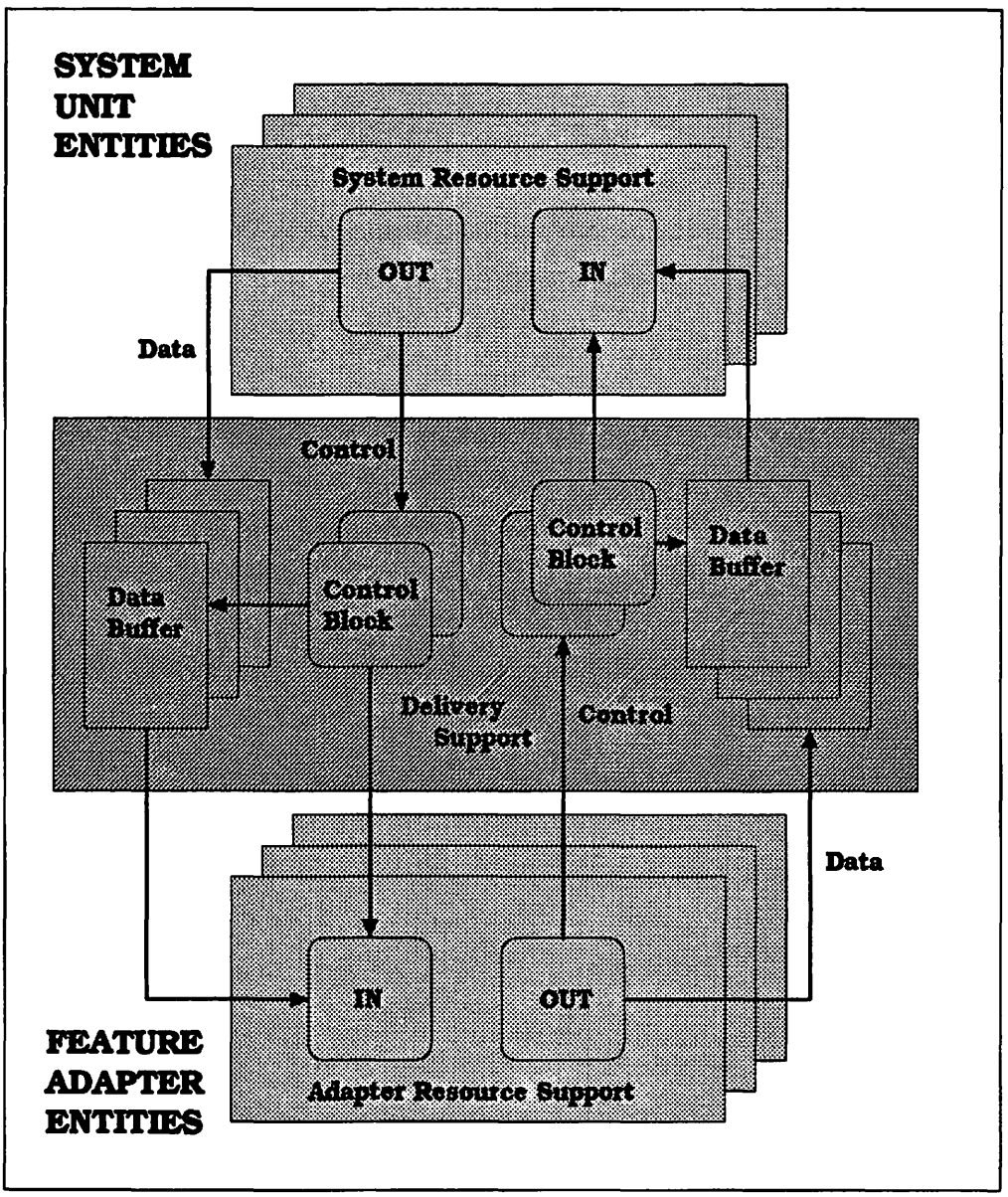


Figure 41. Overview of Delivery Support

The delivery support delivers one request at a time. A control block request is initiated by the client, which passes the physical address of the control block to the server through a command register (port) in I/O. The I/O space ports are described in B.6.1, "I/O Port Definitions" on page 143.

Locate mode provides:

- **Requests to Devices**

  To use a device, a client sends requests in the form of control blocks to the device ID that represents the device and receives replies from it for each such request.

- **Support for Multiple Devices per Adapter**

  Adapters may provide support for multiple devices. For example, multiple I/O devices may be connected to a single SCSI adapter. The delivery support enables the delivery of requests to specific devices through the use of unique device identification numbers.

- **Adapter Management**

  To be able to manage the adapter and deliver device/resource control information to it, the adapter's unit manager is assigned device ID 0. It receives all adapter unit management information.

- **Command and Data Chaining and Detailed Status**

  The control structure defined for locate mode provides for:

  - Immediate command type requests
  - Requests that contain one command control and status block
  - Requests made up of multiple chained command control blocks.

  Command blocks may point to data directly or via an indirect address list. Figure 42 on page 147 shows a sample request control structure consisting of two command control blocks (command chaining). The first control block uses an indirect list control block to reference multiple buffers (data chaining). The second control block has a direct pointer to a single buffer. The pointer to the first control block is communicated via the shared I/O address space.

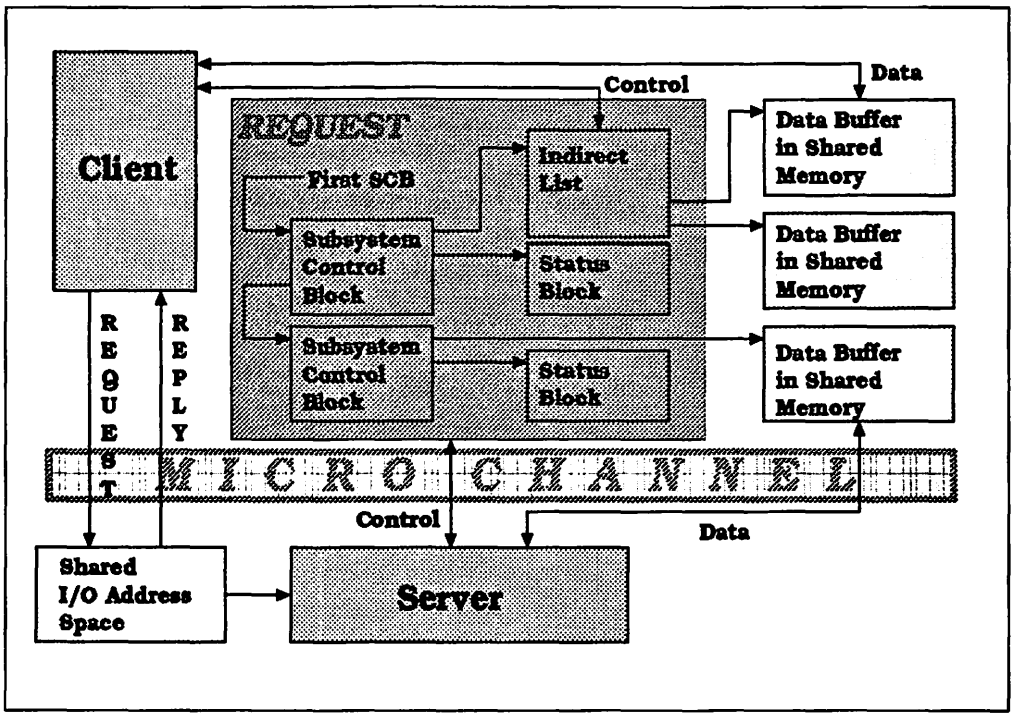


Figure 42. Locate Mode Control Block Delivery Structure

- **Use of DMA**

  In locate mode, both the control block structure for a request and the data associated with the request are transferred between the system unit and the adapter using DMA. This DMA operation is managed by the adapter's delivery support. The IBM SCSI adapters have a DMA with a 32-bit addressing capability, hence they are not limited by the fact that the DMA in the IBM PS/2 system units currently only supports 24-bit addressing.

- **Interrupts**

  The delivery service allows the client to define when and under what conditions interrupts should be generated. Generally, a single interrupt is generated for each completed request. When a request contains a chain of control blocks, additional interrupts may be requested for synchronization at intermediate points within the request.

There are three different control block structures used by locate mode requests:

- **Command Control Blocks**

  The base command control block is a variable length structure created and maintained in shared memory by the client in the system unit. It is used to convey requests to a subsystem or device in an adapter. These requests are used to direct the operation of the subsystem or device. In Figure 43 on page 148 we show the structure of the command control block as used by the IBM SCSI adapters.

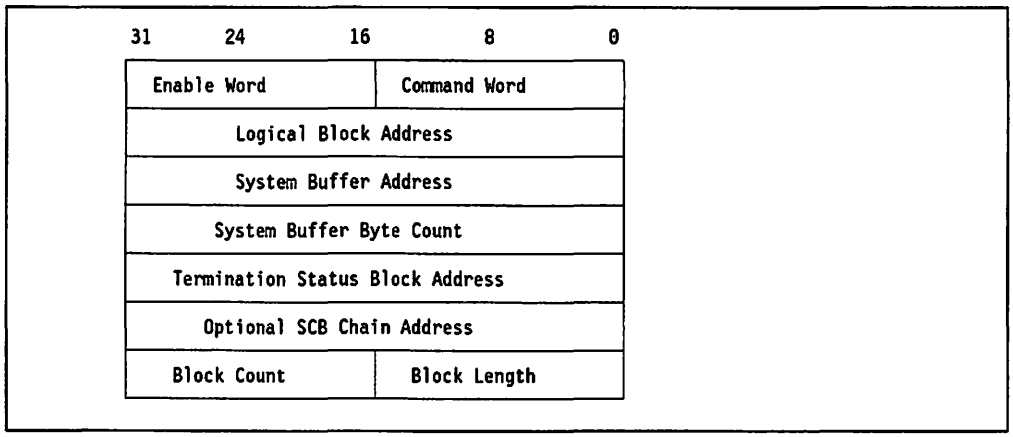


Figure 43. SCB Structure Used by the IBM SCSI Adapter

The **command word** specifies the operation to be executed, for example read or write. The **enable word** controls optional features to be used during the operation, for example whether the system buffer address points to actual data or is an indirect reference.

The OS/2 V2.0 disk device driver builds the SCB and other control blocks required. It then calls the ABIOS transfer SCB function to initiate the operation. ABIOS uses the adapter's I/O ports to pass the SCB address. The adapter then uses its DMA controller to read the SCB into private memory on the adapter where it decodes it and initiates the required operation. Data is read from or written to system memory using the adapters DMA controller.

The IBM SCSI adapters support chaining of SCBs. The adapter then uses the chain address to find the next SCB in a chain. All SCBs in a chain must be for the same device. SCBs are executed in the order that they are chained in. The device driver will try to sort the SCBs into an order that will minimize head movement unless requested not to. Where the order of execution is critical (for example, paging might require that certain writes be done before reads) this optimization will be disabled.

A single IBM SCSI adapter can overlap processing of one command for up to 15 attached devices.

- **Indirect Address Lists**

  An indirect address list is a variable length list containing address-count pairs, and is used to support data chaining. The location of the indirect list control block and its length are specified by the System Buffer Address and System Buffer Byte Count fields in the command control block.

  Each entry in the list is 8 bytes (4 byte address and 4 byte length).

  The IBM SCSI adapters use the indirect address list structure for specifying scatter/gather lists. The layout of a scatter/gather list is shown in Figure 44 on page 149. When passing a memory address to an adapter, the actual physical address at which the data is located must be used. Because of the mapping of virtual memory to pages in the physical address space, a data buffer could be split across multiple discontiguous pages. This means that a single logical disk I/O operation could require several physical operations. The scatter/gather list allows one I/O operations to access data in more than one physical memory location. Each entry in the list specifies a physical memory address and a data length.

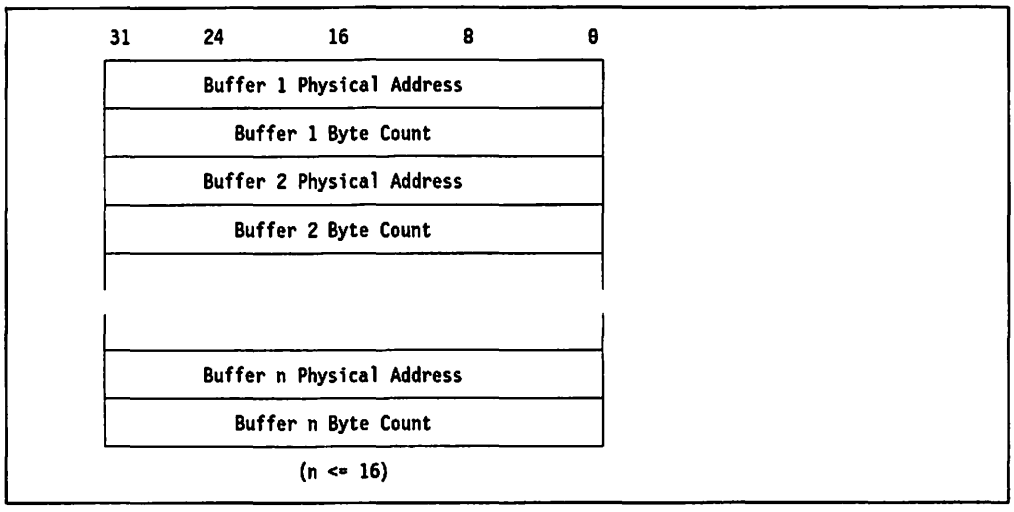


Figure 44. IBM SCSI Adapter Scatter/Gather List

During a write to disk, data is then gathered from different memory locations and written to a contiguous block on disk. During a read from disk, data can be scattered to different locations in system memory. The IBM SCSI adapters support scatter/gather lists with up to 16 entries.

When coupled with the ability to chain SCBs together, scatter/gather lists form an efficient mechanism for communicating paging I/O operations. By being able to chain SCBs together, both reads and writes can be specified in a single call from the paging subsystem to the file system. Scatter/gather lists allow multiple page reads or writes for a single SCB.

- **Termination Status Control Blocks**

  In addition to the status indication in the ISP (Interrupt Status Port), the SCB architecture allows status information to be reported for each command. This information can be reported in a termination status block, which is connected to each command control block. The adapter writes request completion status or termination status into this control block. The layout of the termination status block used by the IBM SCSI adapters is shown in Figure 45.

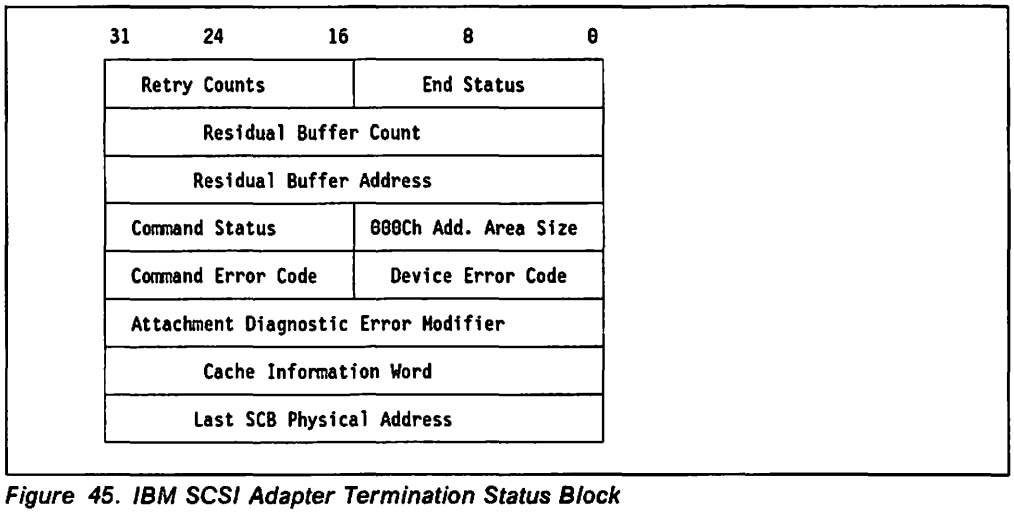


Figure 45. IBM SCSI Adapter Termination Status Block

B.6.4 Additional Information

More detailed information about the SCSI adapters can be found in the *Personal System/2 Micro Channel SCSI Adapter Technical Reference* (S68X-2397-00) or the *Personal System/2 Micro Channel SCSI Adapter with Cache Technical Reference* (S68X-2365-00).

The SCSI BIOS information can be found in the *Supplement for the Personal System/2 and Personal Computer BIOS Interface Technical Reference, December 1989* (S15F-2161-00).

Appendix C. Lab Session - 32-Bit Memory Model

This lab session provides practical experience with the 32-bit flat memory model implemented under OS/2 Version 2.0, and demonstrates some of the new memory management facilities that were introduced in OS/2 Version 2.0. The exercise shows the way in which OS/2 Version 2.0 memory management differs from the memory management in previous releases. A look is also taken at enhancements made in the area of thread processing.

C.1 Objectives

Subjects which will be covered in this lab session are:

1. Memory allocation using the DosAllocMem() function
2. Page granularity in memory allocation
3. The general protection exception error message
4. The guard page fault exception error message
5. The new SWAPPER.DAT functionality
6. Starting multiple DOS sessions
7. Starting multiple threads.

Software required to do these exercises:

- OS/2 Version 2.0
- C Set/2
- IBM Developer's Toolkit for OS/2 2.0

C.2 Exercise 1 - Memory Allocation

This exercise focuses on the DosAllocMem() function and its usage. The sample program used in the exercise allocates and uses memory in order to illustrate the new memory allocation mechanism, particularly with regard to paged memory.

In this exercise, the student is required to run the program:

```
MEMLAB1.EXE
```

This program does the following:

1. Asks for an amount of memory to be allocated, in integer-sized (4 byte) units.
2. Allocates and commits the requested amount of memory.
3. Asks for the amount of memory to be used for read/write operations. The amount of memory to be used is specified as the number of integers to be written then read.
4. Performs writes to fill the requested amount of memory.
5. Checks whether the written values are correct.
6. Frees the allocated memory.
7. At initiation the program registers two termination routines:

a. An abnormal termination routine, to which general protection exceptions will be routed.

b. A termination routine, which is called when the program exits normally.

These routines are used to print out the program status at termination and to pinpoint where a general protection exception occurred. Note that C Set/2 routines are used to register the exception handler. **DosSetExceptionHandler()** could also have been used.

The program will be executed three times during the course of the exercise. The program listing is shown in C.3.4, "Source Code MEMLAB1.C" on page 153.

C.2.1 Step 1 - Normal Memory Allocation

Execute the program MEMLAB1.EXE, and instruct the program to perform the following:

- Allocate one integer.
- Specify 200 integers for read/write.

The program should execute without error. Given that more memory has been accessed during the write operation than was originally allocated, explain how this is possible.

C.2.2 Step 2 - Memory Protection Violation

Execute the program MEMLAB1.EXE, and instruct the program to perform the following:

- Allocate one integer.
- Specify 1200 integers for read/write.

In this step, the program should terminate with an error. Note the system error message, and the memory location where the error occurred. Given that each integer is a double word (4 bytes), explain why the error occurred.

C.2.3 Step 3 - Large Memory Allocation

Execute the program MEMLAB1.EXE, and instruct the program to perform the following:

- Allocate 1048576 integers (4MB).
- Use 67584 integers (254KB) for read/write.

Note: This will take some time to complete. Please be patient.

Check the program code to see how the memory allocation is done. Explain why this method works for all sizes of memory allocation. Compare this to the OS/2 V 1.x memory allocation scheme.

If you are able to do the exercise

Please do so now. If not, refer to the following explanation.

If the program does not work, check for the free disk size on the logical drive where the SWAPPER.DAT file is located. Erase excess files and rerun the program. Contact your instructor for further assistance.

C.3 Expected Results from Exercise 1

After successfully completing the exercise, you will have tried a number of memory allocation options available using the DosAllocMem() function. The results from each step are explained below.

C.3.1 Step 1

You will notice that the operating system always allocates at least 4KB (1 page) of memory, even although only 1 byte may be specified in the allocation request. This is because all memory management in OS/2 Version 2.0 is handled on a per-page basis, using the flat memory model. The page is the lowest level of granularity in OS/2 Version 2.0; this differs from previous versions of OS/2, which had a byte-level granularity.

This principle applies to *all* memory protection and memory access types (read, write, etc.). This allows a programmer to use more memory for read/write than is actually requested in an allocation request as shown in step 1, provided that memory is within the physical page boundary.

C.3.2 Step 2

In step 2, a general protection exception occurs when attempting to access 1200 integers for read/write. Since each addressable memory location is a double word (4 bytes), there are a maximum of 1024 addressable locations in a single page. Attempting to access 1200 integers resulted in a general protection exception when the 1025th location was accessed.

Should the program have done a second **DosAllocMem**, the page following the page containing the memory object used in the exercise would have also been valid in the process address space. The protection exception would then not have been generated. 32-bit programs see the process address as a single segment.

C.3.3 Step 3

The flat memory model allows a programmer to address any location within a memory object on a contiguous basis. The DosAllocMem() function returns a 32-bit memory address, and *not* a segment selector as in previous versions of OS/2; hence, the programmer does not have to consider the 64KB segment boundary limitation. This allows the programmer to allocate a memory object of an arbitrary size up to 512MB (the process space address limit).

C.3.4 Source Code MEMLAB1.C

```
/*************************************************************/
/*************************************************************/
/***                                                       ***/
/***  Program name: MEMLAB1.EXE                            ***/
/***                                                       ***/
/***  Created     : 7. May 1998                            ***/
/***                                                       ***/
/***  Author      : Bo Falkenberg                          ***/
/***                                                       ***/
/***  Revised     : February, 1992 by Darryl Frost         ***/
/***                                                       ***/
/***  Purpose     : To demonstrate the use of the new      ***/
/***                DosAllocMem API, and the handling      ***/
/***                of General Protection Exceptions.      ***/
/***                                                       ***/
/***  Compile     : icc /W2 memlab1.c;                     ***/
/***                                                       ***/
/***  Execute     : memlab1 (No command line parameters)***/
/***                                                       ***/
/***  Input param : 1. Memory to allocate                  ***/
/***                2. Memory to use for read/write        ***/
/***                                                       ***/
/*************************************************************/
/*************************************************************/

/*************************************************************/
/***  DEFINES                                              ***/
/*************************************************************/
#define INCL_DOSMEMMGR

/*************************************************************/
/***  INCLUDE                                              ***/
/*************************************************************/
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/***     GLOBAL VARIABLES                                  ***/
ULONG    ulLoop;     /* loop variable                       */
```

```
/***      FUNCTION PROTOTYPES                              ***/
void main(int argc, char *argv[], char *envp[]);
void traphandler(int sig);
void normalexit(void);

/***  MAIN PROGRAM                                         ***/
void main(int argc, char *argv[], char *envp[])
{                          /*****************************************/
   PULONG    pulBlock;     /* pointer to the starting memory location */
   ULONG     ulErr;        /* error variable                          */
   ULONG     ulAmount;     /* amount of memory to be allocated        */
                           /*           (long integers)               */
   ULONG     ulBytes;      /* amount of memory to be allocated (bytes)*/
   ULONG     ulUse;        /* amount of memory to be used             */
                           /*           (long integers)               */
   BOOL      OK = TRUE;    /* memory check indicator                  */
                           /*****************************************/
   setbuf(stdout, NULL);
/* Register an exception handler for memory exception     */
   if (signal(SIGSEGV, traphandler) != SIG_ERR)
      printf("\nSignal Handler registered for memory exceptions\n");
/* Register an exit routine for normal exits */
   if (atexit(normalexit) == 0)
      printf("\nExit handler for normal termination registered\n");
/* Read parameters: 1. Memory to allocate    */
/*                  2. Memory to use         */
/* Both parameters as number of long integers*/
   printf("\nFor how many long integers should memory be allocated : ");
   scanf("%u", &ulAmount);
/* Determine number of bytes to allocate     */
   ulBytes = ulAmount * sizeof(ULONG);
   printf("\nHow long integers should be written into this memory : ");
   scanf("%u", &ulUse);
/* Allocate the memory                       */
   ulErr = DosAllocMem ( (PPVOID)&pulBlock, ulBytes,
                         PAG_COMMIT | PAG_READ | PAG_WRITE);
   if (!ulErr)
   {
/* Insert values into ulUse memory           */
      printf("\nInserting integers into memory\n");
      for (ulLoop = 0; ulLoop < ulUse; ulLoop++)
      {
         *(pulBlock + ulLoop) = '\xAB';
      } /* endfor */
/* Read the memory to check that it is OK    */
      for (ulLoop = 0; ulLoop < ulUse; ulLoop++)
      {
         if (*(pulBlock + ulLoop) != '\xAB')
         {
            printf("\nError in byte %u\n", ulLoop);
            OK = FALSE;
         } /* endif */
      } /* endfor */

      if (OK)
      {
         printf("\nAll memory checked out OK\n");
      } /* endif */

/* Free the memory                           */
      ulErr = DosFreeMem (pulBlock);
      if (ulErr != 0)
      {
        printf("\nError in freeing : code %u\n", ulErr);
      } /* endif */
   } else
   {
     printf("\nError in allocation : code %u\n", ulErr);
   } /* endif */

}
/* ABNORMAL TERMINATION HANDLER              */
void traphandler (int sig)
{
     printf("\nA General Protection Exception was detecting writing to"
            " position %u\n", ulLoop+1);
}
/* NORMAL TERMINATION ROUTINE                */
void normalexit(void)
{
     printf("\n%u integers successfully inserted into memory\n", ulLoop);
}
```

C.4 Exercise 2 - Memory Protection

This exercise demonstrates the different memory attributes used for different types of memory access, and the resulting impact on the allowed usage of memory (read or write). The access types used in this lab exercise are READ, WRITE, and EXECUTE. The effect of not committing pages and accessing a guarded page is also shown.

In this exercise, the student is required to run the program:

MEMLAB2.EXE

This program does the following:

1. Asks for the amount of memory to be allocated
2. Asks for the type of allocation:
   - PAG_COMMIT and PAG_READ
   - PAG_COMMIT and PAG_WRITE
   - PAG_COMMIT and PAG_EXECUTE
   - PAG_COMMIT and PAG_GUARD and PAG_WRITE
   - PAG_WRITE
3. Asks whether to READ or WRITE to this memory
4. Allocates the memory with the requested attributes and either reads from or writes to this memory.

The program listing is shown in C.5.1, "Source Code MEMLAB2.C" on page 155.

This program should be run a number of times during the course of the exercise, trying the different attributes and reading from/writing to the memory allocated.

If you are able to do the exercise

Please do so now. If not, refer to the following explanation.

C.5 Expected Results from Exercise 2

After successfully completing the exercise, you will see the following results:

1. Allocation type PAG_COMMIT and PAG_READ:
   - Read request is OK.
   - Write request fails with a general protection exception.
2. Allocation type PAG_COMMIT and PAG_WRITE:
   - Read request is OK.
   - Write request is OK.
3. Allocation type PAG_COMMIT and PAG_EXECUTE:
   - Read request is OK.
   - Write request fails with a general protection exception.
4. Allocation type PAG_COMMIT and PAG_GUARD and PAG_WRITE:
   - Both read and write requests fail with a guard page exception.
5. Allocation type PAG_WRITE:
   - Both read and write requests fail with a general protection exception.

The behavior seen in cases 1, 2 and 3 occurs because the 80386 processor does not distinguish between READ and EXECUTE access. Furthermore, READ or EXECUTE access is allowed even for memory objects specified with WRITE access. This is because WRITE access implies both READ and EXECUTE access. However, WRITE access *must* be specified in order for the application to write into a memory object.

In case 4, because the guard page bit is set, a guard page fault will always occur when accessing the page. You also should have seen a second message, "*Exception occurred handling a prior exception*". This occurs when the default guard page exception handler is invoked and tries to treat the area as a stack.

In case 5, because the memory allocated was not committed, no physical space was reserved. Whenever accessing any of this memory a general protection exception is always raised.

C.5.1 Source Code MEMLAB2.C

```
/*************************************************************/
/*************************************************************/
/***                                                       ***/
/***  Program name: MEMLAB2.EXE                            ***/
/***                                                       ***/
/***  Created     : 7. May 1990                            ***/
/***                                                       ***/
/***  Author      : Bo Falkenberg                          ***/
/***                                                       ***/
/***  Revised     : February, 1992 by Darryl Frost         ***/
/***                                                       ***/
/***  Purpose     : To demonstrate the different types     ***/
/***                of memory allocation.                  ***/
/***                                                       ***/
/***  Compile     : icc /W2 memlab2.c                      ***/
/***                                                       ***/
/***  Execute     : memlab2 (no command line parameters)   ***/
/***                                                       ***/
/***  Input param : 1. Amount of memory in KB              ***/
/***                2. Type of memory allocation.          ***/
/***                3. Type of memory usage                ***/
/***                                                       ***/
/*************************************************************/
/*************************************************************/

/*************************************************************/
/***  DEFINES                                              ***/
```

```
/*********************************************************/
#define INCL_DOSMEMMGR

/*********************************************************/
/***  INCLUDE                                          ***/
/*********************************************************/
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>

/*********************************************************/
/***  FUNCTION PROTOTYPES                              ***/
/*********************************************************/
void main(int argc, char *argv[], char *envp[]);

/*********************************************************/
/***  MAIN PROGRAM                                     ***/
/*********************************************************/
void main(int argc, char *argv[], char *envp[])
{                       /*****************************************/
   PULONG pulBlock;     /* pointer to the starting memory location */
   ULONG ulErr;         /* error variable                          */
   ULONG ulLoop;        /* loop variable                           */
   ULONG ulAmount;      /* amount of memory to be allocated        */
   ULONG ulSelection;   /* input selection                         */
   char cLetter;        /* input char                              */
   BOOL OK = TRUE;      /* memory check indicator                  */
                        /*****************************************/

   setbuf(stdout, NULL);

/* Read the parameters                                  */
/*     1. Amount of memory in KBytes to be allocated    */
/*     2. Combination of allocation flags               */
/*     3. Whether data should read from or written      */
/*        the memory.                                   */
   printf("How much memory (in KB) do you want to allocate : ");
   scanf("%u", &ulAmount);
   ulAmount = ulAmount * 1024;

   printf("\nWhat type of memory allocation do you want: \n");
   printf("  1. PAG_COMMIT and PAG_READ\n");
   printf("  2. PAG_COMMIT and PAG_WRITE\n");
   printf("  3. PAG_COMMIT and PAG_EXECUTE\n");
   printf("  4. PAG_COMMIT and PAG_GUARD and PAG_WRITE\n");
   printf("  5. PAG_WRITE\n");
   printf("Enter your selection (1, 2, 3, 4 or 5 ) : ");
   scanf("%u", &ulSelection);

   printf("\nDo you want to Read or Write in the memory (R/W) : ");
   fflush(stdin);
   scanf("%c", &cLetter);
/* Allocate the memory                                  */
   switch (ulSelection)
   {
      case 1:
         ulErr = DosAllocMem ((PPVOID)&pulBlock, ulAmount,
                              PAG_COMMIT | PAG_READ);
         break;
      case 2:
         ulErr = DosAllocMem ((PPVOID)&pulBlock, ulAmount,
                              PAG_COMMIT | PAG_WRITE);
         break;
      case 3:
         ulErr = DosAllocMem ((PPVOID)&pulBlock, ulAmount,
                              PAG_COMMIT | PAG_EXECUTE);
         break;
      case 4:
         ulErr = DosAllocMem ((PPVOID)&pulBlock, ulAmount,
                              PAG_COMMIT | PAG_GUARD | PAG_WRITE);
         break;
      case 5:
         ulErr = DosAllocMem ((PPVOID)&pulBlock, ulAmount,
                              PAG_WRITE);
         break;
      default:
         printf("\nYou made a WRONG selection !!!\n");
         exit (0);
   } /* endswitch */

   if (ulErr != 0) {
     printf("Error in allocation : code %u\n", ulErr);
     exit(1);
   } /* endif */

   if (cLetter == 'W' || cLetter == 'w')
   {
/* insert data into allocated memory */
      printf("\nWriting...\n");
      for (ulLoop = 0; ulLoop < ulAmount/4; ulLoop++)
      {
         *(pulBlock + ulLoop) = 7;
      } /* endfor */
   } else
   {
/* read data from allocated memory */
      printf("\nReading...\n");
      for (ulLoop = 0; ulLoop < ulAmount/4; ulLoop++)
      {
         if (*(pulBlock + ulLoop) != 7)
         {
            OK = FALSE;
         } /* endif */
      } /* endfor */
   } /* endif */

   ulErr = DosFreeMem (pulBlock);
   if (ulErr != 0)
   {
     printf("\nError in freeing : code %u\n", ulErr);
   } /* endif */
}
```

C.6 Exercise 3 - Multiple DOS Sessions

This exercise demonstrates how an OS/2 protected mode program may start a DOS program in a virtual DOS machine. Furthermore, the growth of the swap file is shown, as well as the decrease in the size of the swap file after termination of the OS/2 program.

The program used in this exercise makes use of the DosStartSession() function.

In this exercise, the student is required to run the memory lab program:

MEMLAB3.EXE

Syntax: MEMLAB3 <no. of DOS sessions>

The steps in the program are as follows:

1. If a parameter is passed to the program, it will use this as the number of DOS sessions to start. If no parameter is provided, four DOS sessions are started.
2. The program reads a file named: *MEMLAB3.PRO*.

A sample *MEMLAB3.PRO* is shown in C.7.2, "Sample Input File for MEMLAB3.EXE" on page 160.

The file must be created using an ASCII editor such as the OS/2 V2.0 System Editor and consists of three lines as follows:

a. The full path and file name of the swap file.
b. The name of the DOS program to be executed in each of the started sessions.

   In the example this is EDLIN.COM.
c. The parameter string which is to be passed to the DOS program.

   If no parameters are to be passed, an empty line must still appear in the file. In the example this is AUTOEXEC.BAT

3. The program starts the specified number of DOS sessions, if possible, with the current system resources. Before starting any sessions, the program displays the size of the swap file. After each session is started the new size of the swap file is displayed.
4. When the requested number of DOS sessions is started or the system resources are exhausted, the program will wait for a keystroke in order to terminate.
5. When the keystroke is received by the program all the DOS sessions are terminated. After terminating each session the swap file size is displayed. The program continues to monitor and display the swap file every 10 seconds until it remains unchanged for a period of 100 seconds.

The program listing is shown in C.7.1, "Source Code MEMLAB3.C" on page 158.

You should perform the following steps:

1. Although MEMLAB3 does give information about the swap file size, you may also want to use the SWAPSIZE program which is described in C.10, "Program to Display Swap File Size" on page 162 to monitor the changes in the size of SWAPPER.DAT. If so start this program now.
2. Run the MEMLAB3 program and notice how the swap file size increases as the sessions are started.
3. Terminate the program and watch the decrease in size of the swap file over time.

Note the 512KB steps in which the swap file size increases and decreases.

If you are able to do the exercise

Please do so now. If not, refer to the following explanation.

C.7 Expected Results from Exercise 3

After successfully completing the exercise, you should see the following results:

- The increase or decrease in the size of the swap file will always be in multiples of 512KB.
- If a large number of DOS sessions is specified, and a CPU-bound DOS program is started in each session, a heavy system load can be observed. This is because the DOS programs continue to execute even when in back-

ground. The system usage can be monitored using the **PULSE** application which can be found in the Productivity folder of the OS/2 System Folder which appears on the OS/2 Desktop.

If the time is available, you may wish to experiment by starting sessions with a type other than DOS full screen. MEMLAB3 will accept a second parameter which is then used as the SessionType parameter when calling the **DosStartSession** function.

```
Syntax: MEMLAB3 <no of sessions> <SessionType>
```

Session type may be any one of the values accepted by **DosStartSession**.

SessionType	Type of Session
1	OS/2 Full Screen
2	OS/2 Windowed
3	Presentation Manager
4	DOS Full Screen
7	DOS Windowed

Edit the MEMLAB3.PRO file to change the name of the program to be started and the parameters with which it must be started. Issue the command to start MEMLAB3 specifying both the number of sessions to start and the session type to be started.

C.7.1 Source Code MEMLAB3.C

```
/*************************************************************/
/*************************************************************/
/***                                                       ***/
/***  Program name: MEMLAB3.EXE                            ***/
/***                                                       ***/
/***  Created     : 7 May, 1990                            ***/
/***                                                       ***/
/***  Author      : Bo Falkenberg                          ***/
/***                                                       ***/
/***  Revised     : February, 1992 by Darryl Frost         ***/
/***                                                       ***/
/***  Purpose     : To demonstrate multiple sessions       ***/
/***                started from a OS/2 program, and       ***/
/***                to show how the size of the swap       ***/
/***                file varies as the sessions are        ***/
/***                started and what happens to the swap***/
/***                file after the sessions are stoppec.***/
/***                                                       ***/
/***  Compile     : icc /O+ /W2 memlab3.c                  ***/
/***                                                       ***/
/***  Execute     : memlab3 n x                            ***/
/***                where n = the number of sessions       ***/
/***                to be started. If nothing is           ***/
/***                entered 4 sessions will be started. ***/
/***                x is the type of sessions to start. ***/
/***                By default x is equal to 4, which      ***/
/***                means full screen DOS sessions will ***/
/***                be started. x can be any of the        ***/
/***                SessionType values recognized by       ***/
/***                DosStartSession.                       ***/
/***                                                       ***/
/***  File input  : Reads file MEMLAB3.PRO which must be***/
/***                setup using an editor such as the      ***/
/***                OS/2 System Editor. The file           ***/
/***                contains three lines. The first        ***/
/***                line contains the name of the swap     ***/
/***                file including its full path. The      ***/
/***                next line contains the name of the     ***/
/***                program which is to be executed in     ***/
/***                the sessions. The path used to         ***/
/***                find the program must be included.     ***/
/***                The last line  contains the            ***/
/***                parameter string which is to be        ***/
/***                passed to the program at start-up.     ***/
/***                                                       ***/
/*************************************************************/
/*************************************************************/

/*************************************************************/
/***  DEFINES                                              ***/
/*************************************************************/
 #define INCL_DOS
 #define LENGTH       sizeof(buffer)
                           /* DosFindFirst returned buffer
                              size */
 #define TIMEINTERVAL 10   /* Seconds to wait when checkinr
                              swap file size */
 #define MAXLOOP      10   /* No of intervals with same
                              swap file size after which
                              program in terminated */
 #define NOSESS       4    /* No of Sessions to start */
 #define DOS_FULL_SCREEN 4 /* Default Session Type    */

/*************************************************************/
/***  INCLUDE                                              ***/
/*************************************************************/
 #include        <os2.h>
 #include        <stdio.h>
 #include        <string.h>
 #include        <stdlib.h>

/*************************************************************/
/***  GLOBAL VARIABLES                                     ***/
/*************************************************************/
 ULONG           SessID;      /* Session ID (returned)   */
 PID             DOSpid;      /* Process ID (returned)   */
 USHORT          rc = 0;      /* return code             */
 USHORT          frc = 0;     /* file return code        */
 struct _STARTDATA StartData; /* start program structure */
 struct _FILEFINDBUF buffer;  /* file information struct */
 char szFname[64];
 char szProgname[64];
 char szProginp[64];
 FILE *fptr;
 PULONG pStartedSessID;
 ULONG *p;

/*************************************************************/
/***  FUNCTION PROTOTYPES                                  ***/
/*************************************************************/

void main(int argc, char *argv[], char *envp[]);
void printtrouble(void);
ULONG GetSwapperSize();
```

```
/*************************************************************/
/***  MAIN PROGRAM                                         ***/
/*************************************************************/

void main(int argc, char *argv[], char *envp[])
{
   int radix = 10;
   int loop;
   int no_of_SESS;
   int sesstype;
   char Related;
   unsigned char *Title1;
   unsigned char *Title2;
   char chloop1[30];
   char chloop2[30];
   char *dummy1;
   char *dummy2;
   char *pchrc;
   ULONG slen;
   ULONG fsize;
   ULONG ulrc;
   ULONG ulTargetOption;
   ULONG ulSessId;
   ULONG ulTimeInterval = TIMEINTERVAL * 1000;
   ULONG elapsed;
   ULONG loopflag;
   ULONG timecount;
   ULONG samecount;
   ULONG savesize;

/* Default no of sessions to start                          */
   no_of_SESS = NOSESS;
   sesstype = DOS_FULL_SCREEN;

/* Get arguments from the command line, if present          */
   if (argc >= 2)
   {
/* Number of sessions to start                              */
      no_of_SESS = atoi (argv[1]);
   }

   if (argc >= 3)
   {
/* Session type                                             */
      sesstype = atoi (argv[2]);
   }

/* Read parameters from MEMLAB3.PRO file                    */
   fptr = fopen("memlab3.pro", "r");
   if (fptr == (FILE *)NULL)
   {
      printf("\nFile MEMLAB3.PRO cannot be found\n");
      return;
   }
/* line 1 : swapper file path and filename                  */
   pchrc = fgets(szFname, sizeof(szFname)-1, fptr);
   if (pchrc == (char *)NULL)
   {
      printtrouble();
      return;
   }
   szFname[strlen(szFname)-1] = '\0';
/* line 2 : name of program to start in the sessions  */
   pchrc = fgets(szProgname, sizeof(szProgname)-1, fptr);
   if (pchrc == (char *)NULL)
   {
      printtrouble();
      return;
   }
   szProgname[strlen(szProgname)-1] = '\0';
/* line 3 : parameters to be passed to the program          */
   pchrc = fgets(szProginp, sizeof(szProginp)-1, fptr);
   if (pchrc == (char *)NULL)
   {
      printtrouble();
      return;
   }
   slen = strlen(szProginp);
   szProginp[slen-1] = '\0';

/* Set up parameter block for DosStartSession               */
   StartData.PgmName = szProgname;
   StartData.PgmInputs = szProginp;
   StartData.Length = 32;
   StartData.FgBg = 1;

   StartData.Related = 1; /* related to parent */

   StartData.TermQ = NULL;
   StartData.InheritOpt = 0;
   StartData.Environment = 0;
   StartData.PgmControl = SSF_CONTROL_MINIMIZE; /* Start Minimized */
   loop = 0;
   rc = 0;

/* Allocate memory to save IDs of started sessions          */
   slen = no_of_SESS * sizeof(ULONG);
   frc = DosAllocMem ((PPVOID)&pStartedSessID, slen,
                      PAG_WRITE | PAG_READ | PAG_COMMIT );
   if (frc != 0)
   {
      printf("Memory Allocation Failure, return code %u\n",frc);
      exit (1);
   }
   p = pStartedSessID;

   /* Keep starting sessions, until an error occurs       */
   /* or the requested number of sessions is reached.     */
   /* Save the IDs of the started sessions.               */
   /* Display the size of the swap file before starting   */
   /* any sessions and after each session is started.     */

   printf("Program MEMLAB3 is executing\n");
   printf("%u sessions will be started\n", no_of_SESS);
   fsize = GetSwapperSize();
   printf("Size of SWAPPER.DAT is now %u KB\n", fsize);
   while (!rc && loop < no_of_SESS)
   {
      loop++;
      StartData.SessionType = sesstype; /* session type       */
                              /* make the program title    */
      Title1 = ". SESS\n\0";      /* with a session number */
      dummy1 = _itoa(loop,  chloop1, radix);
      strcat (dummy1, Title1);
      StartData.PgmTitle = dummy1;

      rc = DosStartSession(&StartData, &SessID, &DOSpid);
      if (rc == 0)
      {
         printf("Session no %u is started; Session ID: %u\n",
               loop, SessID);
         fsize = GetSwapperSize();
         printf("Size of SWAPPER.DAT is now %u KB\n", fsize);
         *p++ = SessID;
       } else
       {
         printf("An error occurred starting Session no %u\n", loop);
         printf("Return code from DosStartSession = %u\n", rc);
         loop = no_of_SESS;
       } /* endif */
   } /* endwhile */

/* Wait for a key on the keyboard to be depressed, then   */
/* terminate the sessions. Display the swap file size */
/* after each session is terminated.                      */
   printf("Press <Enter> to terminate the Sessions...");
   fflush(stdout);
   loop = getchar();  /*wait for input */
   p = pStartedSessID;
   for (loop = 1; loop <= no_of_SESS; loop++)
   {
      ulSessid = *p++;
      ulTargetOption = 0;
      ulrc = DosStopSession (ulTargetOption, ulSessid);
      if (ulrc == 0)
      {
         printf("Session with ID %u has been stopped\n", ulSessid);

         fsize = GetSwapperSize();
         printf("Size of SWAPPER.DAT is now %u KB\n", fsize);
      }
   }

/* Monitor the swap file size and display it at intervals */
/* of TIMEINTERVAL seconds. When the size has remained    */
/* constant for TIMEINTERVAL * MAXLOOP seconds, terminate */
/* the program.                                           */
   loopflag = TRUE;
   savesize = fsize;
   timecount = 1;
   while (loopflag)
   {
      ulrc = DosSleep(ulTimeInterval);
      elapsed = timecount * TIMEINTERVAL;
      printf("Elapsed time since closing DOS sessions is %u seconds\n",
            elapsed);
      fsize = GetSwapperSize();
      printf("Size of SWAPPER.DAT is now %u KB\n", fsize);
      timecount++;
      samecount++;
      if ( fsize != savesize)
      {
         savesize = fsize;
         samecount = 0;
      }
      if (samecount == MAXLOOP - 1)
      {
         elapsed = MAXLOOP * TIMEINTERVAL;
         printf("No change in SWAPPER.DAT size for %u seconds\n"
                "Program is terminating\n",elapsed);
         loopflag = FALSE;
      }

   }
   exit(0);
}
```

```
/* Function to report errors with MEMLAB3.PRO          */
void printtrouble(void)
{
   printf("\nSorry, trouble reading MEMLAB3.PRO\n");
   fclose(fptr);
   return;
}

/* Function which returns swap file size in KB         */
ULONG GetSwapperSize ()
{
   HDIR fhandle;
   unsigned LONG count;
   int fsize;
   count = 1;
   fhandle = 0xFFFF;
   frc = DosFindFirst (szFname, &fhandle, 0, &buffer, LENGTH,
                       &count, 1L);
   if (frc != 0)
   {
      fflush(stdout);
      printf("File error :%u\n", frc);
      exit(0);
   } /* endif */
   fsize = buffer.cbFileAlloc / 1024;  /* in KBytes */
   DosFindClose (fhandle);
   return(fsize);
}
```

C.7.2 Sample Input File for MEMLAB3.EXE

The file, *MEMLAB3.PRO*, must appear in a directory accessible to MEMLAB3.EXE when it is executed.

```
C:\OS2\SYSTEM\swapper.dat
C:\os2\mdos\edlin.com
C:\autoexec.bat
```

C.8 Exercise 4 - Multiple Threads

This exercise demonstrates that an OS/2 program can start a very large number of threads (up to a maximum of 4096, minus those threads the system has already started). The program makes use of C Set/2 *_beginthread function*.

In this exercise, the student is required to run the memory lab program:

MEMLAB4.exe

Syntax: MEMLAB4 <number of threads>

If number of threads is not entered, ten (10) is assumed.

This program does the following:

1. Starts the specified number of threads.

   Each thread is very simple; it merely issues a DosSleep() function call and an occasional write to screen. This is in order that all threads may start within a reasonable time frame.

   For practical reasons, do not try to start more than 200 threads.

2. When all the threads are started, the program waits for keyboard input.

   At this time we suggest you issue the *PSTAT* command from an OS/2 Command Prompt. Use the "\C" option. This displays a list of all processes and threads open in the system. You can also see the number of threads in use for system functions.

3. When the program receives a keystroke, it sets a flag to request the threads to terminate. When all started threads have terminated, the program exits.

The program listing is shown in C.9.1, "Source Code MEMLAB4.C" on page 161.

If you are able to do the exercise

Please do so now. If not, refer to the following explanation.

The following error message may be generated:

DosCreateThread error = 164.

This typically occurs when the maximum allowed number of threads specified in CONFIG.SYS is less than the number of threads requested.

Note: Remember that the specified number of threads in the CONFIG.SYS file includes threads used by the operating system. This amounts to approximately 22 threads.

If the required number of threads cannot be started, check the CONFIG.SYS file. This file contains a specification for the maximum number of threads allowed in the system. Increase the threads command to: **THREADS = 4095**, and re-IPL the system.

C.9 Expected Results From Exercise 4

After successfully completing the exercise, you will observe the following results:

- The number of threads per process is not limited to 53 as in previous versions of OS/2.
- The maximum number of threads system wide is increased to 4096 threads.

C.9.1 Source Code MEMLAB4.C

```
/*************************************************************/
/*************************************************************/
/***                                                       ***/
/***  Program name: MEMLAB4.EXE                            ***/
/***                                                       ***/
/***  Created     : 7 May, 1990                            ***/
/***                                                       ***/
/***  Author      : Bo Falkenberg                          ***/
/***                                                       ***/
/***  Revised     : February, 1992 by Darryl Frost         ***/
/***                                                       ***/
/***  Purpose     : To demonstrate the new system          ***/
/***                limit for the number of threads        ***/
/***                per process in OS/2 2.0 and the        ***/
/***                effect of thread creation on the       ***/
/***                growth of SWAPPER.DAT.                 ***/
/***                                                       ***/
/***  Compile     : icc /W2 /Gm+ memlab4.c                 ***/
/***                                                       ***/
/***  Execute     : memlab4 n f                            ***/
/***                where n is the number of secondary     ***/
/***                threads the program must create,       ***/
/***                and f if present and 0 (zero) causes   ***/
/***                the created threads to terminate       ***/
/***                after 40 seconds else the threads      ***/
/***                after 40 seconds enter a wait and      ***/
/***                print loop. The wait time is           ***/
/***                randomly determined. If n is not       ***/
/***                specified 10 is assumed.               ***/
/***                                                       ***/
/*************************************************************/
/*************************************************************/



/*************************************************************/
/***  DEFINES                                              ***/
/*************************************************************/
#define INCL_DOS
#define INCL_DOSPROCESS


/*************************************************************/
/***  INCLUDE                                              ***/
/*************************************************************/
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>

void _Optlink NewThread( PVOID pThreadArg ); /* procedure declaration */

BOOL loopflag = TRUE;
int threadcount = 0;

/*************************************************************/
/***  MAIN PROGRAM                                         ***/
/*************************************************************/
main( int argc, char *argv[], char *envp[] )
{
  TID       ThreadID;                 /* thread identification */
  ULONG     ulThreadArg;              /* thread arguments      */
  ULONG     ulThreadFlags;            /* thread flags          */
  ULONG     ulStack_size;             /* thread stack size     */
  int       no_of_threads;            /* number of threads     */
  int       i;                        /* loop variable         */
                                      /*************************/

  ulThreadFlags = 0;          /* start thread immediately   */
  ulStack_size  = 1024;       /* give stack size in bytes   */
  ulThreadArg   = 1;

   if (argc < 2)
      {
        no_of_threads = 10;
      } else
      {
        no_of_threads    = atoi(argv[1]);
      }
  if (argc > 2)
  {
     if (*argv[2]=='0')
        loopflag = FALSE;
  }

  for (i = 1; i < no_of_threads+1; i++)
  {
    if ( ( ThreadID = _beginthread( NewThread, NULL, ulStack_size, (PVOID)i ) ) == -1 )
    {
      printf("_beginthread error\n");
      exit (1);
    }
    printf("Thread number %d created\n",i);
  }

  printf("To end the program press <CR> \n");
  getchar();
  loopflag = FALSE;
  printf("MEMLAB4 terminating\n");
/* Wait for all the threads to stop */
  while (TRUE)
  {
     DosSleep(1888);
     DosEnterCritSec();
     if(threadcount == 0){
        DosExitCritSec();
        printf("All threads stopped, program is terminated\n");
        exit (1);
     }
     DosExitCritSec();
  }
}

/*************************************************************/
/***  THREAD                                               ***/
```

```
/*************************************************************/
void NewThread( PVOID pThreadArg )
   {
   ULONG ulThreadArg = ( ULONG )pThreadArg;

   DosEnterCritSec();
   threadcount++;
   DosExitCritSec();

   printf( "Thread %u has started\n", ulThreadArg );
   srand( (int)ulThreadArg); /* seed random generator */
   DosSleep ( 40000 );       /* 40 SEC. sleep interval */

   while (!copflag)
      {
      printf("Thread %d just woke up\n", ulThreadArg);
      DosSleep ( rand() ); /* random sleep interval */
      }

   DosEnterCritSec();
   threadcount--;
   DosExitCritSec();
   }
```

C.10 Program to Display Swap File Size

Listed below is a small program, **SWAPSIZE**, which may be used to monitor the size of SWAPPER.DAT. The size is displayed in a PM window. The size of SWAPPER.DAT is queried every 10 seconds and the display updated when the size changes. The sampling interval can be changed to either 30 seconds or 60 seconds by selecting the *Interval* action bar. By default the program assumes the swap file to be C:\OS2\SYSTEM\SWAPPER.DAT. If this is not the case then when executing SWAPSIZE a command line parameter must be included which gives the full path and file name of the swap file.

C.10.1 Source Code SWAPSIZE.C

```
/*************************************************************/
/*************************************************************/
/***                                                       ***/
/***  Program name: SWAPSIZE.EXE                           ***/
/***                                                       ***/
/***  Created     : February, 1992                         ***/
/***                                                       ***/
/***  Revised     :                                        ***/
/***                                                       ***/
/***  Author      : Darryl Frost                           ***/
/***                                                       ***/
/***  Purpose     : To be used with the Lab Session        ***/
/***                Examples given in Appendix C of the    ***/
/***                OS/2 Version 2.0 Volume 1: Control     ***/
/***                Program Document no GG24-3730.         ***/
/***                This program interrogates and          ***/
/***                displays the size of the               ***/
/***                SWAPPER.DAT at regular intervals       ***/
/***                in a PM window. The interval is        ***/
/***                initially set to 10 seconds. This      ***/
/***                may be changed to 30 seconds or        ***/
/***                60 seconds by selecting the            ***/
/***                Interval action bar.                   ***/
/***                                                       ***/
/***  Parameter   : If the path of SWAPPER.DAT is other    ***/
/***                than C:\OS2\SYSTEM, the full path      ***/
/***                and file name of the swapper file      ***/
/***                must be the first parameter passed     ***/
/***                to the progam when it is started.      ***/
/***                                                       ***/
/*************************************************************/
/*************************************************************/
#define INCL_WIN
#define INCL_GPI

#include <os2.h>                        /* PM header file              */
#include <string.h>
#include <stdlib.h>
#include "swapsize.h"                   /* Resource symbolic identifiers*/

#define STRINGLENGTH 20                 /* Length of string            */

/*************************************************************************/
/* Function prototypes                                                   */
/*************************************************************************/
INT main(int argc, char *argv[], char *envp[] );
VOID AbortSS(HWND hwndFrame,HWND hwndClient);
MRESULT EXPENTRY SSWindowProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2 );
ULONG GetSwapperSize (void);
                                        /* Define parameters by type   */
HAB  hab;                               /* PM anchor block handle      */
HWND hwndClient=0L;                     /* Client area window handle   */
HWND hwndFrame=0L;                      /* Frame window handle         */
CHAR *szString;                         /* procedure.                  */
PSZ  pszErrMsg;
char szFname[64] = "C:\\OS2\\SYSTEM\\SWAPPER.DAT";
ULONG swapsize;
ULONG oldswapsize;
ULONG timerinterval = 10 * 1000;
ULONG idTimer = 1;
```

```
/*************************************************************************/
/*                                                                       */
/* MAIN initializes the process for OS/2 PM services, and processes the  */
/* application message queue until a WM_QUIT message is received. It     */
/* then destroys all OS/2 PM resources and terminates.                   */
/*                                                                       */
/*************************************************************************/
INT main (int argc, char *argv[], char *envp[])
{
  HMQ  hmq;                             /* Message queue handle        */
  QMSG qmsg;                            /* Message from message queue  */
  ULONG flCreate;                       /* Window creation control flags*/

  if ( argc >= 2 ) {
     strcpy( szFname, argv[1] );
  }

  if ((hab = WinInitialize(0)) == 0L) /* Initialize PM     */
     AbortSS(hwndFrame, hwndClient); /* Terminate the application    */

  if ((hmq = WinCreateMsgQueue( hab, 0 )) == 0L)/* Create a msg queue */
     AbortSS(hwndFrame, hwndClient); /* Terminate the application    */

  if (!WinRegisterClass(                /* Register window class       */
     hab,                               /* Anchor block handle         */
     (PSZ)"SSWindow",                   /* Window class name           */
     (PFNWP)SSWindowProc,               /* Address of window procedure */
     CS_SIZEREDRAW,                     /* Class style                 */
     0                                  /* No extra window words       */
     ))
     AbortSS(hwndFrame, hwndClient); /* Terminate the application    */

   flCreate = FCF_STANDARD &            /* Set frame control flags to  */
              FCF_SHELLPOSITION &       /* standard except for shell   */
              FCF_ACCELTABLE;           /* positioning.                */

  if ((hwndFrame = WinCreateStdWindow(
              HWND_DESKTOP,             /* Desktop window is parent    */
              0,                        /* STD. window styles          */
              &flCreate,                /* Frame control flag          */
              "SSWindow",               /* Client window class name    */
              "",                       /* Window text                 */
              0,                        /* No special class style      */
              (HMODULE)0L,              /* Resource is in .EXE file    */
              ID_WINDOW,                /* Frame window identifier     */
              &hwndClient               /* Client window handle        */
              )) == 0L)
     AbortSS(hwndFrame, hwndClient); /* Terminate the application    */

  WinSetWindowText(hwndFrame, "Swapper Size");

  if (!WinSetWindowPos( hwndFrame,      /* Shows and activates frame   */
                HWND_TOP,               /* window at position 100, 100, */
                100, 100, 100, 100,  /* and size 100, 100.          */
                SWP_SIZE | SWP_MOVE | SWP_ACTIVATE | SWP_SHOW
               ))
     AbortSS(hwndFrame, hwndClient); /* Terminate the application    */

/*************************************************************************/
/* Get and dispatch messages from the application message queue          */
```

```
/* until WinGetMsg returns FALSE, indicating a WM_QUIT message.        */
/*************************************************************************/

  while( WinGetMsg( hab, &qmsg, 0L, 0, 0 ) )
    WinDispatchMsg( hab, &qmsg );
  WinDestroyWindow(hwndFrame);          /* Tidy up...                  */
  WinDestroyMsgQueue( hmq );            /* Tidy up...                  */
  WinTerminate( hab );                  /* Terminate the application   */
} /* End of main */

/*************************************************************************/
/*                                                                       */
/*  SSWINDOWPROC is the window procedure which continuously monitors the size */
/*  of the swap file and displays the size in the client area.           */
/*                                                                       */
/*************************************************************************/
MRESULT EXPENTRY SSWindowProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2 )
{

  switch( msg )
  {
    case WM_CREATE:
      swapsize = GetSwapperSize();

      oldswapsize = swapsize;
      WinStartTimer(hab, hwnd, idTimer, timerinterval);
      break;

    case WM_COMMAND:
      {
      USHORT command;                   /* WM_COMMAND command value    */
      command = SHORT1FROMMP(mp1);      /* Extract the command value   */
      switch (command)
      {
        case ID_10SECS:
          timerinterval = 10 * 1000;
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_10SECS, TRUE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_30SECS, FALSE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_60SECS, FALSE);
          WinStartTimer(hab, hwnd, idTimer, timerinterval);
          break;
        case ID_30SECS:
          timerinterval = 30 * 1000;
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_10SECS, FALSE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_30SECS, TRUE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_60SECS, FALSE);
          WinStartTimer(hab, hwnd, idTimer, timerinterval);
          break;
        case ID_60SECS:
          timerinterval = 60 * 1000;
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_10SECS, FALSE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_30SECS, FALSE);
          WinCheckMenuItem(WinWindowFromID(hwndFrame,FID_MENU),
                           ID_60SECS, TRUE);
          WinStartTimer(hab, hwnd, idTimer, timerinterval);
          break;
        default:
          return WinDefWindowProc( hwnd, msg, mp1, mp2 );
      }
      break;
      }
   case WM_TIMER:
      swapsize = GetSwapperSize();
      if (swapsize==0) {
         WinMessageBox(HWND_DESKTOP,        /* Parent window is desk top */
           hwndFrame,             /* Owner window is our frame */
           "Cannot locate the Swapper File, Check your parameter.",
           "Error Msg",           /* Title bar message         */
           MSGBOXID,              /* Message identifier        */
           MB_MOVEABLE | MB_CUACRITICAL | MB_CANCEL ); /* Flags */
         WinPostMsg( hwnd, WM_CLOSE, (MPARAM)0,(MPARAM)0 );
      }
      if (swapsize != oldswapsize)
      {
         oldswapsize = swapsize;
         WinInvalidateRect( hwnd, (PRECTL)NULL, FALSE);
      }
      break;
    case WM_ERASEBACKGROUND:
      /*******************************************************************/
      /* Return TRUE to request PM to paint the window background        */
      /* in SYSCLR_WINDOW.                                               */
      /*******************************************************************/
      return (MRESULT)( TRUE );
    case WM_PAINT:
      /*******************************************************************/
      /* Window contents are drawn here in WM_PAINT processing.          */
      /********************* ***********************************************/
      {
      HPS    hps;                       /* Presentation Space handle    */
      RECTL  rc;                        /* Rectangle coordinates        */
      POINTL pt;                        /* String screen coordinates    */
                                        /* Create a presentation space  */
      char   buffer[STRINGLENGTH];
      char   temp[STRINGLENGTH];
      int len;

      hps = WinBeginPaint( hwnd, 0L, &rc );
      pt.x = 50; pt.y = 20;             /* Set the text coordinates,    */
      GpiSetColor( hps, CLR_NEUTRAL );  /* colour of the text,          */
      GpiSetBackColor( hps, CLR_BACKGROUND ); /* its background and     */
      GpiSetBackMix( hps, BM_OVERPAINT );   /* how it mixes,            */
                                        /* and draw the string...*/
      WinFillRect( hps, &rc, CLR_BACKGROUND );
      szString = _ltoa (swapsize, buffer, 10);
      if ((len=strlen(buffer)) > 3) {
         memcpy(temp, buffer, len+1);
         memcpy(buffer, temp, len-3);
         buffer[len-3] = ' ';
         strcpy(&buffer[len-2], &temp[len-3]);
         }

      strcat( szString, " KB");
      GpiCharStringAt( hps, &pt, (LONG)strlen( szString ), szString );
      WinEndPaint( hps );                   /* Drawing is complete  */
      break;
      }
    case WM_CLOSE:
      /*******************************************************************/
      /* This is the place to put your termination routines              */
      /*******************************************************************/
      WinStopTimer( hab, hwnd, idTimer);
      WinPostMsg( hwnd, WM_QUIT, (MPARAM)0,(MPARAM)0 );/* Cause termination*/
      break;
    default:
      /*******************************************************************/
      /* Everything else comes here.  This call MUST exist               */
      /* in your window procedure.                                       */
      /*******************************************************************/

      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return (MRESULT)FALSE;
} /* End of MyWindowProc */

/*************************************************************************/
/* AbortSS    -- report an error returned from an API service.           */
/*                                                                       */
/* The error message is displayed using a message box and the program    */
/* exit will be affected with the error status of 1.                     */
/*************************************************************************/
VOID AbortSS(HWND hwndFrame,HWND hwndClient)
{
PERRINFO  pErrInfoBlk;
PSZ       pszOffSet;
void      stdprint(void);

      DosBeep(100,10);
      if ((pErrInfoBlk = WinGetErrorInfo(hab)) != (PERRINFO)NULL)
      {
      pszOffSet = ((PSZ)pErrInfoBlk) + pErrInfoBlk->offoffszMsg;
      pszErrMsg = ((PSZ)pErrInfoBlk) + *((PSHORT)pszOffSet);
      if((INT)hwndFrame && (INT)hwndClient)
         WinMessageBox(HWND_DESKTOP,        /* Parent window is desk top */
                       hwndFrame,           /* Owner window is our frame */
                       (PSZ)pszErrMsg,      /* PMWIN Error message       */
                       "Error Msg",         /* Title bar message         */
                       MSGBOXID,            /* Message identifier        */
                       MB_MOVEABLE | MB_CUACRITICAL | MB_CANCEL ); /* Flags */
      WinFreeErrorInfo(pErrInfoBlk);
      }
      WinPostMsg(hwndClient, WM_QUIT, (MPARAM)0, (MPARAM)0);
} /* End of AbortSS */
/* Function which returns swap file size in KB          */
ULONG GetSwapperSize ()
{
   HDIR fhandle;
   unsigned LONG count;
   int fsize;
   USHORT frc;
   FILEFINDBUF buffer;  /* file information struct */

   count = 1;
   fhandle = 0xFFFF;
   frc = DosFindFirst
     (szFrame, &fhandle, 0, &buffer, sizeof(buffer),                    &count, 1L);
   if (frc != 0){
      return(0L);
   } /* endif */
   fsize = buffer.cbFileAlloc / 1024;  /* in KBytes */
   DosFindClose (fhandle);
   return((ULONG)fsize);
}
```

C.10.2 Include File SWAPSIZE.H

```
#pragma linkage (main,optlink)
#define MSGBOXID    1001

#define ID_WINDOW   256

#define ID_INTERVAL  257
#define ID_10SECS  258
#define ID_30SECS  259
#define ID_60SECS  260
```

C.10.3 Resource File Source SWAPSIZE.RC

```
#include <os2.h>
#include "swapsize.h"

ICON    ID_WINDOW swapsize.ico

MENU    ID_WINDOW PRELOAD
BEGIN
  SUBMENU " Interval", ID_INTERVAL  BEGIN
    MENUITEM "10 Seconds", ID_10SECS, MIS_TEXT ,MIA_CHECKED
    MENUITEM "30 Seconds", ID_30SECS, MIS_TEXT
    MENUITEM "60 Seconds", ID_60SECS, MIS_TEXT
  END
END
```

C.10.4 MAKE File SWAPSIZE.MAK

```
#-------------------------------------------------------------------
#
#   Sample application makefile,common definitions for the IBM C
#   compiler environment
#-------------------------------------------------------------------
FFIXES:
FFIXES: .rc .res .obj .lst .c
#-------------------------------------------------------------------
# Default compilation macros for sample programs
#
# Compile switchs  that are enabled
# /c      compile don't link
# /Gm+    use the multi-threaded libraries
# /ss     allow  "//" for comment lines
# /Ms     use the system calling convention and not optilink as
#         the default
# /Gd-    Disable optimization
# /Se     allow cset  extensions
#
#     accordingly.
#

CC        = icc /c /Gd- /Se /Re /ss /Ms /Gm+


AFLAGS  = /Mx -t -z
ASM     = ml /c /Zm
LFLAGS   = /NOE /NOD /ALIGN:16 /EXEPACK /M /De
LINK    = LINK386  $(LFLAGS)
LIBS    = DDE4MBS + OS2386
STLIBS  = DDE4SBS + OS2386
MTLIBS  = DDE4MBS + DDE4MBM  + os2386
DLLLIBS = DDE4NBS + os2386
VLIBS   = DDE4SBS + vdh + os2386

.c.lst:
    $(CC) -Fc$*.lst -Fo$*.obj $*.c

.c.obj:
    $(CC) -Fo$*.obj $*.c

.rc.res:
        rc -r -p -x $*.rc

HEADERS = swapsize.h

#-------------------------------------------------------------------
#   A list of all of the object files
#-------------------------------------------------------------------
ALL_OBJ1 = swapsize.obj


all: swapsize.exe


swapsize.l: swapsize.mak
    echo $(ALL_OBJ1)            > swapsize.l
    echo swapsize.exe           >> swapsize.l
    echo swapsize.map           >> swapsize.l
    echo $(LIBS)                >> swapsize.l
    echo swapsize.def           >> swapsize.l



swapsize.res: swapsize.rc swapsize.ico swapsize.h

swapsize.obj: swapsize.c $(HEADERS)

swapsize.exe: $(ALL_OBJ1)  swapsize.def swapsize.l swapsize.res
    $(LINK) @swapsize.l
    rc swapsize.res swapsize.exe
```

C.10.5 Module Definition File SWAPSIZE.DEF

```
NAME    SWAPSIZE WINDOWAPI

DESCRIPTION 'OS/2 2.0 Control Program Labs SWAPSIZE Program'

STUB    'OS2STUB.EXE'

DATA    MULTIPLE

STACKSIZE   8192
HEAPSIZE    4096

PROTMODE
```

C.10.6 Linkage Parameters SWAPSIZE.L

```
swapsize.obj
swapsize.exe
swapsize.map
DDE4MBS + OS2386
swapsize.def
```

Glossary

address conversion. The process of converting a 0:32 memory reference to the 16:16 addressing scheme, and vice versa.

address translation. **(1)** The process of resolving a 0:32 memory reference into a physical memory address. When using the paged memory option in the 80386 processor, a memory pointer passed by an application consists of Page Directory and Page Table entries, and an offset within a physical page. This is resolved by the processor into a 32-bit physical memory address. The validity and legality of the memory reference is also checked during the translation process, and a general protection exception is generated if necessary. **(2)** The process of resolving a 16:16 memory reference into a physical memory address using a process's local descriptor table. The validity and legality of the memory reference is also checked during the translation process, and a general protection exception is generated if necessary.

alias. Term used in the 80386 segmented memory model, to refer to the case where two different addresses reference the same physical memory location; the location is said to be *aliased*. This technique is used when sharing memory between processes, and when mapping memory references between 16:16 and 0:32 addressing schemes.

ANSI. American National Standards Institute; U.S.-based organization which defines standards for computing devices, protocols, programming languages etc.

API. Application Programming Interface; term used to describe the set of functions by which an application may gain access to operating system services.

arena. Refers to a contiguous subset of the processor's virtual address space. In OS/2 V2.0, arenas are used to manage regions of the linear address space.

Bidi. Bidirectional; term used to describe support for national languages such as Arabic and Hebrew, which use bidirectional character sets.

BIOS. Basic Input/Output System; code which controls the interface between a system and its attached devices, at the hardware level.

bit. A binary digit, which may be either zero or one. Bits are represented within a computing device by the presence or absence of an electrical or magnetic pulse at a particular point, indicating a one or a zero respectively.

Boot Manager. Feature of OS/2 Version 2.0 which allows multiple partitions to exist on fixed disks in the same machine, with a separate operating system on each partition. At boot time, the user may select the desired operating system with which to start the machine.

boot name. A name used to refer to a selectable logical drive under Boot Manager, from which an operating system may be started.

byte. A logical data unit composed of eight binary digits (bits).

cdecl. Keyword used in C programming, which causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in right-to-left order, and the calling routine clears the stack after control returns from a called function or subroutine. This is the default for C programs. Contrast this with the *pascal* keyword.

CD-ROM. Compact Disk Read-Only Memory; technology where data is stored on an optical disk for reading by a computer system equipped with an appropriate reading device. CD-ROM storage media may not be updated by the computer system, although certain implementations allow the media to be erased and rewritten.

compatibility region. In the OS/2 Version 2.0 flat memory model, the address region below 512MB, which may be addressed by a 16-bit application using the 16:16 addressing scheme and tiled local descriptor tables. Under OS/2 Version 2.0, this region is equivalent in size to the process address space.

compatibility region mapping algorithm. Algorithm used in thunks to convert 16:16 memory references to the 0:32 addressing scheme and vice versa.

CRMA. See compatibility region mapping algorithm.

DDE. Dynamic Data Exchange; interprocess communication protocol used by applications to define dynamic links. Information updated in one application is automatically reflected in other applications linked to the first application via DDE.

debugging. The process of removing "bugs" or errors from application code.

device driver. Code which handles the translation of generic device commands into specific commands for the required physical device and vice versa, allowing operating system interaction with physical devices attached to the system.

DLL. Dynamic Link Library; application module containing routines and/or resources, which is dynamically linked with its parent application at load time or

runtime rather than during the linkage editing process. The use of DLLs enables decoupling of application routines and resources from the parent program, enhancing code independence, facilitating maintenance and reducing resident memory consumption.

DMA. Direct Memory Addressing; technique by which transfers to and from system memory are made by an independent control chip rather than by the system's main processor, thereby resulting in improved overall performance.

DOS. Disk Operating System; generally used in reference to IBM PC DOS, the single-tasking 16-bit real-mode operating system designed for Intel 8086 processors, and developed by Microsoft Corporation as MS DOS in the early 1980s. IBM subsequently licensed MS DOS for use on IBM Personal Computer and Personal System/2 machines, and has since undertaken joint development of later versions of the operating system in conjunction with Microsoft.

DosDebug(). Function provided within the application programming interface under OS/2 Version 2.0, which enables access to the debugging functions provided by the operating system.

ESDI. Enhanced Small Device Interface; physical access protocol used by many computer storage devices, particularly fixed disk devices.

extended attributes. Information which may be associated with a file under OS/2 Version 1.2 or above (including Version 2.0), to indicate various properties of that file. Extended attributes are available with both the FAT and HPFS file systems. An application may define extended attributes for files which it creates, and may update the extended attributes of files upon which it operates. A number of standard extended attributes are defined by the operating system for commonly-used information. Extended attributes and their uses are described in the *IBM OS/2 Version 2.0 Control Program Reference*.

extended partition. In the context of Boot Manager, a fixed disk partition which contains one or more logical drives, and on which certain operating systems such as OS/2 Version 2.0 may be installed. An extended partition and its logical drives are visible to the operating system at all times, regardless of which primary partition is currently active. Hence extended partitions are useful for the storage of programs and data which must be shared between multiple operating systems in an OS/2 Version 2.0 Boot Manager environment.

far. Term used to describe a memory reference in the 16:16 addressing scheme, where the memory location to be referenced is outside the current memory segment. Contrast with *near* memory references.

FAT. File Allocation Table; term used to describe the file system implemented by DOS and OS/2. This file system uses a file allocation table to contain the physical sector addresses of all files on the disk. The FAT file system is supported by OS/2 Version 2.0, along with the newer HPFS and other installable file systems.

flat memory model. Conceptual view of real memory implemented by OS/2 Version 2.0, where the operating system regards memory as a single linear address range of up to 4GB.

free swap frame linked list. Linked list in memory which contains unused page frames from the swap frame array. These frames may be used by the operating system to store references to new pages which must be swapped to disk. The linked list improves performance by avoiding the need to search the swap frame array for a free swap frame.

FSD. File System Driver; code within the operating system which supports the implementation of a particular file system such as FAT or HPFS.

gate. Specialized form of segment base address in memory, to which control is passed by an application when that application requires the execution of code which must run at a different privilege level. The gate routine then redirects control to the appropriate routine to perform the function. The use of gates insulates the calling application from the called routine, thereby providing greater protection.

GB. Gigabyte; 1024 megabytes, or 1024 x 1024 x 1024 bytes.

GDT. See global descriptor table.

general protection exception. Operating system error which occurs when an application attempts to access memory in a page which has not been allocated to that process. OS/2 Version 2.0 allows an application to trap a general protection exception using an exception handler registered with the operating system. If an exception handler is not registered, the operating system will terminate the application as a result of a general protection exception. Also known as a Trap 000D.

global descriptor table. Table in memory used under the segmented memory model, to store the segment base addresses of all memory segments in the system. This table is used by the operating system when allocating and controlling memory, and is not available to processes executing in the system.

guard page. Page within a memory object, for which the PAG_GUARD attribute has been specified. Any attempt by the application to reference memory within the guard page results in a guard page exception.

guard page exception. Operating system warning condition which occurs when an application accesses memory within a page which has been declared as a guard page. This exception may be trapped using an exception handler registered by the application in order to handle such occurrences. The typical processing performed by the exception handler is to commit more memory within the memory object. If an exception handler is not registered, the operating system's default handler commits the next available page within the memory object and sets its attribute to PAG_GUARD.

guard page technique. Technique by which the size of a memory object is progressively increased during application execution, by using guard pages and trapping the resulting guard page exceptions in order to commit additional pages of memory.

HPFS. High Performance File System; file system first implemented under OS/2 Version 1.2, offering enhanced performance over the original FAT file system implemented in DOS and prior versions of OS/2. HPFS is an optional installation item under OS/2 Version 2.0; the FAT system may also be used to retain compatibility with DOS.

IDT. See interrupt descriptor table.

IFS. See Installable File System

Information Presentation Facility. The Information Presentation Facility is a tool that enables you to create online information, to specify how it will appear on the screen, to connect various parts of the information and to provide help information that can be requested by the user.

interrupt. An electrical signal generated by a device or adapter within the system, to inform the operating system that an event, such as the completion of an I/O operation, has occurred. The operating system then processes the interrupt by passing control to a particular piece of code which handles the appropriate action in response to the event indicated.

Installable File System. An installable file system is a file system that is loaded by the operating system during system initialization, because the user has included statements in the CONFIG.SYS to request installation of the file system.

interrupt descriptor table. Table in memory which contains the addresses of processing routines for all defined interrupt levels in the system. When a device signals an interrupt to the operating system, the interrupt contains a code which is used as an index into the interrupt descriptor table.

I/O. Input/output; term used to collectively describe the techniques and devices through which a computer system interfaces with storage devices, external systems and the user.

IOPL. Input Output Privilege Level; term used in Intel 80x86 processor terminology to refer to tasks executing at privilege level 2, which have the authority to directly access physical I/O devices.

IPF. See Information Presentation Facility

KB. Kilobyte; 1024 bytes.

LDT. See local descriptor table.

local descriptor table. Table in memory used under the segmented memory model, to store the segment base addresses for memory segments used by the current process.

logical drive. In the context of Boot Manager, a subdivision of a fixed disk partition, which is regarded by an operating system as a logical entity. Logical drives are typically accessed using logical drive letters (for example, "C"). Logical drives are used to store operating system code, programs and data files. A primary partition may only contain a single logical drive, while an extended partition may contain multiple logical drives.

MB. Megabyte; 1024 kilobytes, or 1024 x 1024 bytes.

MBB. Multi-Boot Block; 16KB area at the start of a physical disk drive (immediately after the MBR), which contains code and data for use by Boot Manager. The MBB is located outside any logical drive, and is not normally accessible by an operating system or application.

MBR. Master Boot Record; the first sector or sectors of a physical disk drive, used to contain partition information and code which accesses the operating system boot record in the active partition. In a Boot Manager environment, the MBR does not access the operating system boot record, but instead accesses an entry point in the MBB, in order to load Boot Manager.

memory object. Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint. A memory object may be up to 512MB in size under OS/2 Version 2.0.

Multiple Virtual DOS Machines. Feature of OS/2 Version 2.0 which enables multiple DOS applications to execute concurrently in full-screen or windowed mode under OS/2 Version 2.0, in conjunction with other 16-bit or 32-bit applications, with full preemptive multitasking and memory protection between tasks. See also virtual DOS machine.

multiplexed streaming data mode. Mode of operation for 32-bit Micro Channel machines, whereby the streaming data mode is extended by using the 32 address lines, unused in transfer cycles after the first,

for the transfer of data. This mode effectively provides a 64-bit data path, resulting in faster memory transfers.

MVDM. See Multiple Virtual DOS Machines.

near. Term used to describe a memory reference in the 16:16 addressing scheme, where the memory location to be referenced is within the current memory segment. In the 0:32 addressing scheme, all memory references are effectively near references. Contrast with *far* memory references.

NPX. Numeric Processor Extension; term used in reference to the exception condition generated by the 80386 processor when an application issues a numeric coprocessor instruction in a machine with no coprocessor installed. Note that OS/2 Version 2.0 will trap the NPX exception and emulate the numeric coprocessor function within the operating system, returning the result to the application exactly as if a physical coprocessor were installed.

NULL. A binary zero. In C programming terms, NULL is typically used to refer to a pointer which is set to the binary zero value.

page. Granular unit for memory management using the 80386 and 80486 processors. A page is a 4KB contiguous unit of memory, which the processor manipulates as a single entity for the purpose of memory manipulation and swapping.

page ager. Component of the operating system which, when the number of free memory pages drops below a threshold level, begins to take pages which have not recently been accessed and change them to the *idle* state. Such pages may then be swapped out to disk if required.

page directory. Table used by OS/2 Version 2.0 to manage storage of pages in physical memory. One page directory exists for each process in the system, and contains entries which point to page tables, which in turn contain the physical page addresses.

page directory base address. Memory address of the page directory for the current process; stored in the page directory base register.

page directory base register. Control register used to store the base address of the page directory for the current process. The page directory base register is stored in the process's task state segment when a task switch occurs.

page fault exception. Operating system error which occurs when an application attempts to access memory which has been allocated but not committed. This exception may be trapped by an application using an exception handler registered with the operating system. If an exception handler is not registered, the operating system's default handler will terminate the application as a result of a page fault exception. Also known as a Trap 000E.

page frame. Data structure maintained by the operating system to store information concerning a physical page in memory. Page frames may be *free*, *idle*, or *in-use*.

page offset. 12-bit field used to specify the offset of a required memory location, within a physical page in real memory.

page table. Table used by OS/2 Version 2.0 to manage storage of pages in physical memory. Multiple page tables exist for each process in the system, and are referenced by the process's page directory. Each entry in the page table contains control information for a single physical page.

partition. An area of a fixed disk. A partition is composed of one or more logical drives, and is used to store the operating system code, programs and data files. A partition may be a primary partition or an extended partition.

pascal. Keyword used in C programming, which causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in left-to-right order, and the stack is cleared by the called routine when control is passed from one routine to another. Originally introduced by Microsoft with early versions of Microsoft Windows, when this convention saved several hundred bytes of system memory. Contrast this with the *cdecl* keyword.

PDBR. See page directory base register.

PIB. Process Information Block; data structure used to store process-specific control information. This structure may be accessed by an application using the **DosGetInfoBlocks()** function. See also TIB.

PIC. Programmable Interrupt Controller; component of the 80386 processor complex which handles interrupts generated by devices within the system.

POST. Power-On Self-Test; code typically stored on ROM (although the IBM PS/2 Model 90 and 95 allow POST code to be stored on fixed disk) which is invoked when a machine is powered on, in order to test the hardware.

primary partition. In the context of Boot Manager, a fixed disk partition which contains a single logical drive, and on which an operating system may be installed. Only one primary partition may be active (visible) on a fixed disk at any one time; other primary partitions are hidden from the operating system. Operating systems such as DOS and OS/2 Version 1.x require a primary partition for their installation; OS/2 Version 2.0 may be installed on a primary partition or an extended partition.

privilege level. In the context of the Intel 80386 processor architecture, the level of authority at which a task executes. There are four available privilege levels; under OS/2 Version 2.0, Level 0 is used for operating system kernel code; Level 1 is not used; Level 2 is used for applications which directly address I/O devices (such as communications applications); Level 3 is used for general application code.

process address space. Region of memory addressable by a single process under OS/2 Version 2.0; each process address space may be up to 512MB in size.

protected mode. Mode of operation for the Intel 80286 and 80386/80486 processors, whereby the address space is expanded to 16MB (80286) or 4GB (80386/80486), and memory references are translated via segment selector and offset, enabling full memory protection between processes executing in the system. With the 80386/80486, paging is available in protected mode.

RAM. Random Access Memory; term used to describe memory which may be dynamically read and written by a processor or other device during system operations. RAM is typically used to store program instructions and data which not being operated upon by the processor at the current moment in time, but which are required for the logical unit of work currently being carried out.

real mode. Default mode of operation for the Intel 80286 and 80386 processors, and the only mode of operation for the 8086 processor. In real mode, the processor acts as a 16-bit device, its physical memory address space is limited to 1MB, and memory references translate directly to physical addresses. With the 80386, paging is not supported in real mode.

ROM. Read-Only Memory; term used to describe memory which may be read, but not written to, during system operations. ROM is typically used to store basic hardware initialization instructions, BIOS or self-testing code, which is required to be available prior to accessing the disk subsystem.

SCB. Subsystem Control Block; data structure used at the hardware level to communicate between devices in a computer system. The structure of SCBs is defined by the Subsystem Control Block Architecture.

SCSI. Small Computer Systems Interface; interface standard defined by ANSI SCSI standard X3.131-1986, which defines a bus architecture and communication protocols for interaction of up to eight computing devices.

segment. Unit of memory addressable by the Intel 80x86 processors. With the 8086 and 80286 processors, a segment may be from 16 bytes to 64KB in size. With the 80386 and 80486 processors, a segment may be up to 4GB in size.

segment selector. Field which specifies the base address of a memory segment when using the segmented memory model. The selector is 16 bits in length on an 80286 processor, and 32 bits in length on an 80386 or 80486 processor.

semaphore. Construct used under OS/2 Version 2.0. and previous versions of OS/2 to enable synchronization between processes and between threads in the same process. OS/2 Version 2.0 provides enhanced semaphore facilities over previous versions.

service layer. Executable code which performs the operating system function requested by an application using an API.

signal. An event occurring within the operating system which will affect the execution of the current process; for example, the user may hit the Ctrl+Break key combination, which would result in a signal being generated, instructing the operating system to terminate the current process. Signals typically override the dispatching algorithms of the operating system.

sparse object. Memory object for which a linear address range has been reserved, but for which no physical memory has yet been committed. This capability is used to reserve storage in the process address space for use by an application, without causing an adverse impact on system performance by requesting large amounts of physical memory.

streaming data mode. Mode of operation for memory transfer on a Micro Channel machine, which allows the transfer of multiple contiguous blocks of data without the necessity to explicitly specify the address of each block. This allows faster transfer operations. See also multiplexed streaming data mode.

system region. In the OS/2 Version 2.0 flat memory model, the address region above 512MB, which is reserved for operating system use. See also process address space.

TCB. Thread control block; data structure used by OS/2 to store control information relating to threads in the system.

TIB. Thread information block; data structure used to store thread-specific control information. This structure may be accessed by an application using the **DosGetInfoBlocks()** function. See also PIB.

thunk. Term used to describe a routine which performs conversion of 16:16 memory references to the 0:32 addressing scheme, and vice versa.

tiled local descriptor table. Form of local descriptor table used by 16-bit applications running in the 16/32-bit region under OS/2 Version 2.0, in order to

allow access to the 512MB process address space using the 16:16 addressing scheme.

transaction lookaside buffer. Caching buffer used by the 80386 processor to store the physical addresses of most-recently used pages, in order to improve performance by avoiding the need to reference page directories and page tables to determine the physical memory address.

trap 000D. See general protection exception.

trap 000E. See page fault exception.

task state segment. Structure used to store the control information for a system task when a task switch occurs. When the task becomes active once again, registers and other control information are loaded from the TSS.

TSS. See task state segment.

virtual device driver. Form of device driver used by DOS applications executing in a DOS virtual machine, in order to access devices which must be shared with other processes in the system, such as the screen or mouse. The virtual device driver maps DOS device commands to the normal (physical) device driver under OS/2 Version 2.0.

virtual page structure. Data structure maintained for each committed page in the system. The virtual page structure is referenced by the Page Table Entry for the page, and contains information including the location of the physical page frame if the page is in memory, or its location within the swap file if the page is swapped out.

virtual DOS machine. A protected mode process under OS/2 Version 2.0 which emulates a DOS operating system environment, such that DOS applications executing within the virtual machine operate exactly as if they were running under DOS. DOS virtual machines support both text and graphics applications. Virtual DOS machines make use of the virtual 8086 mode of the 80386 and 80486 processors.

virtual machine. See virtual DOS machine.

virtual 8086 mode. Mode of operation of the Intel 80386 and 80486 processors, which allows the processor to execute multiple concurrent tasks with each regarding the processor as its own distinct 8086 processor. This mode of operation provides full preemptive multitasking and full memory protection between the virtual 8086 tasks.

0:32. Term used to describe the addressing scheme used for the 32-bit flat memory model, where a memory address is expressed as a 32-bit offset within the linear address range.

16:16. Term used to describe the addressing scheme used for the 16-bit segmented memory model, where a memory address is expressed as a 16-bit segment selector, and a 16-bit offset within that segment.

16-bit. Term used to describe an application which uses the 16:16 addressing scheme implemented under DOS and previous versions of OS/2. In fact, such applications use a 24-bit address since the segment selector and offset are normally overlapped. Such applications typically use the 16-bit instruction set implemented under the Intel 80286 processor.

16/32-bit region. See compatibility region.

32-bit. Term used to describe an application which uses the 0:32 addressing scheme implemented under OS/2 Version 2.0. Such applications may make full use of the 80386 instruction set.

80386. Intel 80386 microprocessor; the 32-bit processor upon which the OS/2 Version 2.0 operating system is based.

80486. Intel 80486 microprocessor; a 32-bit processor which implements a superset of the 80386 processor instruction set.

_far16. Keyword used in C programming language to indicate that conversion to 16-bit internal representation is required during compilation.

Index

Numerics

A

B

C

D

E

F

G

H

I

L

M

Readers' Comments

OS/2 Version 2.0
Volume 1: Control Program
Publication No. GG24-3730-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name _____ Address _____

Company or Organization _____ _____

Phone No. _____ _____

IBM®

Cut o
Alon

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Center
Department 91J, Building 235-2
Internal Zip 4423
901 NORTHWEST 51ST STREET
BOCA RATON FL 33431-1328

Fold and Tape | **Please do not staple** | Fold and Tape

Cut o
Along

Readers' Comments

OS/2 Version 2.0
Volume 1: Control Program
Publication No. GG24-3730-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name _____ Address _____

Company or Organization _____ _____

Phone No. _____ _____

Readers' Comments

OS/2 Version 2.0
Volume 1: Control Program
Publication No. GG24-3730-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

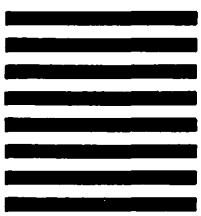
Company or Organization

Phone No.

Cut
Alo

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Center
Department 91J, Building 235-2
Internal Zip 4423
901 NORTHWEST 51ST STREET
BOCA RATON FL 33431-1328

Fold and Tape | **Please do not staple** | Fold and Tape

Cut
Alor

GG24-3730-00

OS/2 Version 2.0 Volume 1: Control Program

GG24-3730-00

PRINTED IN THE U.S.A.

IBM®