Microsoft® Operating System/2

Programmer's Guide

Microsoft Corporation

Pre-release

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1987

Microsoft®, MS-DOS®, and XENIX® are registered trademarks of Microsoft Corporation.

IBM®, Personal Computer AT®, and PC-DOS® are registered trademarks of International Business Machines Corporation.

INTEL® is a registered trademark of Intel Corporation.

Document Number 510830021-100-O00-0487
Part Number 110-098-014

Contents

Figures

Tables

How to Use These Manuals

Introduction

The Microsoft® Operating System/2 (MS OS/2) manual set describes MS OS/2, a multitasking operating system for the Intel 80286 microprocessor. Three manuals in the set discuss the operating system, and are intended for the novice user:

- *Microsoft Operating System/2 Setup Guide*
- *Microsoft Operating System/2 Beginning User's Guide*
- *Microsoft Operating System/2 User's Reference*

The other three manuals discuss topics of interest to programmers:

- *Microsoft Operating System/2 Programmer's Guide* (this manual)
  For programmers, a tutorial introduction to MS OS/2
- *Microsoft Operating System/2 Programmer's Reference*
  A reference for all MS OS/2 functions
- *Microsoft Operating System/2 Device Drivers*
  A reference for the MS OS/2 device drivers

Manual Organization

Each manual is organized so that you can easily find the information you need. The Table of Contents, for instance, can be found in two places: at the beginning of the manual and, in greater detail, at the beginning of each chapter.

Another place to look for information is in the index for each manual. The indexes are cross-referenced several different ways to help you find the information you're looking for as well as any information about related topics.

Suggested Reading Sequence

To be provided at a later date.

Notational Conventions

Throughout this manual, the following conventions are used to distinguish elements of text:

bold	Bold type is used for commands, options, switches, and literal portions of syntax that must appear exactly as shown.
italic	In addition to giving emphasis to new terms, italics are used for filenames, variables, and placeholders that represent the type of text to be entered by the user.
`monospace`	Monospace type is used for sample command lines, program code and examples, and sample sessions.
SMALL CAPS	Small caps are used for keys, key sequences, and acronyms.

Special Characters and Symbols

The MS OS/2 operating system requires that you use, and be familiar with, many special characters and symbols. For simplicity and consistency, the MS OS/2 manuals use the following set of names and symbols, some more frequently than others:

&	ampersand, *also* command separator
&&	AND operator
*	asterisk, *also* wildcard character
@	at symbol, used in code examples
\	backslash, path and filename separator

[]	brackets, enclose optional syntax items
^	caret, *or* escape character
$	dollar sign, used in device names
>	greater-than sign, *or* redirection symbol
<	less-than sign, *or* redirection symbol
!=	not-equal sign
#	number sign, used in code examples
||	OR operator
()	parenthesis, parentheses (plural), *also* command grouper
%	percent sign, used for replaceable parameters
|	pipe symbol, *or* pipe operator
±	plus/minus sign
?	question mark, *also* wildcard character
>>	redirection symbol, used to append output
/	slash, used in switches

Chapter 1
Introduction to Microsoft Operating System/2

1.1 Overview

Microsoft Operating System/2 (MS OS/2) is a new, advanced, single user, multitasking operating system for Intel 80286- and 80386-based microcomputers. MS OS/2 includes the following features:

- Memory management features allowing usage of the processor's architectural storage limit of 16 megabytes
- Memory support for system RAM larger than 640K
- Full multitasking support
- System services invoked by a CALL rather than INT 21H
- A structured means of "hooking" keyboard input for the purpose of examining individual keystrokes
- An extensive set of session management features
- Compatibility mode for applications
- Code sharing
- National Language Support

1.2 Real Mode and Protected Mode

The 80286 microprocessor can operate in two memory addressing modes: *real mode* and *protected virtual address mode* (*protected mode*). Both modes execute a superset of the Intel 8086 and 8088 microprocessors's instruction sets.

- In real mode, programs use real addresses with up to one megabyte of address space.
- In protected mode, programs use virtual addresses. The 80286 CPU maps one gigabyte of virtual addresses per task into a 16-megabyte real address space.
- Protected mode provides memory protection to isolate the operating system and to ensure privacy of each task's programs and data.

For existing MS-DOS 3.*x* applications, MS OS/2 provides a compatibility mode of operation, called the *3.x box*. The 3.*x* box (sometimes known as

the compatibility box) lets MS-DOS 3.*x* applications execute in real-mode and provides the familiar INT21H system interface in an MS-DOS 3.*x*-type environment. Programs executing in the compatibility box execute only in real mode.

1.3 MS OS/2 Function Requests

The MS OS/2 operating system uses a CALL-RETURN system request mechanism that is more extendable and more efficient than the INT 21H interface used by MS-DOS. This interface is known as the *Application Program Interface.* Programs using this interface are smaller in size (both on disk and in memory) and are not affected by changes in any MS OS/2 service routines that they use.

In addition, MS OS/2 supports a restricted subset of the application programming interface. Applications that are written using only this subset of the interface and that are linked with the appropiate MS OS/2 library routines are able to execute in either MS OS/2 or MS-DOS 3.*x* environments. The subset of MS OS/2 function requests is known as the MS OS/2 *Family Application Program Interface* (Family API).

1.4 Multitasking

The multitasking features of MS OS/2 let a user operate several applications at the same time. In most cases, each application appears to have the entire computer to itself and may be designed and coded in much the same manner as an MS-DOS 3.*x* application. Alternatively, an application may be designed so that its functions are divided among a collection of cooperating threads or processes.

Since multitasking is such an integral part of MS OS/2, a priority-based, time-slicing scheduler is provided with the product. This scheduler offers special consideration to applications with time-critical response requirements. Other functions provided with MS OS/2 allow

- starting new processes
- starting/stopping/modifying execution threads within a process

- coordinating execution among several processes

The simplest instance of multitasking occurs when running concurrent applications: each application's execution is managed under a *process* and at least one *thread* of execution. A *process* represents the execution of an application and the ownership of any resources associated with that execution, while a *thread* is the dispatchable element used by MS OS/2 to track execution cycles of the processor.

An application may be designed as several distinct processes, or as multiple threads within a single process. When you are designing an application and deciding your implementation strategy, consider the following:

- The creation and termination of a thread is very fast compared with the creation of a process, which is relatively slow.
- Sharing of data and resources between threads is natural, but sharing between processes takes special consideration.

Figure 1.1 shows the process structure when the user has started three applications: an editor, a database manager, and a communications handler. These are independent applications and they have no knowledge of one another. The fact that they may share a physical disk on which their individual disk files are located is only "known" and managed by MS OS/2.

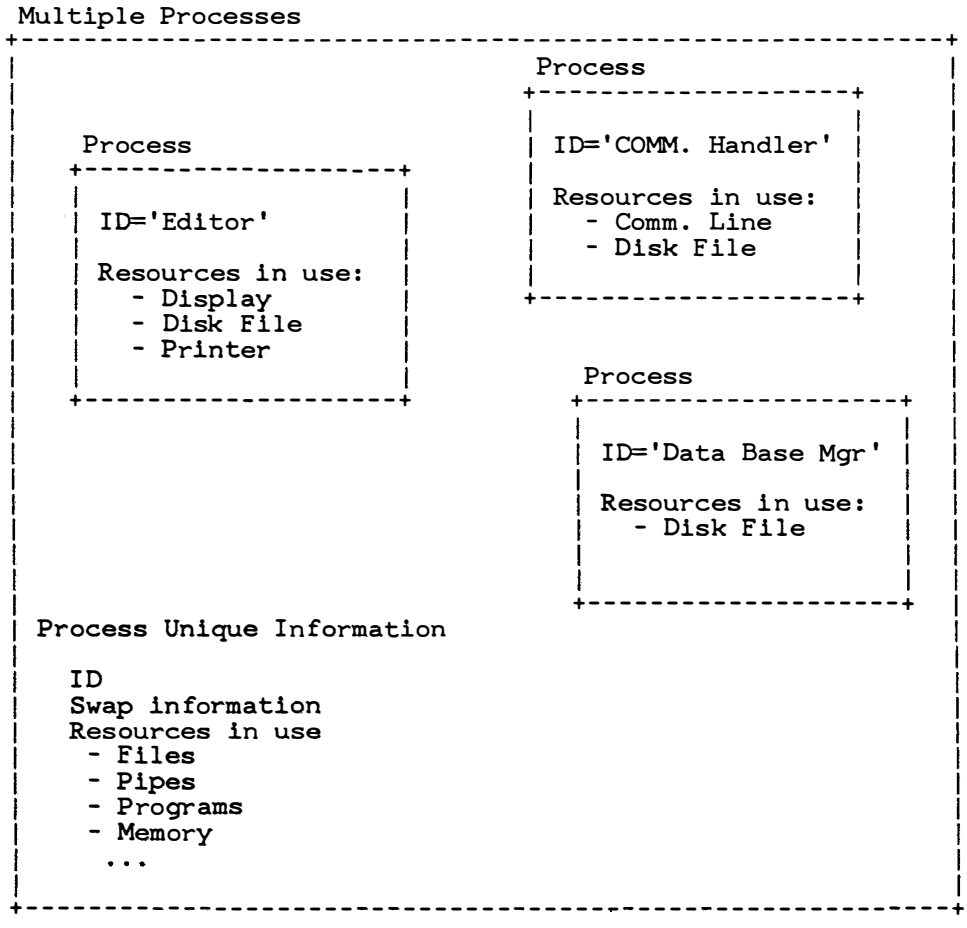


Figure 1.1 Multiple Independent Processes

For an application requiring multiple asynchronous execution threads, Figure 1.2 demonstrates how the various threads might operate, each using different devices to accomplish a portion of the overall task.

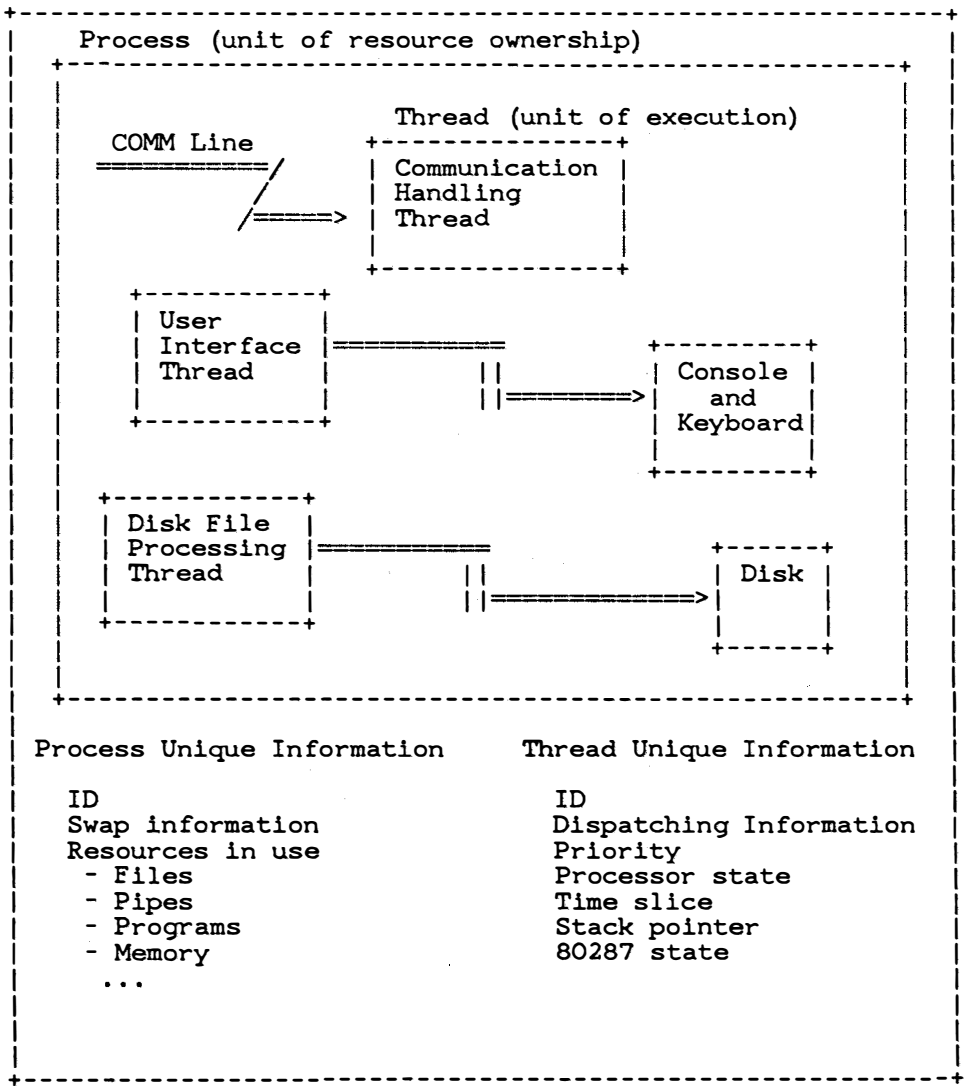


Figure 1.2 Multiple Threads within a Process

1.5 Interprocess Communication

MS OS/2 uses several means for interprocess communication:

- Pipes
- Queues
- Semaphores
- Shared memory
- Signals

Pipes, queues, and semaphores are created by the applications that use them; their handles or addresses are passed among the interested processes as necessary.

MS OS/2 extends the standard naming conventions of the file system to queues, system semaphores, and shared memory. This simplifies managing resources that are shared between programs and ensures that references to the same name are resolved to the same resource.

For example, system semaphores are defined within the file-system name space as "pseudo-files" instead of as a location in RAM. The semaphore is a "pseudo-file" because its name takes the form of a file in the subdirectory "SEM". This subdirectory does not actually exist in the physical file system; the semaphores and their names are actually kept in RAM. Like files, these resources use handles as means of referring to them within a program.

1.5.1 Pipes

Pipes establish file I/O communication between two programs — typically without the programs being aware that they are using a pipe rather than a file. Pipes can contain a maximum message text of 64K bytes whereas queues are not.

Figure 1.3 illustrates communication between two programs using a pipe.

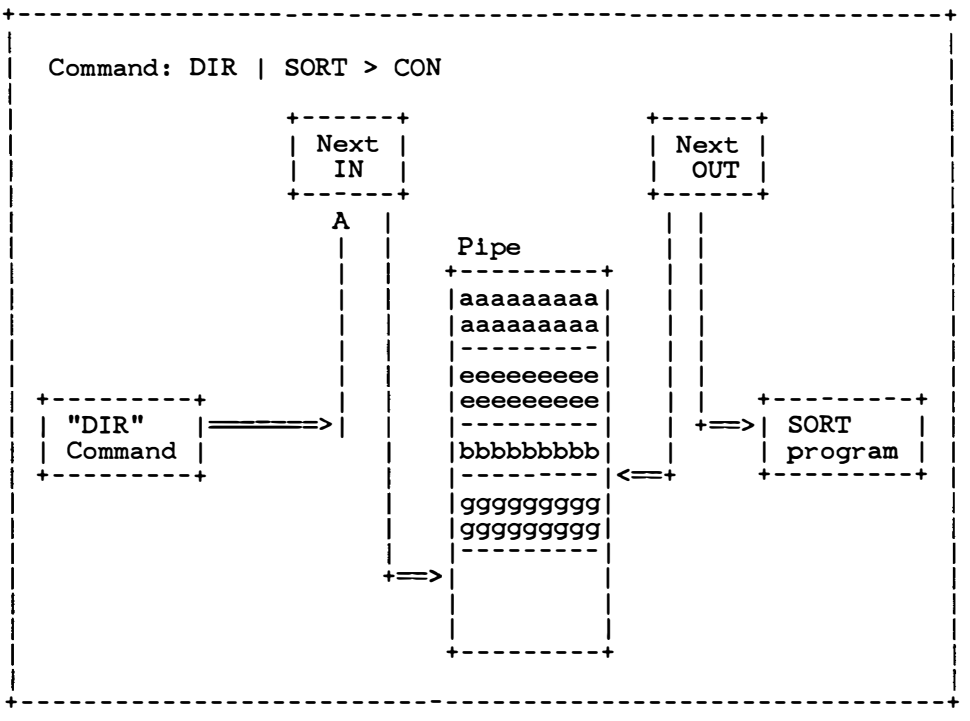


Figure 1.3 Connecting Two Programs Via a Pipe

Pipes might also serve as a fixed-length, First-In-First-Out, (FIFO) circular queue that provides communication between processes.

Figure 1.4 depicts the use of a pipe to pass data to a server process called Process X, from three child processes A, B, and C. The sending processes send data independently of one another; Process X removes data from the pipe at will.

MS OS/2 keeps track of the data and free space in the pipe with the Next IN and Next OUT pointers. When the pipe is full, the process that next writes to the pipe waits until Process X has removed enough data to make room for the new message.

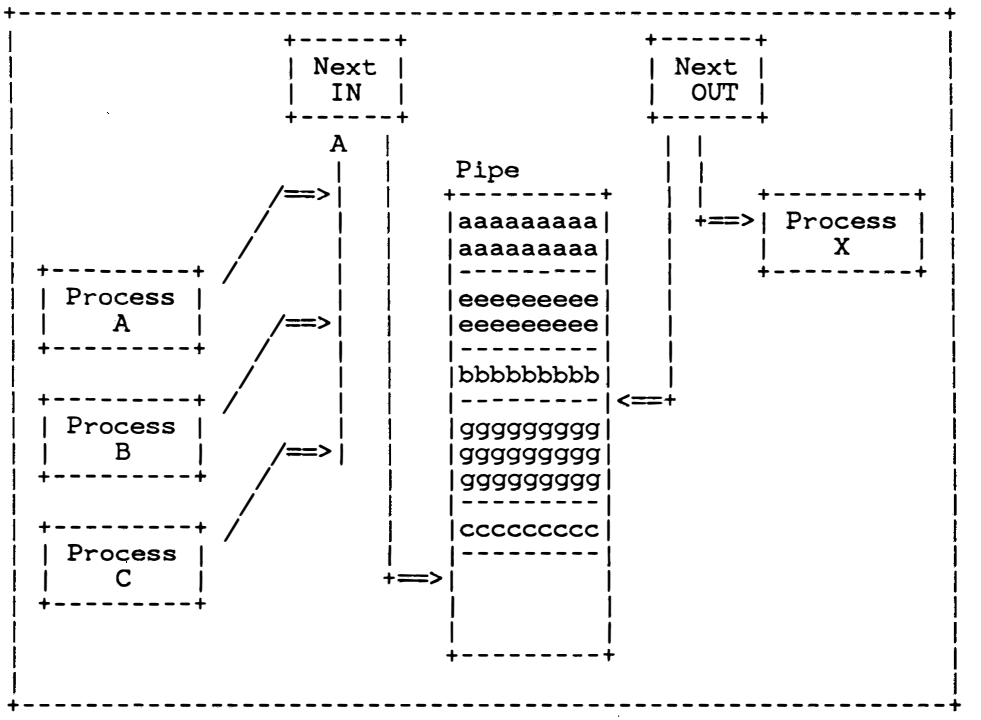


Figure 1.4 Communicating with a Pipe

1.5.2 Queues

For some applications, queues provide a more useful mechanism for interprocess communication of data between processes.

Figure 1.5 depicts the use of a queue that passes data to a server process called Process Y, from three child processes, D, E, and F. The sending processes may send data independently of one another as do pipes. However, in a process unique to queues, outstanding elements may be ordered by priority *or* by order of arrival—First-In-First-Out or Last-In-First-Out (LIFO); also, Process Y may examine the elements in the queue and remove them whenever, and in any order, desired.

Queues have several performance advantages over pipes:

- Data in the queue are not copied, but are passed instead in a shared segment.
- There is virtually no size limit for the messages themselves.

In contrast, pipes can contain a maximum message text of 64K bytes, whereas queues can contain large amounts of data, because

- Each message is a unique block of storage.
- The aggregate of all messages may be dispersed across the entire machine.

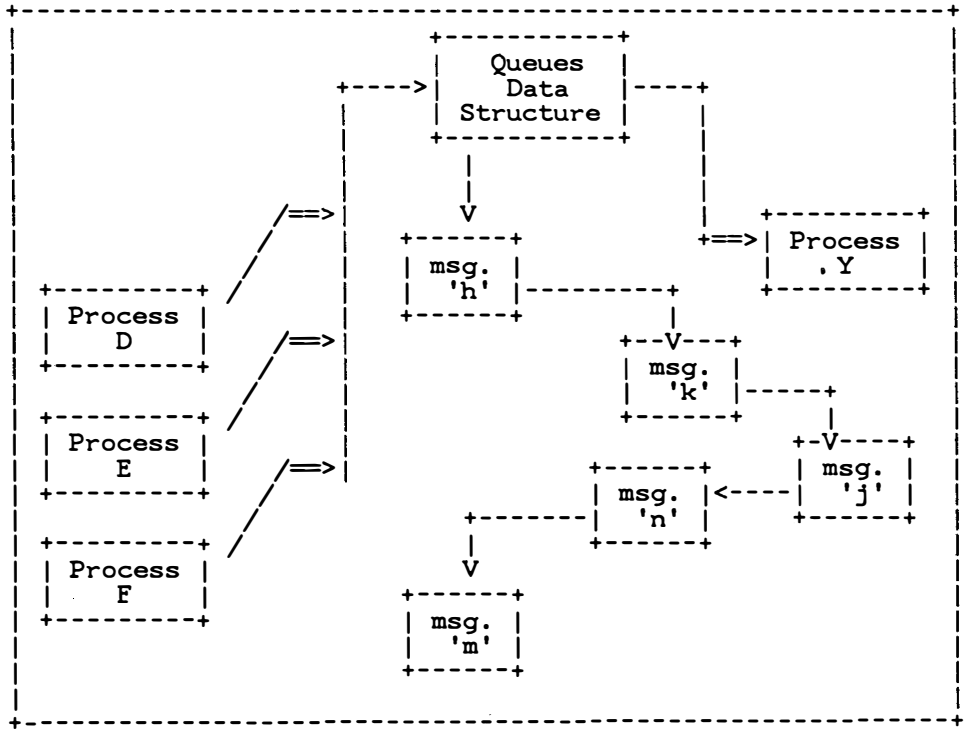


Figure 1.5 Communicating with a Queue

1.5.3 RAM Semaphores

The semaphore support provides serialization/signaling by means of RAM semaphores and system semaphores:

- RAM semaphores are a high-performance mechanism best used between the threads *within* a process. RAM semaphores are local to the process they serve.
- System semaphores are a full function mechanism suited for use *between* processes and can be considered a global resource.

Figure 1.6 depicts the use of semaphores for serializing access to a serially reusable resource. Only a single thread can enter the semaphore at a time. The effect is that all other threads are locked-out from use of the resource until the entering thread clears the semaphore.

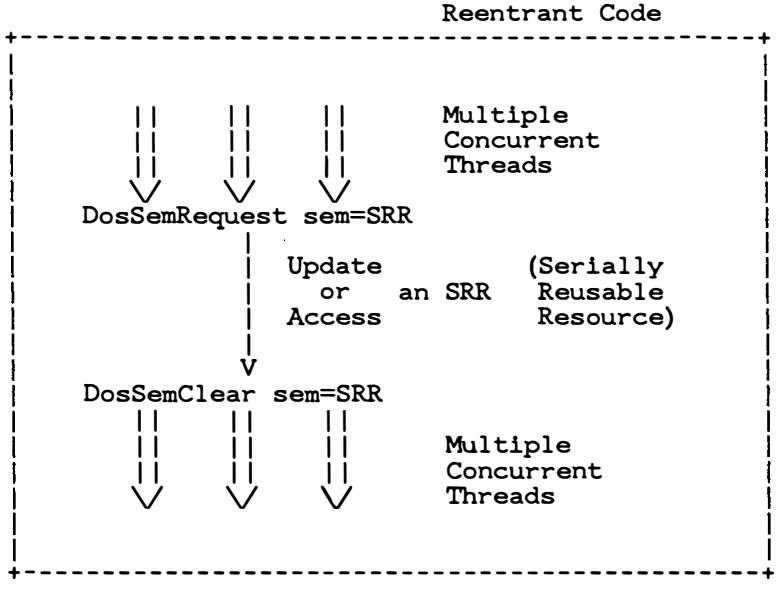


Figure 1.6 RAM Semaphores

1.5.4 Shared Memory

MS OS/2 provides several functions to allow processes to communicate via shared memory. The functions provided allow allocating, accessing, reallocating, and freeing a shared memory segment. When shared memory is requested via the memory allocation functions, the MS OS/2 memory protection features provide total isolation between processes using the shared memory.

1.5.5 Signals

Signals are a mechanism which allows a process to intercept and deal with a variety of external events. The signal facility allows a program to specify an on-condition handler routine which is executed when the event occurs. The following are examples of events which can cause signal handlers to be executed:

- CONTROL-C key pressed
- CONTROL-BREAK key pressed
- Process signalled via **DosFlagProcess**
- Program terminated via **DosKill**

The signals package provides functions that allow one process to set an external event flag to another process, and notify a process via one of three signals. The target process receives control at the signal handler that it defined for that signal.

1.6 Timer Services

MS OS/2 provides date and time functions similar to those provided with MS-DOS 3.*x*. In addition, MS OS/2 provides the following timer-related functions:

- Asynchronous intervals (the system notifies a task that a period of time has elapsed)
- Regularly occurring intervals (the system continuously notifies a task that a designated period of time has elapsed)

- Sleep for a period of time (a task delays its execution for a designated period of time)

Note

All time-related functions are based on a periodically interrupting time source. Since servicing a timer at a high interrupt frequency would require an inordinate amount of system overhead, the timer operates at a frequency of approximately 32 hertz. Note that while this rate is sufficient for the needs of most applications, it does not allow specifying a time interval with a precision of less than 50 milliseconds. All related discussions in this manual are subject to this limitation.

1.7 Memory Management

One of the most notable features of MS OS/2 is its segmented memory management, which takes advantage of the virtual segmentation hardware in the protected address mode of the 80286 processor. Segmented memory management allows applications to use memory beyond the 640K limit imposed by MS-DOS—up to the processor's 16-megabyte architectural storage limit. Applications written for MS-DOS are executable in the MS OS/2 environment. These applications execute in real-mode using the *compatibility mode*, which executes a single MS-DOS 3.*x* application in real mode.

For new applications, in addition to allowing the user to concurrently execute more programs than will fit in available memory, any single program and its data can be larger than real memory. MS OS/2 maintains the most active set of segments in storage at any time, discarding unneeded segments as required and reloading the application from disk when next accessed.

The new memory management functions allow a program to

- allocate a large number of data segments
- keep each segment private or to share it with other programs

- package an application so that the linkage to library routines or infrequently used routines is not made until runtime
- access an application as a number of distinct callable segments, with MS OS/2 loading segments on demand as necessary

In addition to these virtual memory management functions, MS OS/2 includes a high-performance mechanism for suballocating memory within a segment (small model).

Figure 1.7 depicts a series of memory suballocation requests that deplete the free storage within the segment.

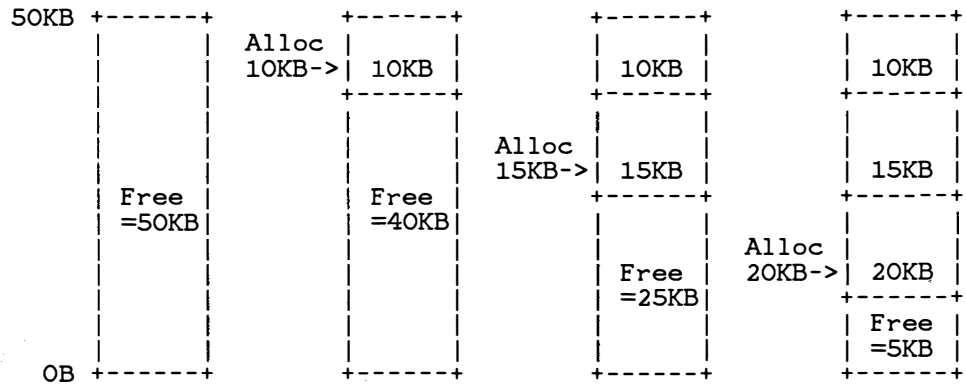


Figure 1.7 Memory Suballocation Request

The application may now begin freeing the storage as it is no longer needed. Adjacent free blocks will be combined as shown in Figure 1.8:

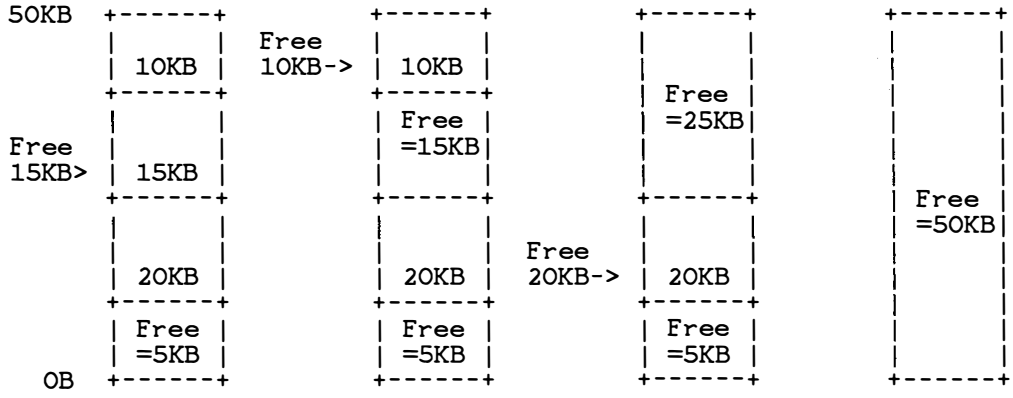


Figure 1.8 Memory after Suballocation

1.8 Dynamic Linking

The 80286 protected-mode CALL architecture also offers some benefits of greater importance than the hierarchical access structure to the various layers of the system and data copying between routines at different privilege levels. MS OS/2 provides a *dynamic-link* facility by which linkage to a called routine is not resolved until run time. The MS OS/2 API functions are implemented using a dynamic link library. Calls to these routines are not resolved until load time.

Dynamic linking offers some definite advantages over the typical static module structure used by MS-DOS.

- Common library routines need not be link-edited into each load module.
- Application programs need only have the most commonly used segments loaded when they are started. Exception processing routines may be left unloaded and be called (and be automatically loaded by the system) only when necessary.
- When a high-level language's library functions are accessed using dynamic linking, future updates to the library may be introduced into the using programs by replacing the library *without* forcing the user to obtain a new version of each program.

- Dynamic-link packages can be updated independently of the applications that use them. This means that an existing application need not be changed when the functions in its dynamic-link package are updated.

The actual programming steps required to use the dynamic-link feature are no different from those required for a static environment:

1. The programmer codes a call to a subroutine that is to be dynamically linked and declares it EXTERNAL FAR.
2. The compiler generates a standard external reference.
3. When the object module is linked, the linker is provided with the names of libraries that contain dynamic-link definition records. These records provide a correspondence between the called entry point and the module file containing the routine being called.

1.9 MS OS/2 Device Drivers

In the MS OS/2 operating system, all standard device drivers are enhanced so that they may service requests in real *or* protected mode. Where appropriate, device drivers provide a queued request interface rather than the serial request design used in MS-DOS 3.*x*.

MS OS/2 device drivers also support multiple synchronous and asynchronous I/O requests. For example, a disk device driver can queue several read and write requests from multiple requesters, servicing those requests in an order that minimizes the movement of the access mechanism across the disk.

Device drivers are divided into two parts—*strategy routines* and *interrupt routines*:

- A strategy routine is called with an I/O packet that describes the request. The strategy routine marks the request "incomplete" and then queues it. If the device is not busy, it starts the device and then returns to the kernel, which typically blocks on the incomplete I/O packet.
- An interrupt routine services the I/O completion, and if there is new work in the queue, services the device. It then indicates that the previous operation is complete and unblocks any threads waiting for this request to complete.

Device driver helper routines (**DevHlp** services) are provided for such tasks as managing the request queue, blocking and unblocking processes or threads, and locking and unlocking memory.

In MS OS/2, asynchronous I/O operations are performed by using a separate thread. When making a read or write request indicating asynchronous processing, the system creates a thread, which then performs the I/O request synchronously (to itself) while the requesting thread continues executing asynchronously. When the operation is complete, the I/O thread flags its completion to a RAM semaphore and terminates.

By accessing asynchronous I/O in this manner, it becomes transparent to a device driver whether an I/O request is synchronous or asynchronous.

Figure 1.9, "Device Driver—Synchronous I/O," depicts the flow of control for synchronous I/O.

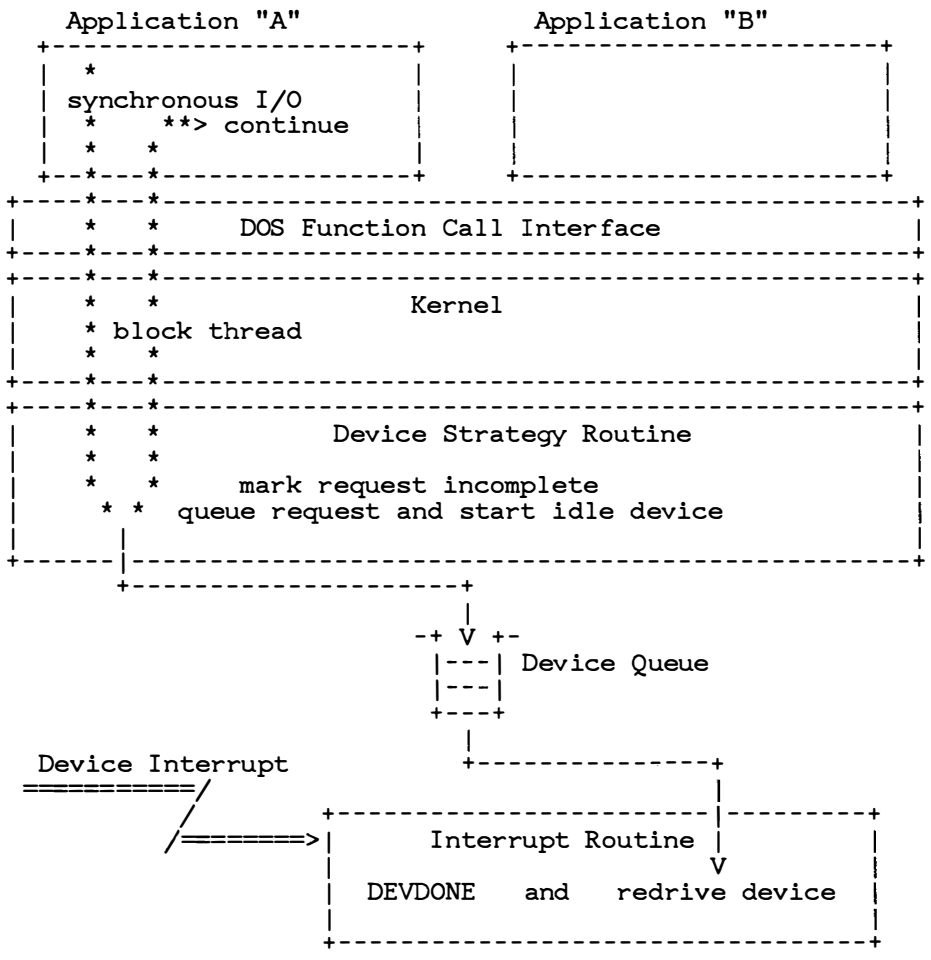


Figure 1.9 Device Driver-Synchronous I/O

The flow of control for synchronous I/O is described as follows:

1. The strategy routine marks the request incomplete and then queues it.
2. If the device is not busy, the strategy routine starts it.

3. Execution of the thread is blocked in the kernel until the interrupt routine indicates the request is done.

Figure 1.10 depicts the flow of control for asynchronous I/O.

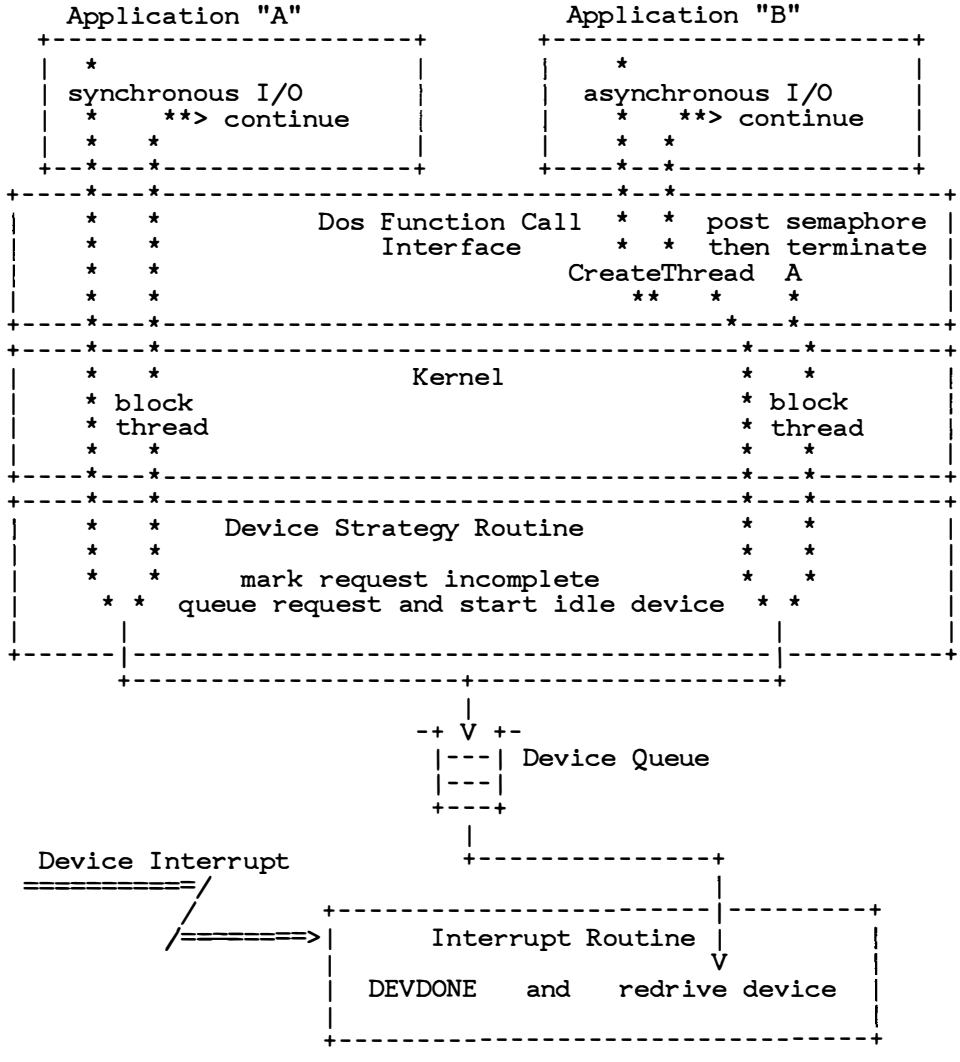


Figure 1.10 Device Driver–Synchronous and Asynchronous I/O

The flow is similar to the synchronous I/O shown in Figure 1.9, except that the requesting thread continues to execute while a separate I/O thread executes the I/O.

1.10 Utilities

Utilities are those commands that are not built into the two command line interpreters. (In MS-DOS, they're the *external* commands.) In addition to utilities similar to those from MS-DOS, MS OS/2 provides several utilities to support its increased capabilities. Many of the programs that accept filenames for arguments also accept multiple arguments. The following utility programs are included:

ansi	ANSI escape sequence support for the display
append	Append directories to PATH
assign	Assign drive letter to another drive
attrib	Display or modify file attributes
backup	Backup files
chkdsk	Check a disk for errors
diskcomp	Compare disk contents
diskcopy	Copy whole disk
edlin	Real-mode line editor
comp	Compare files
fdisk	Create or modify partition on a hard disk
find	Search for text in file(s)
format	Format a disk
graftabl	Enable extended character set in graphics mode
helpmsg	Provide help information related or a warning or error message

join	Join a disk drive to a specific directory
keyb[*yy*]	National Language keyboard support
label	Create or change volume label of disk
mode	Set operation modes for devices
more	Dispense information to the display one screen at a time
patch	Apply patches to a file
print	Print spooler
replace	Selectively replace all instances of a file
restore	Restore files from backup
sort	Sort data
spool	Start print spooler
subst	Substitute a string alias for a pathname
sys	Transfer MS OS/2 system files to a disk
tree	Display directory tree
xcopy	Copy directory trees

Programs for preparing applications:

link	Link application code and library routines
bind	Merge MS OS/2 *.exe* file and resolve dynamic links for execution in MS-DOS 3.*x* environment
mkmsgf	Create message file for MS OS/2 message facilities
msgbind	Modifies an MS OS/2 EXE file to contain RAM based messages.
implib	Creates an import library from a dynamic-link module for use by other application developers.

1.11 Print Spooling

The printer spooler (**spool**) is a true spooler serving one printer device at a time. Spool data is sorted by process number so that the printer output from separate processes is not intermixed. *Print streams* that are not closed (in the process of being updated) are placed in temporary files on the disk in the subdirectory specified on the **spool** command line.

Since printers may be attached to the computer by both parallel and serial connections, the printer spooler supports printer devices (LPT1, LPT2, LPT3, PRN) and serial devices (COM1, COM2, AUX).

The printer spooler also supports any other character device whose device driver contains monitor support compatible with the printer device driver.

1.12 General System Requirements

As a *minimum* configuration, the user's MS OS/2 system requires the following hardware:

- IBM PC-AT/compatible system unit with real-time clock
- One megabyte of memory
- Two disk drives (1.2 megabytes each) or one disk drive with a hard disk drive
- Keyboard
- Display (monochrome, CGA, EGA, or compatible)

The MS OS/2 software development environment consists of the following hardware and software:

- To be provided at a later date

1.13 Memory Requirements

The diagram in Figure 1.11 defines the fixed and transient storage requirements of the MS OS/2 system with and without compatibility mode in the system.

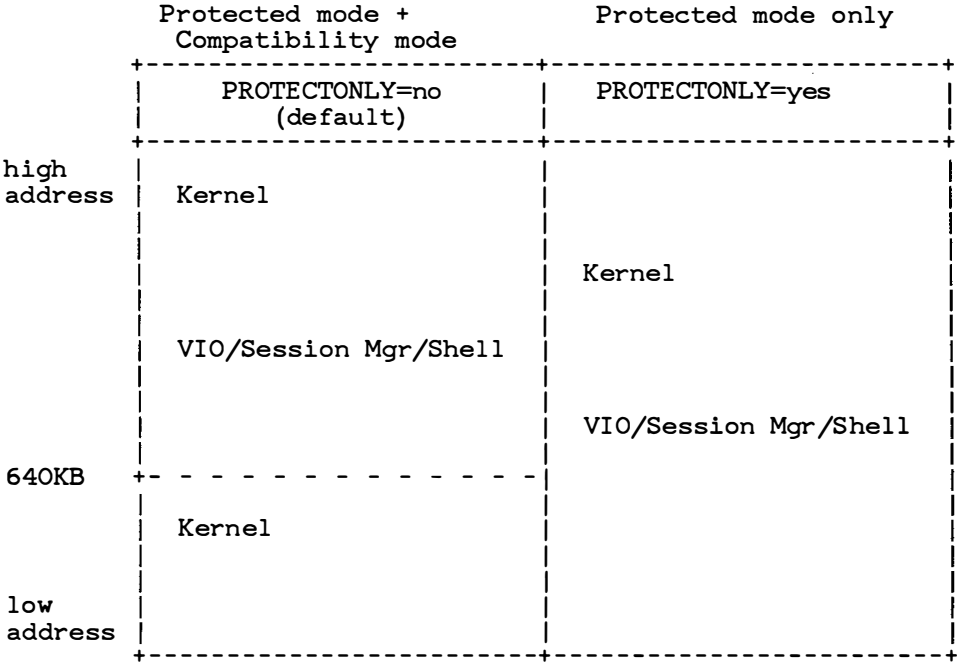
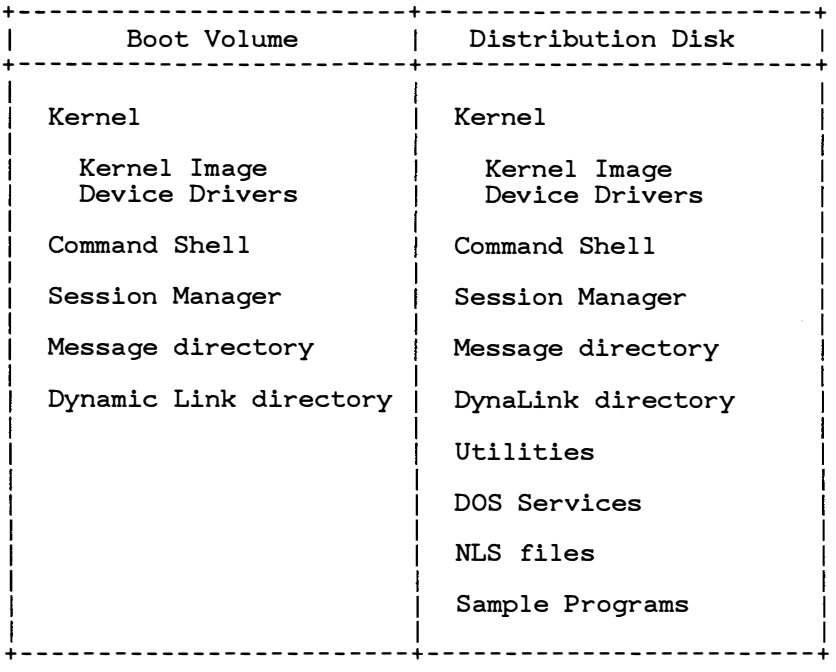


Figure 1.11 MS OS/2-DOS 3.*x* Memory Maps

1.14 Disk Requirements

Figure 1.12 lists the minimum system disk requirements for MS OS/2.



1.15 MS OS/2 System Initialization

The purpose of system initialization is to establish the environment in which MS OS/2 executes. This environment, in turn, has options that you may choose to configure. For more information about these options see the *Microsoft Operating System/2 Setup Guide.*

System initialization occurs as a result of turning on or resetting the computer. The following modules initialize the system:

- The boot sector
- The MS OS/2 device interface module
- The MS OS/2 kernel module
- The MS OS/2 system initialization process

1.15.1 The Boot Sector

On floppy disks, the boot sector begins on track 0, sector 1, side 0. On hard disks, the boot sector begins on the first sector of the MS OS/2 partition.

When you reset your computer or turn on its power, the processor is in real mode, the ROM BIOS is invoked and performs hardware checks and initialization. The ROM BIOS then examines drive A for the boot sector. If it locates a boot sector, ROM BIOS reads it into low memory and gives it control. If it doesn't find the boot sector, ROM BIOS then looks in the active partition of the fixed disk. If it still doesn't find the boot sector, then ROM BIOS invokes ROM BASIC.

When the code in the boot sector gains control, the processor is still operating in real mode.

The boot code loads the MS OS/2 device interface module into low memory and gives it control.

1.15.2 The MS OS/2 Device Interface Module

When called, the MS OS/2 device interface module invokes its initialization routine. This routine performs an equipment check and loads the MS OS/2 kernel module, which it then invokes.

1.15.3 The MS OS/2 Kernel Module

When the device interface module calls the MS OS/2 kernel module, the kernel module invokes its initialization code, which relocates the code for the system initialization process into high conventional memory (below the 640K line). It also relocates kernel code and data segments to the appropriate places in memory and initializes the kernel components and the default device drivers. The system IDLE and system initialization processes then initialize the scheduler. Finally, the kernel module switches the processor to protected mode and invokes the system initialization process.

1.15.4 The MS OS/2 System Initialization Process

The system initialization process handles the configuration commands in the *config.sys* file, establishing the final operating environment.

1.15.5 The System Configuration File (*config.sys*)

The *config.sys* file contains commands used to configure the system. Only one *config.sys* file is needed to configure the system for both real-mode and protected-mode operations.

During system initialization, MS OS/2 opens and reads the *config.sys* file in the root directory of the drive from which it was started, and interprets the commands within the file. If it doesn't find the file, MS OS/2 assigns default values for the configuration commands.

The following list summarizes the configuration commands for MS OS/2:

buffers	Determines the number of buffers to allocate for disk I/O
country	Selects the format for country-dependent information
device	Specifies the pathname of a device driver to be installed
iopl	Specifies whether I/O privilege is to be granted
libpath	Specifies the location of dynamic-link modules.
maxwait	Sets the time limit for calculating CPU starvation
memman	Selects memory management options
priority	Disables dynamic priority variation in scheduling regular class processes
protectonly	Selects the modes of operation
protshell	Specifies the pathname of the MS OS/2 top-level command processor
run	Starts a system or daemon process during system initialization
swappath	Specifies the location of the swap file

timeslice	Sets the time-slice values for process scheduling
threads	Sets the maximum number of threads in the system
trace	Select the system trace
tracebuf	Sets the trace buffer size

The following two commands are ignored.

files

lastdrive

The following commands apply only to the configuration for real-mode and only if the **protectonly** command specifies that old MS-DOS 3.x applications will be run. These real-mode-only commands are documented following the MS OS/2 configuration commands.

break	Checks for CONTROL-BREAK
fcbs	Determines information about file control block management
rmsize	Selects the amount of memory for real-mode applications
shell	Loads and starts the top-level command processor

Note

With the exception of the **break** command, the commands listed above are usable only in the *config.sys* file. They are not useable in the *autoexec.bat* file or from the MS OS/2 command line.

1.16 The MS OS/2 Command Processors

Two command processors are supplied with MS OS/2:

- *command.com* — the real-mode command processor
- *cmd.exe* — the protected-mode command processor

1.16.1 *Command.com*

Command.com is the real-mode command processor. It consists of three parts:

- A *resident* part resides in memory immediately following *msdos.sys* and its data area. The resident part contains routines to process Interrupt 23H (CONTROL-C Exit Address).

- An *initialization* part follows the resident part. During startup, the initialization part is given control; it contains the processor setup routine in the *autoexec.bat* file. The initialization part determines the segment address at which programs can be loaded, and then, because it is no longer needed, it is overlaid by the first program that *command.com* loads.

- A *transient* part is loaded at the high end of memory. This part contains all internal command processors and the batch file processor.

  The transient part of the command processor produces the system prompt (A>, for example). It also reads commands from the keyboard (or batch file) and causes them to be executed. For external commands, the transient part builds a command line and issues the **Exec** system call (Function Request 4B00H) to load and transfer control to the program.

1.16.2 *Cmd.exe*

Cmd.exe is the protected-mode command processor. It includes an extended command syntax, which is described in another section of this manual.

Cmd.exe executes command lines under MS OS/2 the same way that *command.com* executes them under MS-DOS 3.*x*, with the exception of multiple command statements, which can be executed through the use of the AND, OR, and Command Separator operators. Generally, execution of individual commands remains the same, but grouping and sequencing can be conditionally altered through the use of the operators. All commands return error codes that can be tested and acted on.

MS OS/2 supports the internal commands of MS-DOS 3.*x* essentially unchanged, although the following commands now take multiple arguments: **del**, **dir**, **type**, **vol**, **mkdir**, and **rmdir**. With few exceptions, external commands are processed in the same manner as in MS-DOS 3.*x*. However, external commands, batch processing, and the *autoexec.bat* file are constrained to prevent the actions of one program from affecting other programs. Environment variables are replicated on a per-program basis to let applications run independently.

Chapter 2
MS OS/2 Application Program Interface (API)

2.1 Introduction

This chapter describes how MS OS/2 system calls are issued, and describes the general methods used in defining the programming interface. This interface is easily expandable and allows high-level languages easy access to MS OS/2. Finally, this chapter describes *compatibility mode*, which is used for executing MS-DOS 3.*x* programs.

2.2 MS OS/2 Function Call Interface

MS OS/2 uses a CALL-RETURN interface for service requests to the operating system, passing the request parameters on the stack. Old-style (INT 21H) system calls are supported only for real mode. The benefits from using the CALL-RETURN interface are:

- Less need for a high-level language's system services library — the appropriate MS OS/2 system call may be accessed directly from a high-level language such as C.
- Optimum performance — the target routine can be invoked directly rather than through an intermediary "router".
- The same interface mechanism is available for invoking both an MS OS/2 routine and a high-level language's library routine.
- Replacing high-level language functions with MS OS/2 calls follows the MS OS/2 architecture and is a well-defined activity rather than an improvised mechanism.
- The CALL interface provides significant performance gains if the parameters are pushed onto the stack before issuing the CALL in protected mode. The 80286 or 80386 hardware copies the parameters from the requester's stack to the receiving program's stack. This copying establishes optimum addressability and protection at minimal execution cost.

  MS OS/2 also provides the means to call system extensions and I/O privilege routines executing at protection ring 2 of the processor. As shown in Figure 2.1 the application interface to protected-mode MS OS/2 and device drivers is strictly hierarchical. The 80286 hardware supports and enforces this hierarchy.

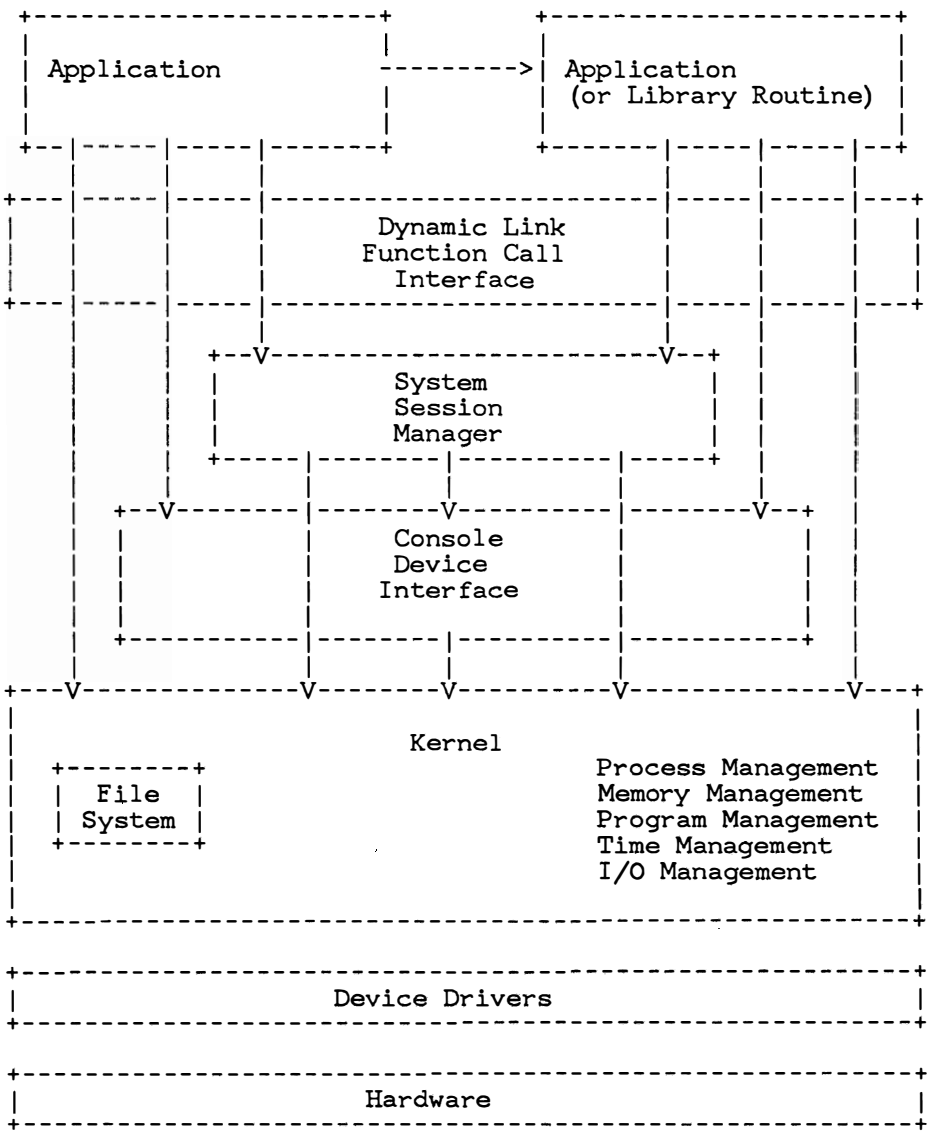


Figure 2.1 Overall System Structure

2.3 Family Programming Model

The full function MS OS/2 API contains a subset of functions available on MS-DOS 3.2. This subset of the full MS OS/2 API is known as the Family API. An application that uses only the Family API functions can be constructed to run in protected mode under MS OS/2, or in real mode in an MS-DOS 3.2 system.

Language bindings are a set of modules that provide the interface to the operating system. These modules are loaded when they are executed in MS-DOS 3.2, but are ignored when loading in the MS OS/2 environment.

The family programming interface takes into consideration that MS OS/2 requests must operate for all memory models in a wide range of supported languages. In addition, these function requests must all be dynamic-link entries.

2.3.1 Interface Rules

The following rules apply to the application program interface. These rules must be followed when writing an MS OS/2 application.

Rule 1: All parameters are passed on the stack (SS:SP).

The practice of passing parameters on the stack is common to a broad base of languages on the 8086 family of processors. This practice allows direct access to the operating system from high-level languages.

Rule 2: All interfaces pass a return code (in AX) back to the caller.

Many languages also use the AX register for a function return code. All other user registers, except the FLAGS register, are preserved. The contents of the FLAGS register are undefined. The state of the direction flag in the FLAGS register is either preserved or cleared. If the direction flag was clear when an MS OS/2 function was called, then it will be cleared when the function returns.

Rule 3: All addresses of output parameters are of the following form:

selector:offset

Fully qualified addresses are available across all memory models as a method for returning values to a requester. This method allows one function entry point to serve all languages and memory models.

Rule 4: All functions are accessed by far calls.

This is a requirement for the functions to be dynamic-link entries.

Rule 5: All functions remove the parameters from the stack.

Parameter lists are fixed-length on a function basis. That is, variable-length parameter lists are not supported.

Rule 6: All functions must be uppercase at link time. If a compiler or assembler generates case-sensitive (upper- and lower-case) external references, all calls or function definitions must be in all UPPER-case characters.

2.4 Format and Characteristics of MS OS/2 Function Request

The function requests described in the *Microsoft Operating System/2 Programmer's Reference* and in this manual follow a pseudo-assembly-language format. Therefore, the application interfaces shown in this manual are descriptive "representations" rather than examples of real coding sequences. The conventions for these representations are described in this section, and are used throughout the MS OS/2 manual set.

2.4.1 Function Request Format

Pseudo-instructions

The following pseudo-instructions describe the operations that push parameters onto the stack:

Instruction	Description
PUSH	Push an item onto the stack. This instruction pushes items of various sizes onto the stack. The types of data items are described in the following section, "Data Items."
PUSH@	Push the address of an item onto the stack. Each address in these interfaces is composed of a 32-bit value: a 16-bit selector, and a 16-bit offset. Each address can point to any type of data item.
CALL	Call a function. All functions are accessed via far calls. This is a requirement for use of the dynamic-link mechanism.

Data Items

The following are types of data items used in a parameter list:

Data type	Description
WORD	2 bytes This type of operand can be passed by value (pushed onto the stack) or by reference (the address of the operand is passed on the stack).
DWORD	4 bytes This type of operand can be passed by value (pushed onto the stack) or by reference (the address of the operand is passed on the stack).

ASCIIZ — Null-terminated character string

This type of operand can be accessed only by reference.

OTHER — Any other structure

This type of operand can be accessed only by reference.

2.4.2 Stack Frame

The bulk of the interface is the stack frame. All parameters are passed via the stack. Figure 2.2 depicts a sample stack frame:

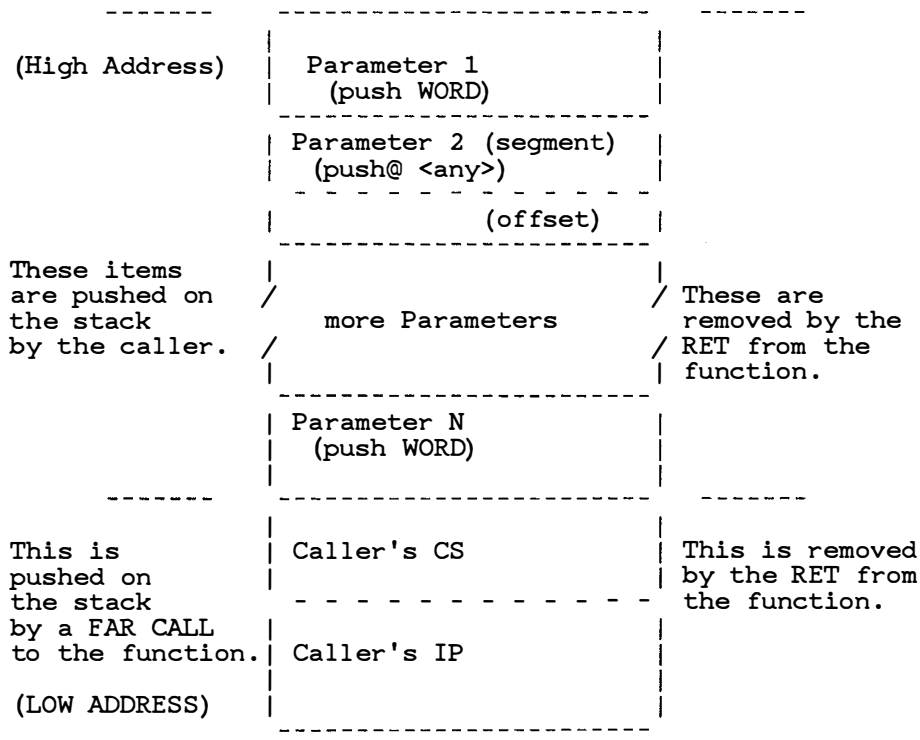


Figure 2.2 Interface Stack Frame

2.4.3 Function Calling Sequence Example

Figure 2.3 shows a sample function request, although some definitions and necessary statements for segments and procedures have been left out. The code used is similar to that of Macro Assembler.

Specification	Actual code
extrn DOSXAMPL:far	extrn DOSXAMPL:far
	;
	; these items are in DS
	;
	IN1 dw 44
	IN2 db 'x'
	OUT1 dw 0
	.
	.
	.
PUSH WORD **IN1**	; push IN1
PUSH@ WORD **OUT1**	; push ds ; mov ax,offset OUT1 ; push ax
PUSH WORD **IN2**	; mov al,IN2 ; xor ah,ah ; push ax
CALL DOSXAMPL	; call DOSXAMPL

Figure 2.3 Function Call Example

2.5 Example of an MS OS/2 Function Implementation

Figure 2.4 shows a sample function—a nonsensical one, but one that does show entry and exit sequences for a function, and demonstrates how to access the parameters. Some definitions and necessary statements for segments and procedures have been left out of the example. The code used in the example is similar to that of Macro Assembler.

```
push bp
mov bp,sp
.
.
.
les bx,[bp+8]           ; get the @ of OUT1

mov word ptr es:[bx],77; put 77 in OUT1

mov ax,[bp+12]          ; put IN1 in ax

mov bl,[bp+6]           ; put IN2 in bl
.
.
.
mov ax,17               ; set the return code to 17

pop bp
ret (far) 8             ; remove parameters from stack
                        ; and return
```

Figure 2.4 Function DOSXAMPL

Note

The ENTER and LEAVE instructions could also be used if the application only needs to run under MS OS/2.

2.6 High-Level Language Interface Examples

The following examples illustrate high-level language interfaces for the examples just shown in Figures 2.3 and 2.4.

2.6.1 C Example

The example shown in Figure 2.5 illustrates the interface to an MS OS/2 function from a C program. The C compiler used is the compiler used for MS OS/2 development.

Declaration:
```
int far pascal DOSXAMPL();
```
Data Declaration:
```
int out1;
char xin2;
```
Invocation:
```
rc=DOSXAMPL(44,(charfar*)&out1,xin2);
```

Figure 2.5 C Example

2.6.2 Pascal Example

Figure 2.6 is an example of an MS OS/2 function within a Pascal program.

Declaration:
```
function DOSXAMPL( p_in1  :integer;
                   vars p_out1 :integer;
                   p_in2  :char
                   ): integer;
```
Data Declaration:
```
var p_in1,
    p_out1 :integer;
    p_in2  :char;
```
Invocation:
```
rc := DOSXAMPL(44,out1,xin2);
```

Figure 2.6 Pascal Example

2.7 Family API Calls

The family API calls are the functions provided with MS OS/2 that let an application be linked to run in both MS OS/2 and the DOS 3.*x* box.

The interfaces provided are a subset of the full MS OS/2 API.

The routines marked with an asterisk (*) in the Subset API Function column are supported, but with restrictions. "Family API Subset Restrictions."

2.8 Family API Calls

The family API calls are the functions provided with MS OS/2 that let an application be linked to run in both MS OS/2 and the DOS 3.*x* box.

The interfaces provided are a subset of the full MS OS/2 API.

The routines marked with an asterisk (*) in the Subset API Function column are supported, but with restrictions.

Tasking

Subset API Function	MS OS/2 only
	DosCreateThread
* DosCWait	
	DosEnterCritSec
* DosExecPgm	
* DosExit	
	DosExitCritSec

	DosExitList
	DosGetInfoSeg
	DosGetPrty
	DosKillProcess
	DosSetPrty
	DosPTrace
	DosGetPid

Asynchronous Notification

Subset API Function	MS OS/2 only
* DosHoldSignal	
	DosSendSignal
* DosSetSigHandler	

Interprocess Communication

Subset API Function	MS OS/2 only
	DosFlagProcess
	DosMakePipe
	DosCloseQueue
	DosCreateQueue
	DosOpenQueue
	DosPeekQueue
	DosPurgeQueue
	DosQueryQueue
	DosReadQueue
	DosWriteQueue
	DosSemClear

	DosSemRequest
	DosSemSet
	DosSemSetWait
	DosSemWait
	DosMuxSemWait
	DosCloseSem
	DosCreateSem
	DosOpenSem
	DosResumeThread
	DosSuspendThread

Timer

Subset API Function	MS OS/2 only
DosGetDateTime	
DosSetDateTime	
DosSleep	
	DosTimerAsync
	DosTimerStart
	DosTimerStop

Memory Management

Subset API Function	MS OS/2 only
* DosAllocSeg	
	DosAllocShrSeg
	DosGetShrSeg
	DosGiveSeg
* DosReAllocSeg	

DosFreeSeg
* DosReAllocHuge
DosGetHugeShift
* DosAllocHuge
DosCreateCSAlias
DosSubAlloc
DosSubFree
DosSubSet

Dynamic Linking

Subset API Function	MS OS/2 only
	DosFreeModule
	DosLoadModule
	DosGetProcAddr
	DosGetModHandle
	DosGetModName

Family API

Subset API Function	MS OS/2 only
DosGetMachineMode	
BadDynLink	

Device I/O Services

Subset API Function	MS OS/2 only
DosBeep	
DosDevConfig	

* DosDevIOCtl

DosIOAccess
DosSgNum
DosSgSwitch
DosSgSwitchMe

KbdRegister
KbdCharIn
KbdFlushBuffer
* KbdPeek
KbdSetStatus
KbdGetStatus
KbdStringIn

VioRegister

VioGetBuf
VioGetCurPos
VioGetCurType
VioGetPhysBuf
VioReadCellStr
VioReadCharStr
VioScrollDn
VioScrollUp
VioScrollLf
VioScrollRt
VioSetCurPos
VioSetCurType
VioSetMode
VioShowBuf

VioWrtCellStr
VioWrtCharStr
VioWrtCharStrAtt
VioWrtNAttr
VioWrtNCell
VioWrtNChar
VioWrtTty

VioSetAnsi
VioGetAnsi
VioPrtScreen
VioSavReDrawWait

* VioScrLock
VioScrUnLock

VioSetMnLockTime
VioSetMxSavetime
VioGetTimes
VioPopUp
VioEndPopUp
MouRegister
MouGetNumButtons
MouGetNumMickeys
MouGetDevStatus
MouReadEventQue
MouGetNumQueel
MouGetEventMask
MouGetScaleFact
MouSetScaleFact
MouSetEventMask

MouOpen
MouClose
MouSetPtrShape
MouRemovePtr
MouDrawPtr
MouSetHotKey

Device Monitors

Subset API Function	MS OS/2 only
	DosMonClose
	DosMonOpen
	DosMonRead
	DosMonReg
	DosMonWrite

File I/O

Subset API Function	MS OS/2 only
	DosBufReset
DosChdir	
DosChgFilePtr	
DosClose	
DosDelete	
DosDupHandle	
* DosFileLocks	
* DosFindClose	
* DosFindFirst	
* DosFindNext	

DosMkdir
DosMove
DosNewSize
* DosOpen
DosQCurDir
DosQCurDisk
DosQFHandState
* DosQFileInfo
DosQFileMode
DosQFSInfo
DosQHandType
DosQVerify
DosRead
DosReadAsync
DosRmdir
DosSelectDisk
* DosSetFHandState
DosSetFileInfo
DosSetFileMode
DosSetMaxFH
DosSetVerify
DosWrite
DosWriteAsync

Errors and Exceptions

Subset API Function	MS OS/2 only
	DosSystemService
* DosError	

DosSetVec

Messages

Subset API Function	MS OS/2 only
DosGetMessage	
DosInsMessage	
DosPutMessage	

Program Startup

Subset API Function	MS OS/2 only
DosGetEnv	
DosGetVersion	

National Language Support

Subset API Function	MS OS/2 only
* DosGetCtryInfo	
* DosSetCtryCode	
* DosGetDBCSev	
* DosCaseMap	
* DosGetSpecChar	

2.8.1 Family API Subset Restrictions

The following family API subset functions are supported with restrictions in the MS-DOS 3.*x* environment:

DosCWait: Wait for Process

The following options operate differently in a 3.*x* environment than in an MS OS/2 protected-mode environment.

- *ActionCode*=1

  The MS-DOS 3.*x* program will wait until the indicated process has ended.

- *WaitOption*=1

  The **DosCWait** call returns an error: "no child process exists."

- *ProcessIDWord*

  This field is 0, but is not related to the PID of the terminating child process.

- *ProcessID*<>0

  The current MS-DOS 3.*x* program will wait until the indicated process has ended.

If this call is ever issued prior to the **DosExecPgm** of a child process, the current process will wait forever since no child process will ever start asynchronously.

DosExecPgm: Start Program (Process)

- *AsyncIndic*=1

  DosExecPgm returns an error: "could not exec."

- *AsyncIndic*=2

  DosExecPgm returns an error: "could not exec."

- *TraceIndic*=1

  DosExecPgm returns an error: "could not exec."

- *ProcessIDWord*

  This field will be 0. This number will not be related to the PID of the program being executed.

DosExit: Exit Thread

Since there are no threads in the MS-DOS 3.*x* environment, this function exits the currently executing program.

- *ActionCode*=0

  This option is ignored in that this function is equivalent to an *ActionCode*=1 call.

DosHoldSignal: Hold Signal

The only signals recognized in the MS-DOS 3.*x* environment are SIGINTR, SIGTERM, and SIGHDERR. Only SIGINTR will be turned off by this call.

DosSetSigHandler: Set Signal Handler

As mentioned in **DosHoldSignal**, SIGINTR is the only signal handled in this environment.

- *Action*=3

  This option generates an error: "invalid signal number."

- *Action*=4

  This option generates an error: "invalid signal number."

- *SigNumber* = anything but SIGINTR or SIGTERM

  These options generate an error: "invalid signal number."

DosAllocSeg: Allocate Segment of Storage

- *Size*

  The value requested will be rounded up to the next paragraph.

- *Selector*

  The selector will be the actual segment address allocated.

DosReAllocSeg: Reallocate Storage

- *Size*

  The value requested will be rounded up to the next paragraph.

DosReAllocHuge: Reallocate Huge Segment

- *Size*

  The value requested will be rounded up to the next paragraph.

DosAllocHuge: Allocate Huge Segment

- *Size*

  The value requested will be rounded up to the next paragraph.

- *Selector*

  The selector will be the actual segment address allocated.

DosDevIOCtl: Device IOCtl Call

These functions in the MS-DOS 2.*x* environment will be more restrictive than in the MS-DOS 3.*x* environment.

- Category 1
- Category 8

KbdPeek: Peek at Next Keystroke

The *CharData* structure includes everything except the time stamp.

VioScrLock: Screen Lock

The *Status* will always indicate that the lock was successful.

DosFileLocks: File Locks

- *Block*=1

  If this option is specified, one of two errors is returned: "invalid range lock list" or "invalid unlock list," whichever is appropriate.

- *NewLockIDList*

  This option is unsupported.

DosFindClose: Find Close

If this function is issued, the next **DosFindNext** will fail if an intervening **DosFindFirst** has not been issued.

DosFindFirst: Find First File

- *DirHandle*

  This parameter must always be 1.

DosFindNext: Find Next File

- *DirHandle*

  This parameter must always be 1.

DosOpen: Open File

- *OpenMode*

  The **W** flag (Write-through) is not supported.

- *AccessMode*

  This parameter is valid only if *SHARE* is loaded.

DosSetFHandState: Query File Handle State

- *FileHandle*

  The validity of the handle is not checked.

  This parameter must be 0.

- *WriteThroughFlag*

  This parameter must be 0.

- *Fail-ErrorsFlag*

  This parameter must be 0.

DosError: Set Error Flag

- *Flags*

  A value of 0000 will cause all subsequent INT 24s to be "FAILED," until a subsequent call returns with a value of 1.

DosGetCtryInfo: Get Country Information

Not all of the country information is available in MS-DOS 2.*x* and MS-DOS 3.*x*.

DosQFileInfo: Query File Information

This call adversely affects **DosOpen** performance.

BadDynLink: Bad Dynamic Link

BadDynLink is a call generated internally by the **286BIND** utility. It resolves dynamic-link calls that are unresolved in MS-DOS 2.*x* or MS-DOS 3.*x* but are valid in the MS OS/2 protected environment. This process allows applications to have a set of functions available in the MS OS/2 protected environment that are not available in MS-DOS 2.*x* or MS-DOS 3.*x*. The **DosGetMachineMode** call can be used in the application to determine at run time whether a call is appropriate.

If a routine is called in MS-DOS 2.*x* or MS-DOS 3.*x* that is not valid for real mode, the **BadDynLink** call issues an error message and aborts the application.

Chapter 3
The Protected-mode Command Interpreter: Cmd.exe

3.1 Protected-mode Command Interpreter

Cmd.exe is the protected-mode command processor. It includes an extended command syntax, which is described in this section.

3.2 Command Language Syntax

The syntax of the command language used by *cmd.exe* is described below.

A statement is made up of a sequence of commands. A command may be an internal command (one that is built into the command processor) or an external command (an application program).

```
statement = s0
FOR variable IN "(" arglist ")" DO statement
IF condition statement ELSE statement
IF condition statement
DETACH extcomline
REM arglist newline

variable: %c or %%c
c: any character
arglist:  (arg)*
arg: any-string

condition: NOT cond or cond
cond: ERRORLEVEL n or arg == arg or EXIST fname
n: any-number
fname: any-file-name
```

s0 is a command statement. Command statements may be single (execute one command) or multiple. Multiple command statements may be constructed to execute based on the successful completion of a previous command. Commands are defined as follows:

```
Single command:
command
Multiple commands:
command1 & command2
command1 || command2
command1 && command2
```

The output of one command may be redirected to another command or to

a file. Likewise, the input to a command may be redirected. The following syntax is acceptable for redirection:

```
command1 | command2
```

Command2 (above) may be a single command or one of the following:

```
redirect command3
command3 redirect
```

Command3 may be a single command or any of the following:

```
redirect command4
command4 redirect
command4
command redirect arglist
```

Command4 may be any of the following:

```
(statement)
command arglist
```

```
command: internal-command or external-command
extcomline: external-command arglist redirect
redirect: external-command arglist
external-command redirect arglist
```

```
redirect:  in out or out in
in -> "<" arg | epsilon
out -> ( ">" | ">>" ) arg | epsilon
```

The operator precedence is as follows, listed from lowest to highest:

&	Command separator
||	OR operator
&&	AND operator
|	Pipe operator
<~>~>>	I/O redirectors
()	Command grouper
^	Lexical escape character

Examples:

```
x & y | z    =>  x & (y | z)
x | y & z    =>  (x | y) & z
x || y | z       =>  x || (y | z)
a & b || c && d | e || f     =>  a & ((b || (c && (d | e))) || f)
```

In addition to the characters reserved by MS-DOS 3.*x*, the following characters are reserved and are no longer valid filename characters:

&	Command-separator operator
(	Opening command-grouping operator
)	Closing command-grouping operator
^	Lexical escape character

In addition to the words reserved by MS-DOS 3.*x*, the following words are reserved:

ELSE	Conditional exception command
DETACH	Background processing command
SETLOCAL	Opening batch file localization command
ENDLOCAL	Closing batch file localization command
EXTPROC	External batch processor command
CALL	Nested batch file command

3.3 Command Execution

Cmd.exe executes command lines under MS OS/2 the same way that *command.com* executes them under MS-DOS 3.*x*, with the exception of multiple command statements, which can be executed through the use of operators. The behavior of these statements is described in the following descriptions. Generally, execution of individual commands remains the same, but grouping and sequencing can be conditionally altered through the use of the following operators.

The AND Operator

The algorithm used for AND (&&) is as follows.

```
Call Dispatch on the left hand side of the operator.

If it returns with an error
    Return the error.
Else
    Call Dispatch on the right hand side of the operator.
    Return whatever Dispatch returns.
```

The OR Operator

The algorithm used for OR (||) is as follows:

```
Call Dispatch on the left hand side of the operator.

If it returns successful
    Return success.
Else
    Call Dispatch on the right hand side of the operator.
    Return whatever Dispatch returns.
```

The Command-Separator Operator

The algorithm used for the Command Separator (&) is as follows:

```
Call Dispatch on the left hand side of the operator.
Call Dispatch on the right hand side of the operator.
Return whatever Dispatch returns the second time.
```

3.4 Internal Command Differences

All commands return error codes that can be tested and acted on by the AND (&&) and OR (||) operators. As before, however, only the return code from external programs can be tested with the IF ERRORLEVEL command.

MS OS/2 supports the internal commands of MS-DOS essentially unchanged, although the following commands now take multiple arguments: **del**, **dir**, **type**, **vol**, **mkdir**, and **rmdir**. If an error occurs during the processing of any of the arguments to these commands, execution ends and the rest of the arguments are ignored.

3.5 External Command Differences

With few exceptions, external commands are processed in the same manner as they are in MS-DOS. However, external commands, batch processing, and the *startup.cmd* file must be constrained to prevent the actions of one program from affecting other programs. Also, to let applications run independently, environment variables are replicated on a per-program basis.

The algorithm for external command execution is as follows:

```
If the command name contains a path
    Search for the target using that path
Else
    Perform the target search using each of the individual
    elements in CMD's path variable.
```

The target of the search itself is a filename made up of the command name appended with a file extension of *.com*, *.exe*, or *.bat*. For example, if the filename given on the command line was *format*, **cmd** searches as follows:

1. First, for a file called *format.com*
2. Second, for *format.exe*
3. Third, for *format.cmd*

The first file found matching the filename halts the search. The algorithm for this procedure is as follows:

```
If the found file is COM or EXE
    Execute it using the DOS exec system call
Else the file is a batch file
    Process it using the batch processor within cmd
```

.Com and *.exe* files pass the entire command line (including the program name), as typed, to the **DosExecPgm** function call. Batch files pass any arguments included on the command line to the batch processor. Note

that any redirection that was also a part of the statement is already set up prior to reaching this point

The execution mode, asynchronous/save return code or asynchronous/discard return code, is passed to the **DosExecPgm** call so that it can properly execute the process. If the process is to be detached with an asynchronous/discard return code, **cmd** continues to run in the foreground. If not, **cmd** waits for the process before displaying another prompt. Note that the **DosCWait** call waits not only for the child process, but also for any grandchild processes that may have been executed by the child itself.

```
If the command was exec'd normally
    If the command was successfully exec'd
        The return code of the command is returned.
    Else
        The exec error code is returned.
If the command was detached
    If the command was successfully exec'd
        Return success.
    Else
        Return the exec error code.
```

External commands (this does not include batch files) cannot change the **cmd** environment, current directory, or drive.

3.6 Background Process Execution

The **detach** command detaches a process to the background. The process can be any program or external MS OS/2 command (including command-line options) that does not require input from the keyboard or output to the screen. The detached process should not issue any console I/O calls (**Vio***xxx*, **Kbd***xxx*, **Mou***xxx*, etc.) except for **VioPopUp** calls.

Detach detaches the process from its parent process. If the parent process is killed, the detached child process is not affected. The detached process continues to run as an orphan outside the caller's screem group.

3.7 Miscellaneous Information

Cmd.exe includes the caret escape character (^), which allows input of normally significant characters as common text. The caret is similar to the XENIX backslash character (\). When used outside of quoted strings during command line input, a caret causes the next character to be treated as normal text; for example:

Command	Output
echo hello ^>foo	hello >foo
echo "hello ^>foo"	hello ^>foo
echo he^llo	hello
echo "he^llo"	he^llo
echo hello ^^	hello ^

3.8 Redirection of Input and Output

This section discusses input/output redirection as it relates to the protected-mode command interpreter, *cmd.exe*.

Before a command is executed, its input and output may be redirected to or from files or other devices using special command sequences that are interpreted by *cmd.exe*, the protected-mode command processor.

Redirection is handle-based, where handles, identified by handle number, may be redirected to some other device or file. Subsequent input or output from or to that handle is received or obtained from the newly substituted device or file. Any handle with a single-digit number can be redirected, even if it is not currently in use. This includes the three standard handles, STDIN, STDOUT and STDERR (handle numbers 0, 1, and 2, respectively), as well as handles 3 through 9, which are not normally used by **cmd** or by the user (however, some of handles 3 through 9 are used by the system). At the completion of the command containing the redirection sequence, the original handle condition is restored.

Note

An application's input and/or output can be redirected only if the application reads from the standard input stream (STDIN), writes to the standard output stream (STDOUT), and (optionally) writes to the standard error stream (STDERR). If the application/command reads directly from the keyboard (via the **Kbd***xxx* calls) and writes directly to the screen (via the **Vio***xxx* calls), then its input/output cannot be redirected.

3.8.1 Redirection Sequences

The following redirection sequences may appear anywhere in a simple command, or within the body of one of the special commands (**if**, **for**, or **detach**). Redirection is performed in the order in which it is seen on the command line; therefore, the order of the redirection sequence is important. Environment variable and batch file variable substitution both occur prior to use of *word* or *digit* in performing the actual redirection.

word	A valid filename or device.
digit	One of the digits 0 through 9.
<*word*	Use the file *word* as STDIN (handle 0). The file must exist or an error will occur.
>*word*	Use the file or the device *word* as STDOUT (handle 1). If the file doesn't exist, create it; otherwise, truncate it to a length of zero.
>>*word*	Use the file *word* as STDOUT. If it exists, append to the file by first seeking to the end-of-file; otherwise, create the file.
<&*digit*	Duplicate STDIN from the handle *digit*.
>&*digit*	Duplicate STDOUT from the handle *digit*.

If *digit* preceeds any of the above forms, the redirected handle is the one specified by *digit*, instead of default handles 0 or 1. The following examples illustrate this.

2>&1	STDERR (handle 2) becomes a duplicate of STDOUT (handle 1). Any output written by a process to handle 2 is redirected to handle 1.
4>*fname*	Handle 4 becomes the file *fname*. Any output written by a process to handle 4 will be sent instead to the file *fname*.

No other characters, including white space, may exist between the leading digit (if used), the redirection indicator, the append indicator (if used), and the '&' character (if used). White space may appear only between the redirection or append indicators and their related *word* or between the '&' character and its related *digit*.

To be recognized, a leading digit must be seen to be the start of a new token. To qualify, it must be the first character on a line or it must be immediately preceeded by white space or by one of the following delimiters:

&|;,()="

If not, the digit is lexed as a part of the previous token and the redirection symbol following it is parsed as the beginning of the redirection. Redirection commands (with or without the leading digit) which immediately follow the closing quote of a quoted string, are recognized, but must not occur inside quoted strings.

3.8.2 Examples of Redirected I/O

The following are examples of correct syntax given the assumption that the applications send their input/output/error to STDIN/STDOUT/STDERR, respectively.

Redirecting STDOUT and STDERR to Separate Files

```
diskcopy a: b: >outlog 2>errlog
```

In this example, the **diskcopy** command's output, which is normally directed to STDOUT, is redirected to a file named *outlog* and any output directed to STDERR is redirected to a file named *errlog*

```
cc program.c  >outlog;2>errlog
```

If the C compiler sends its output to STDOUT and its errors to STDERR,

the output will be sent to *outlog* and the error report sent to *errlog*. Note the use of the semicolon (;) as a command delimiter.

Redirecting All Output to the Same File

```
(chkdsk *.* >filelog) 2>&1
```

This statement sends all output streams (STDOUT and STDERR) to the file named *filelog*. Note the use of parentheses as delimiters. This statement is equivalent to

```
chkdsk *.* >filelog 2>&1
```

Redirecting the Output of a Background Process

```
detach >datalog app 2>&1

detach app >datalog 2>&1
```

These statements execute the program *app* in the background, redirecting its handle 1 output (STDOUT) to a file named *datalog* and redirecting its handle 2 output (STDERR) to the current target of handle 1; in this case a file named *datalog*.

Redirecting STDERR Only

```
echo "hello"2>>outfile
```

This statement echoes the character string, "hello" to the screen but appendes the error output to the file named *outfile*.

Note that if the character string were not enclosed in quotes, e.g.,

```
echo hello2>>outfile
```

the character string, "hello2" would be appended to the file *outfile* because the quote characters in the preceeding example act as delimiters. You can also prevent this problem by using whitespace to separate the various portions of the statement, e.g.,

```
echo hello 2>>outfile
```

appends the string "hello" to the file named *outfile.*

Redirection and Conditional Execution

```
dir xyz || dir xyz 2>errlog
```

If the file *xyz* does not exist, the error output of the first **dir** command will go to the screen, while the error output of the second **dir** command will be sent to the file *errlog.*

```
>outfile (if exist makefile echo "It exists")
```

The character string, "It exists", is written to the file named *outfile,* if the file named, *makefile* exists.

3.8.3 Examples of Incorrect Syntax

The following are examples of incorrect syntax:

```
make >makelog2>errlog
echo hello4>>outfile
>outfile if exist makefile echo "It exists"
>makelog det make 2>&1
```

In each case, only one redirection command for each handle will be performed. The first two examples are missing delimiter characters between the filename and the handle number. The third and fourth examples are missing the command grouping parenthesis, so the commands after the redirection symbol are never performed.

Suppose the user typed the following command:

```
dir >fname1 >fname2 >fname3 >fname4
```

The four redirection operations are screened following parsing and duplicates removed with the last one taking precedence. The output still winds up in fname4, with no creation of the files: fname1, fname2, fname3.

Chapter 4
Device I/O

4.1 Device I/O Function Call Summary

This chapter is the reference for the MS OS/2 Device I/O function calls. These calls are summarized as follows:

DosBeep	Generate Sound from Speaker
DosDevConfig	Get Device Configuration
DosDevIOCtl	I/O Control for Devices
DosCLIAccess	Request CLI/STI Privilege
DosIOAccess	Request I/O Access to Devices
DosPortAccess	Request Port Access
KbdCharIn	Read Character-Scan Code
KbdDeRegister	Deregister Keyboard Subsystem
KbdFlushBuffer	Clear Keystroke Buffer
KbdGetStatus	Get Keyboard Status
KbdPeek	Peek at Character
KbdRegister	Register Video Subsystem
KbdSetStatus	Set Keyboard Status
KbdStringIn	Read Character String
VioDeRegister	Deregister Video Subsystem
VioEndPopUp	Deallocate Popup Display Screen
VioGetAnsi	Get ANSI Status
VioGetBuf	Get Logical Video Buffer
VioGetConfig	Get Video Configuration
VioGetCurPos	Get Cursor Position
VioGetCurType	Get Cursor Type
VioGetFont	Get Font Selector
VioGetMode	Get Display Mode
VioGetPhysBuf	Get Physical Video Buffer

VioPopUp	Allocate Popup Display Screen
VioPrtSc	Print Screen
VioPrtScToggle	Toggle Print Screen
VioReadCellStr	Read Character-Attribute String
VioReadCharStr	Read Character String
VioRegister	Register Video Subsystem
VioSavReDrawUndo	Undo Previous Save/Redraw Registration
VioSavReDrawWait	Wait for Screen Save/Refresh Notification
VioScrLock	Lock Screen for I/O
VioScrollDn	Scroll Page Down
VioScrollLf	Scroll Page Left
VioScrollRt	Scroll Page Right
VioScrollUp	Scroll Page Up
VioScrUnLock	Unlock Screen for I/O
VioSetAnsi	Turn ANSI On or Off
VioSetCurPos	Set Cursor Position
VioSetCurType	Set Cursor Type
VioSetMode	Set Display Mode
VioShowBuf	Update Display with Logical Video Buffer
VioWrtCellStr	Write Character-Attribute String
VioWrtCharStr	Write Character String
VioWrtCharStrAtt	Write Character String with Attribute
VioWrtNAttr	Replicate Attribute
VioWrtNCell	Replicate Cell
VioWrtNChar	Replicate Character
VioWrtTty	Write TTY String

Chapter 5
Device Monitor

5.1 Introduction

Character Device Monitors provide a mechanism for applications or subsystems to monitor all characters passing through a device driver. This mechanism allows any registered process to remove, insert, or modify the information passing through the device.

Below is a diagram that shows how monitors fit into the system structure and how MS OS/2 supports these functions.

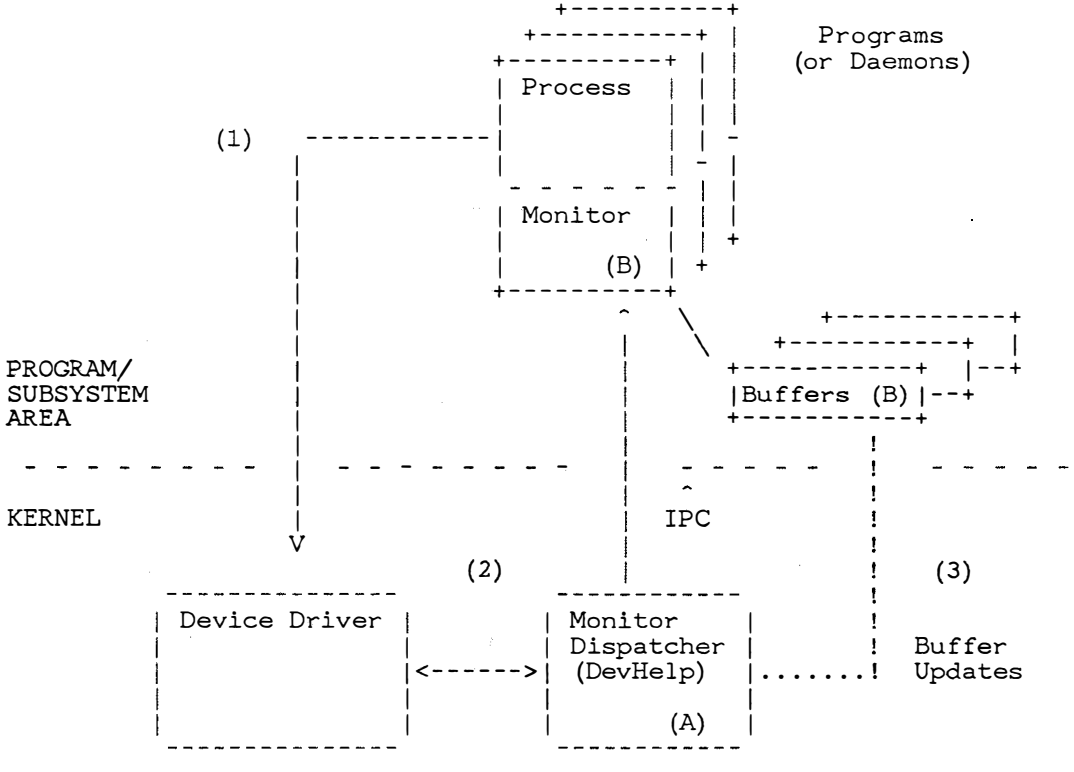


Figure 5.1 Character Device Monitors

5.2 Monitor Interfaces

There are three areas of interest in Figure 5.1:

- Program/Daemon to Device Driver interface (1)
- Device Driver to Monitor Dispatcher interface (2)
- Monitor Dispatcher to Monitor Process interface (3)

Program/Daemon to Device Driver

This interface relates to setting up the monitor connection and tearing it down. This is generally called the *registration* process.

- **DosMonOpen** registers the monitor with the device driver.
- **DosMonClose** deregisters the monitor.

Device Driver to Monitor Dispatcher

This interface uses the following **DevHelp** routines to allow communication between a device driver and the monitor dispatcher.

- **MonitorCreate** creates a monitor.
- **Register** adds a monitor to the chain of monitors.
- **Deregister** removes the monitor associated with a specified task from the monitor chain.
- **MonWrite** passes data records to the monitors.
- **MonFlush** removes all data from the monitor stream.

Monitor Dispatcher to Monitor Process

The monitor dispatcher communicates with the monitor process using these interfaces.

- Interprocess Communication
- Buffers for sending and receiving records

5.3 Module Description

Monitor Dispatcher

This package is common for all character device drivers and serves the device drivers as well as the monitors in user storage space.

Monitor Buffer Management

Figure 5.2, "Monitor Details," is a sample arrangement of the components in the monitors. It also includes design considerations.

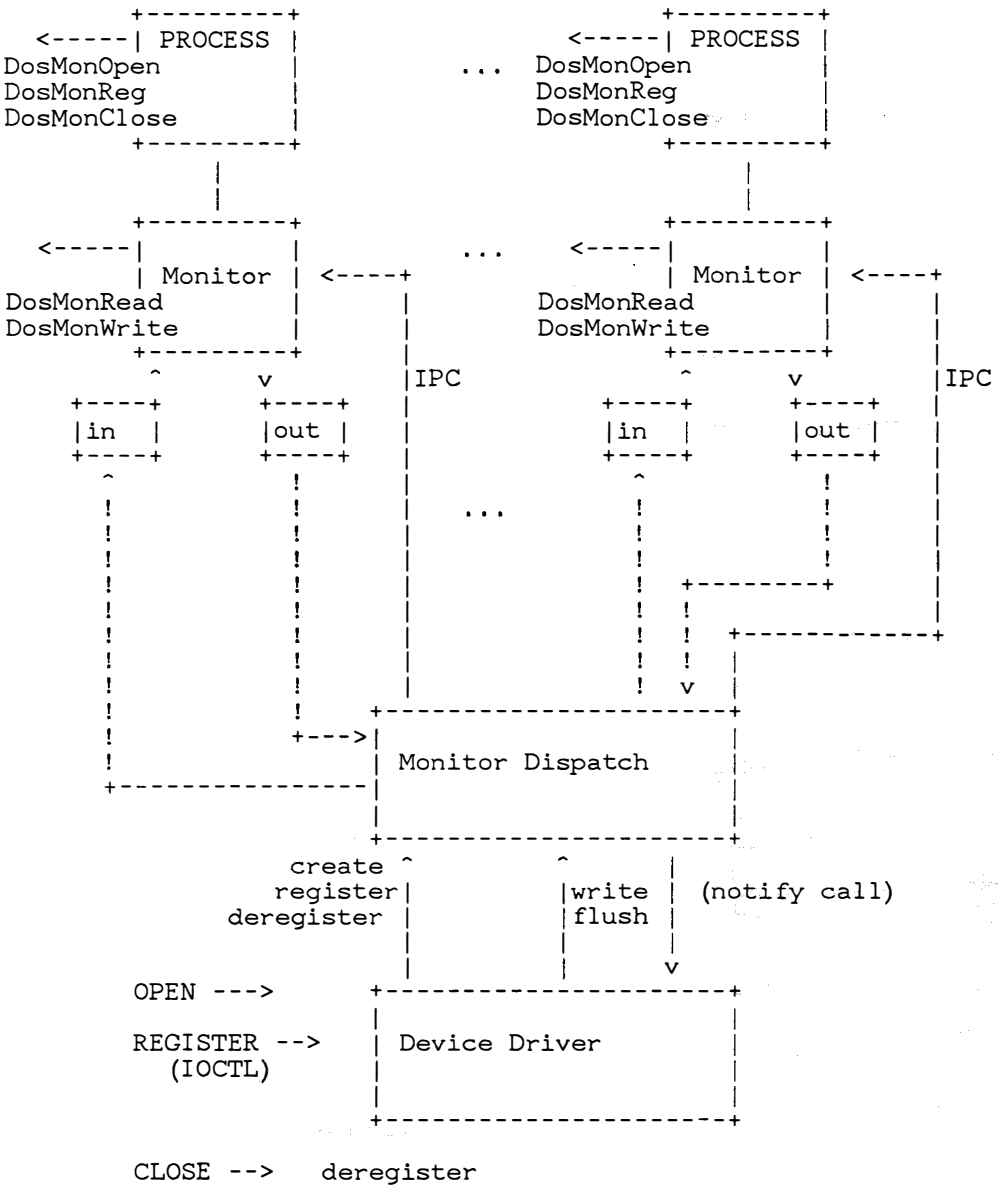

PROCESS
DosMonOpen
DosMonReg
DosMonClose
Monitor
DosMonRead
DosMonWrite
in
out
IPC
Monitor Dispatch
create
register
deregister
write
flush
(notify call)
OPEN --->
REGISTER -->
(IOCTL)
Device Driver
CLOSE --> deregister

Figure 5.2 Monitor Details

5.4 Device Monitor Record

General Monitor Record Format

Each monitor record can be of variable length. However, there must be a word of flags at the beginning of each monitor record. The monitor flags are defined as follows:

Byte 0:

Bit	Meaning
0	Open
1	Close
2	Flush
3–7	Reserved

Byte 1:

Bit	Meaning
0–7	Device driver dependent

5.5 Device Monitor Calls

The device monitor function calls are summarized as follows:

DosMonOpen	Open Connection to MS OS/2 Device Monitor
DosMonClose	Close Connection to MS OS/2 Device Monitor
DosMonReg	Register Set of Buffers as Monitor
DosMonRead	Read Data from Monitor Structure
DosMonWrite	Write Data to Monitor Structure

Chapter 6
Dynamic Linking

6.1 Introduction

Dynamic linking is the delayed binding of application program's external references to subroutines. For example a program may dynamically link to one or more libraries. A library may in turn dynamically link to other libraries. The libraries are created by the linker, which links object files with their corresponding definition files. Object files that provide entry points do so by specifying those entry points in their corresponding definition file. There are two forms of dynamic linking: *load-time* and *run-time.*

Dynamic-link routines, both load-time and run-time, are shared by invoking applications. Both the code and data segments of a dynamic-link routine are shared (unless the nonshared data option is selected during the module link process). It is a dynamic-link routine's responsibility to serialize access, as required, to its shared data segments.

Note

A dynamic-link routine can allocate non-shared memory dynamically.

6.1.1 Load-time Dynamic Linking

In load-time dynamic linking, a program calls a dynamically-linked routine just as it would any external routine. When the program is assembled or compiled, a standard external reference is generated. At link time, the programmer specifies one or more libraries that contain routines to satisfy external references. External routines to be dynamically linked contain special *definition records* in the library. These definition records tell the linker that the routines in question are to be dynamically linked and provide the linker with dynamic-link module names and entry names.

A module name is the name of a special *.exe* file with the filename extension *.DLL* containing dynamic-link entry points. The linker stores module-name/entry-name pairs describing the dynamic-link routines in the *.exe* file created for the program. When the calling program is run, MS OS/2 loads the dynamic-link routines from the modules specified and links the calling program to the called routines.

6.1.2 Run-time Dynamic Linking

To invoke run-time dynamic linking, the application uses the following system calls:

DosFreeModule	Free Dynamic-Link Module
DosGetModHandle	Get Dynamic-Link Module Handle
DosGetModName	Get Dynamic-Link Module Name
DosGetProcAddr	Get Dynamic-Link Procedure Address
DosLoadModule	Load Dynamic-Link Module
DosGetResource	Get Resource Segment Selector

6.2 Initialization Routines

A dynamic-link module can optionally have an initialization routine that is invoked when the dynamic link module is first loaded, before any dynamic-link routine is actually called. The initialization routine is the entry point specified in an END statement in one of the source files making up the dynamic-link module, although only one source file may have an END statement identifying an initialization routine. If no entry point is specified in any END statement, no initialization routine is called. Input to the initialization routine is as follows:

AX = *ModuleHandle*, where *ModuleHandle* has the same definition as described in the functional descriptions that follow.

SI = HEAPSIZE parameter from the *.exe* file.

DI = Module handle for the dynamic-link module.

DS = the library's DGROUP data segment, if one exists. Otherwise, DS = the application's DS.

Initialization routines exit via far return. Contrary to the standard convention, initialization routines set AX not equal to zero to indicate success, and equal to zero to indicate failure.

6.3 Demand Loading

Compared to the practice of preloading code segments before program execution, the MS OS/2 demand-load feature supports the loading of code segments on demand *during* program execution. Code segments are discarded as required to provide space for the code segments of other currently active programs. Demand-loaded code segments are identified in the program's *.exe* file header.

Demand load is supported for the code and data segments of any program, as well as for libraries. Initial code and auto data segments are always preloaded.

Chapter 7
Error Handling

7.1 Introduction

Hard errors in MS OS/2 are handled by a process that has arranged, through the **DosSystemService** call, to receive notification of hard errors. Only one process in the system may use this service ; this process will normally be the session or window manager. On special-purpose dedicated systems, one process (possibly a dedicated system process) will take responsibility for this function.

When a hard error needs to be serviced, the **DosSystemService** call returns, passing back the necessary information the handler needs to act upon the error.

7.2 Hard-Error Override

Hard error processing normally occurs without direct application notification. **DosError** allows an MS OS/2 application to process these events. This function allows the application program to notify MS OS/2 that all permanent errors associated with the process, or for an open handle belonging to the process, are to be reflected as immediate failures.

7.3 Errors from Function Requests

MS OS/2 function calls return error codes in AX when a requested service fails. **DosErrorClass** helps applications deal with these error codes by classifying them and recommending a course of action. The application could then follow the recommended recovery action or try something else instead.

7.4 Error Handling Calls

The following system calls are used by MS OS/2 for error handling:

DosErrClass	Classify Error Codes
DosError	Enable Hard-Error processing
DosSetVec	Establish Handler for Exception Vector
DosSystemService	DOS System Process Services

Note

All the MS OS/2 function calls return AX equal to zero if the operation was successful. If an error condition was encountered, AX is equal to an error code.

Chapter 8
Family API Program Execution Control

8.1 Introduction

The following interfaces are for Family API (Application Program Interface) use. They aid the programmer in writing an application in such a way that it can execute in both the MS OS/2 protected-mode environment and in the MS-DOS 2.*x* and 3.*x* environments.

8.2 Family API Program Execution Control Calls

The routines in this section are:

BadDynLink	Bad Dynamic Link
DosGetMachineMode	Get Current Processor Mode

Chapter 9
File I/O Calls

9.1 Introduction

This chapter describes the MS OS/2 file system interface calls.

Existing file systems that conform to the Standard Application Program Interface (Standard API) are also described in this section; however, they may not necessarily support all the described information kept on a file basis. In these cases, file system drivers are required to return to the application a null value for the unsupported parameter. Null pointers are defined to be 00000000H throughout this chapter.

File handle values of FFFFH do not represent actual file handles but are used throughout the file system interface to indicate specific actions to be taken by the file system. Use of this "special file handle" where it is not expected by the file system will result in an error.

9.2 File I/O Calls

MS OS/2 provides the following file I/O function calls:

DosBufReset	Flush File Buffers
DosChdir	Change Current Directory
DosChgFilePtr	Change (Move) File Read/Write Pointer
DosClose	Close File Handle
DosDelete	Delete File
DosDupHandle	Duplicate File Handle
DosFileLocks	Lock/Unlock Range of Bytes in Open File
DosFindClose	Terminate Usage of Directory Search Handle
DosFindFirst	Find First Matching File
DosFindNext	Find Next Matching File
DosMkdir	Make Subdirectory
DosMove	Move File

DosNewSize	Change Size of File
DosOpen	Open/Create File
DosQCurDir	Query Current Directory
DosQCurDisk	Query Current Default Drive
DosQFHandState	Query File Handle State
DosQFileInfo	Query File Information
DosQFileMode	Query File Mode
DosQFSInfo	Query File System Information
DosQHandType	Query Handle Type
DosQVerify	Query Verify Setting
DosRead	Read from File
DosReadAsync	Asynchronous Read from File
DosRmdir	Remove Subdirectory
DosSelectDisk	Select Disk
DosSetFHandState	Set File State
DosSetFileInfo	Set File Information
DosSetFileMode	Set File Mode
DosSetFSInfo	Set File System Device Information
DosSetMaxFH	Define New Maximum File Handle
DosSetVerify	Set Verify Setting
DosWrite	Write to File or Device
DosWriteAsync	Asynchronous Write to File or Device

Chapter 10
Interprocess Communication: Pipes, Queues, and Semaphores

10.1 Introduction

MS OS/2 supports four general methods of Interprocess Communication (IPC) as the means for allowing processes to communicate with each other:

- Flags
- Pipes
- Semaphores
- Shared Memory

The IPC functions provided allow

- Communication via flags
- Communication via messages
- Coordinating execution among several processes
- One process to directly control execution of other processes

10.2 Interprocess Communication Calls

MS OS/2 uses the following calls for Interprocess Communication (IPC):

DosFlagProcess	Set Process External Event Flag
DosMakePipe	Create Pipe
DosCloseQueue	Close Queue
DosCreateQueue	Create Queue
DosOpenQueue	Open Queue
DosPeekQueue	Peek Queue
DosPurgeQueue	Purge Queue
DosQueryQueue	Query Size of Queue
DosReadQueue	Read from Queue
DosWriteQueue	Write to Queue

DosCloseSem	Close System Semaphore
DosCreateSem	Create System Semaphore
DosMuxSemWait	Wait for One of *n* Semaphores to be Cleared
DosOpenSem	Open Existing System Semaphore
DosSemClear	Clear (release) Semaphore
DosSemRequest	Request Semaphore
DosSemSet	Set Semaphore Owned
DosSemSetWait	Set Semaphore and Wait for Next Clear
DosSemWait	Wait for Semaphore to be Cleared

Chapter 11
Memory Management

11.1 Introduction

MS OS/2 protected-mode memory management enables an application to allocate large real memory—beyond the 640K boundary imposed by previous versions of DOS. Memory management is supported with both memory compaction and segment swapping.

11.2 Protected-mode-only System

Figure 11.1 shows the memory layout for an MS OS/2 protected-mode-only system. In protected mode there is one LDT per process for all threads running under that process. MS OS/2 dynamic-link routines have LDT, as opposed to GDT, descriptors. For more information on dynamic-link routines, see the *Microsoft Operating System/2 Programmer's Guide.*

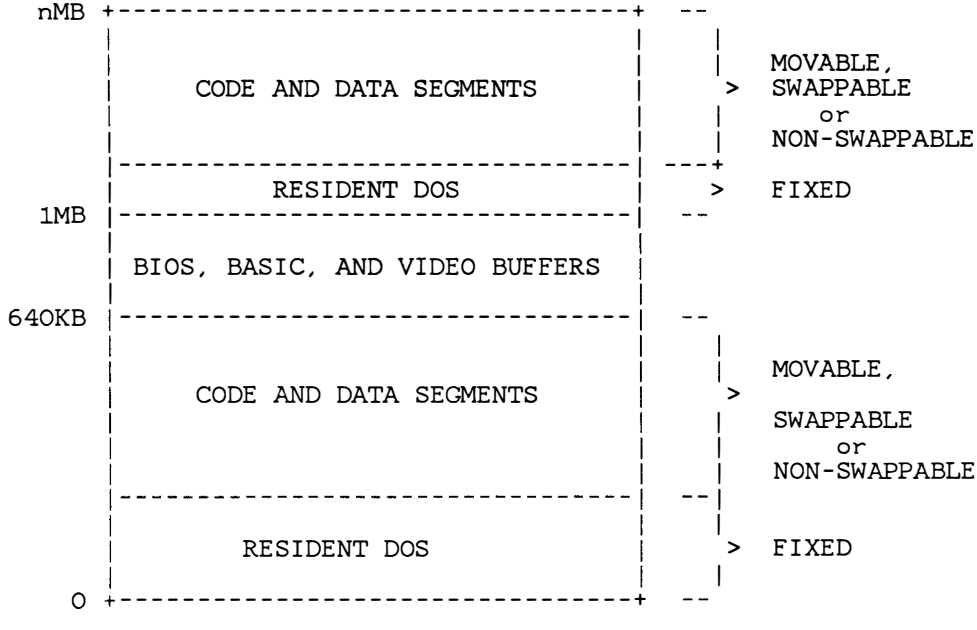


Figure 11.1 Protected-mode-only Memory Layout

11.3 Protected/Compatibility-mode System

The following figure shows the memory layout for an MS OS/2 system running both protected-mode applications and a single, real compatibility-mode application. Protected-mode code and data segments are loaded above a boundary specified in *config.sys*. Figure 11.2 assumes the boundary is set at 640K.

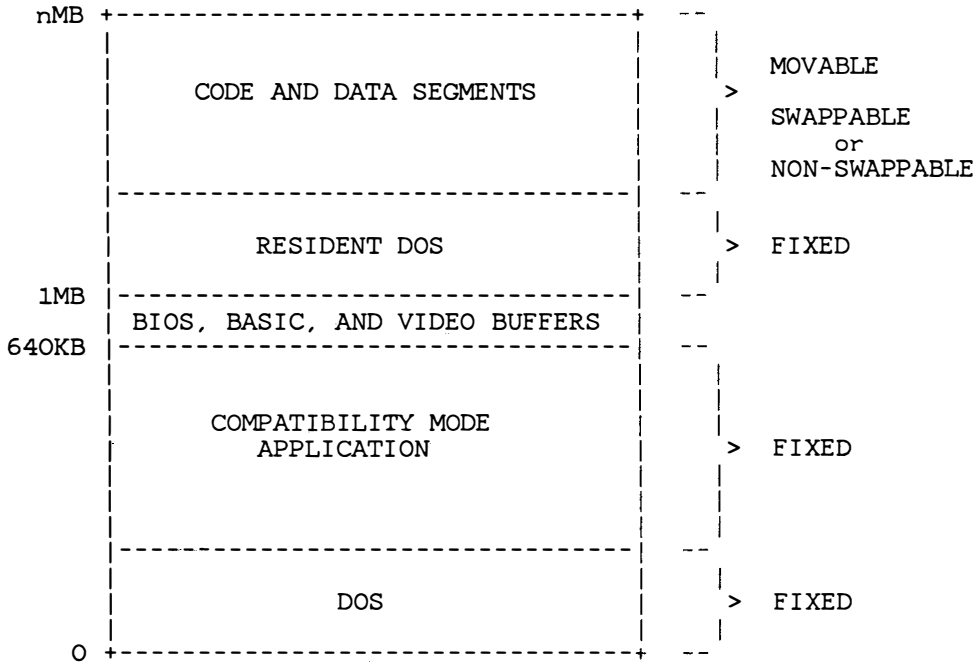


Figure 11.2 Protected/Compatibility-mode Memory Layout

11.4 Memory Management Function Call Summary

MS OS/2 supports the following memory management functions:

DosAllocHuge	Allocate Huge Memory
DosAllocSeg	Allocate Segment
DosAllocShrSeg	Allocate Shared Segment
DosCreateCSAlias	Create CS Alias
DosFreeSeg	Free Segment
DosGetHugeShift	Get Shift Count
DosGetShrSeg	Access Shared Segment
DosGiveSeg	Give Access to Segment
DosReAllocHuge	Change Huge Memory Size
DosReAllocSeg	Change Segment Size
DosSubAlloc	Suballocate Memory within Segment
DosSubFree	Free Memory Suballocated within Segment
DosSubSet	Initialize or Set Allocated Memory

11.5 Protected-mode Memory Management

Figure 11.3 depicts the virtual-segmented, protected-mode memory management of MS OS/2. Note that each application has its own distinct address space, mapped by a Local Descriptor Table (LDT) and a Global Descriptor Table (GDT). The LDT provides a per application private address space, while the GDT provides addressability for system-wide data and programs that are shared among all applications. Together, these tables provide a virtual address space of 1 gigabyte (2**30). The LDT and GDT provide a mapping from this 1-gigabyte virtual address space to the 16-megabyte (maximum) physical address space of the 80286.

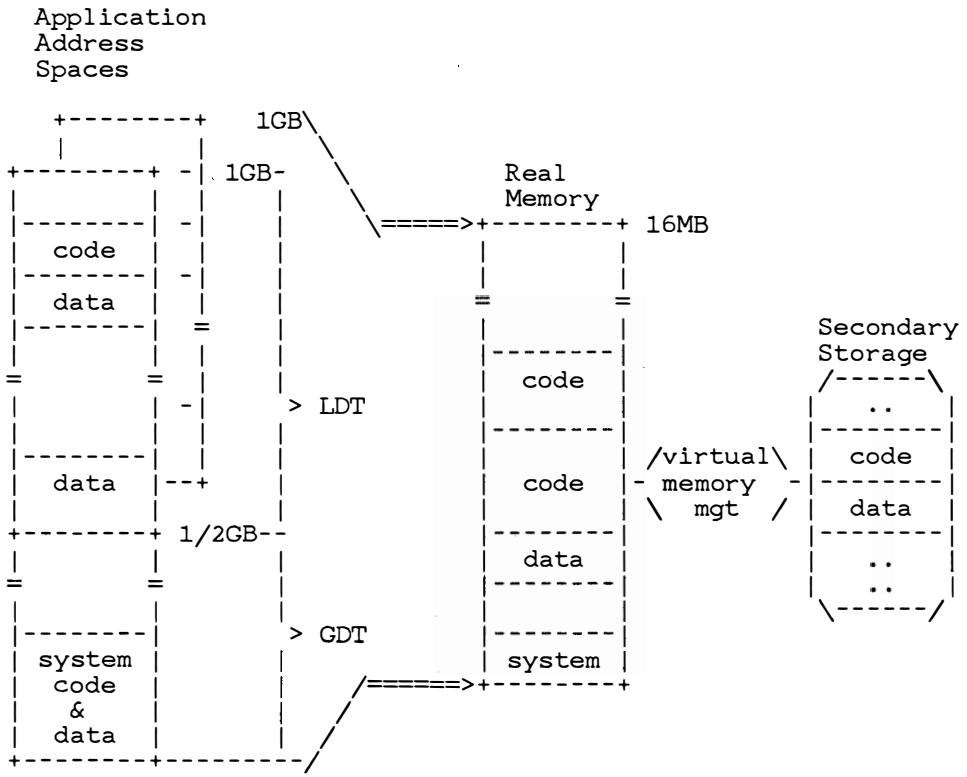


Figure 11.3 Protected-Mode Memory Management

Features of this memory management support include:

Storage over-commitment. The amount of virtual memory allocated at any instant for data and code segments can, and typically will be, greater than the amount of real memory available.

The memory for the system as a whole can be over-committed, as can the memory for a single application.

Segment discard. Real memory occupied by pure segments that are still in the address space of an application, but are not currently in use, can be reclaimed by "discarding" the segment. When the discarded segment is later referenced, a fresh copy is read from the disk.

Segment motion. Because segments have variable lengths, real memory is subject to external fragmentation. "Holes" of deallocated real memory, each of which is insufficient in size to satisfy a request for memory, but which together would be sufficient, are merged in order to satisfy the request.

Protection. Applications only have addressability to memory segments specifically authorized to them by the system. The MS OS/2 system and each individual application are protected from access by other applications.

11.5.1 Real-memory Map (Protected-mode-only)

Figure 11.4 depicts the use of real memory for a protected-mode-only system:

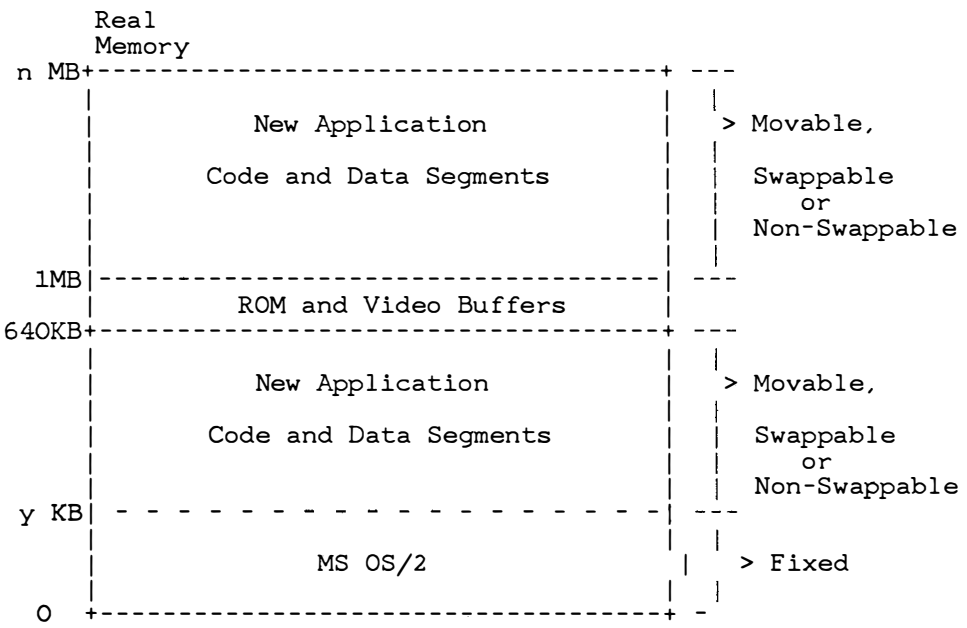


Figure 11.4 Real-memory Map (Protected-mode-only)

Note the "y KB" boundary. This is a movable boundary separating the fixed and movable regions of memory; its location varies over time, depending on the system load and the amount of real RAM installed.

11.5.2 Real-memory Map (Compatibility Mode)

Figure 11.5 depicts the use of real memory for a *compatibility-mode system.* A compatibility-mode system allows execution of a single real-mode application (one originally intended for MS-DOS 2.*x* or) in addition to new, protected-mode applications.

A real-mode application can execute only below 1 megabyte (in fact, because of the reserved ROM and video buffer space, only below 640K), while a protected-mode application can execute at any address.

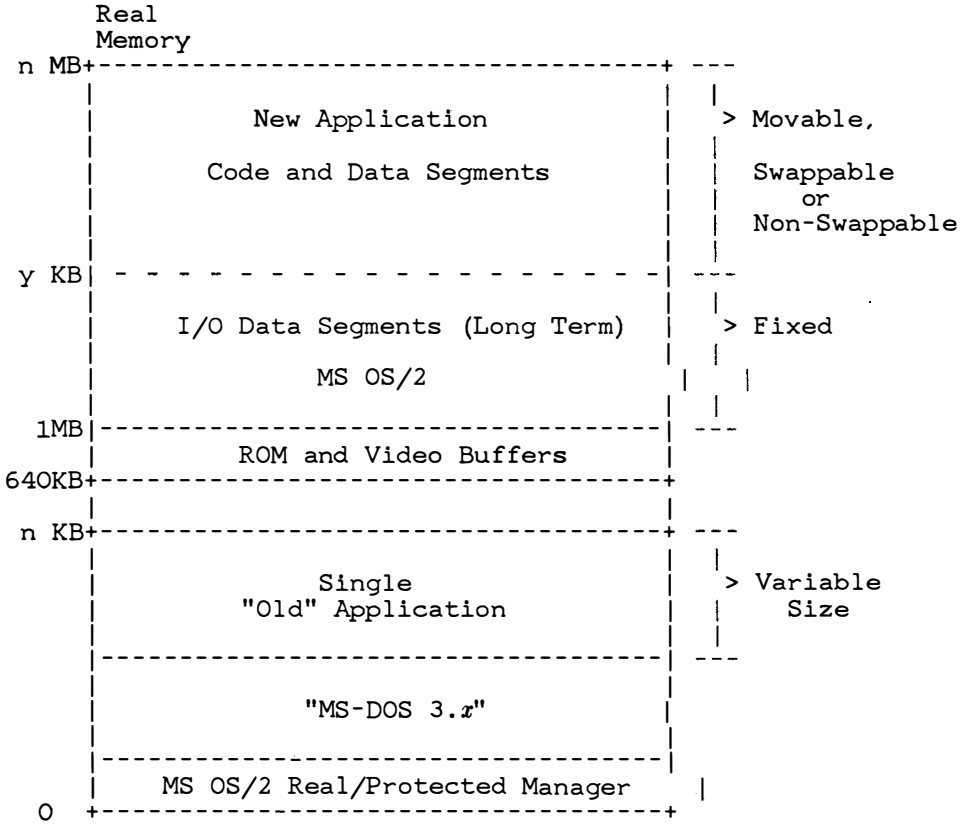


Figure 11.5 Compatibility-mode Memory Map

Note the "y KB" boundary. As in the real-memory map in Figure 11.4, this is a movable boundary separating the fixed and movable regions of memory; its location also varies over time, depending on the system load and the amount of real RAM installed.

Note the "n KB" boundary. This point, which defines the logical "end of memory" for the MS-DOS 3.*x* application, may vary up to the 640K limit. Located at the end of the real-mode memory partition, this boundary is fixed at boot time and is not dynamically variable in a running system. Above "n KB" is protected-mode memory.

11.6 MS OS/2 Program Segment (Compatibility-mode-only)

When you type an external command or execute a program through the **Exec** system call, MS OS/2 determines the lowest available free memory address to use as the start of the program. The memory starting at this address is called the *program segment.*

The **Exec** system call sets up the first 256 bytes of the program segment for the program being loaded into memory. The program is then loaded following this block. An *.exe* file with the *minalloc* and *maxalloc* variables both set to zero is loaded as high as possible in memory.

At offset 0 within the program segment, MS OS/2 builds the program segment prefix. The program returns from **Exec** by one of five methods:

- by issuing an INT21H with AH=4CH
- by issuing an INT21H with AH=31H (Keep Process)
- by a long jump to offset 0 in the program segment prefix
- by issuing an INT20H with CS:0 pointing at the PSP
- by issuing an INT21H with register AH=0 and with CS:0 pointing at the PSP

Each of these methods transfers control to the program that issued the **Exec**. The first two methods return a completion code, and are the preferred methods. They also restore the addresses of Interrupts 22H, 23H, and 24H (Terminate Process Exit Address, CONTROL-C Handler Address, and Critical Error Handler Address) from the values saved in the program segment prefix of the terminating program. Control then passes to the terminate address.

If this is a program returning to *command.com,* control transfers to its resident portion. If this program is a batch file (in process), it continues. Otherwise, *command.com* performs a checksum on the transient part, reloads it if necessary, issues the system prompt, and waits for you to type another command.

When a program receives control, the following conditions are in effect:

For all programs:

- The segment address of the passed environment is at offset 2CH in the program segment prefix.
- The environment is a series of ASCII strings (totaling less than 32K) in the form

  NAME=*parameter*
- A byte of zeros terminates each string, and another byte of zeros terminates the set of strings.

  Following the last byte of zeros is a set of initial arguments that the operating system passes to a program. This set of arguments contains a word count followed by an ASCII string. If the file is in the current directory, the ASCII string contains the drive and pathname of the executable program as passed to the **Exec** function call. If the file is not in the current directory, **Exec** concatenates the name of the file with the name of the path. Programs can use this area to determine where the program was loaded.
- The environment built by the command processor contains at least a COMSPEC=*string* (the parameters on COMSPEC define the path that MS OS/2 uses to locate *command.com* on disk). The last **path** and **prompt** commands issued are also in the environment, along with any environment strings you have defined with the MS OS/2 **set** command.
- **Exec** passes a copy of the invoking process environment. If your application uses a "keep process" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if you issue subsequent **set**, **path**, or **prompt** commands. Conversely, any modification of the passed environment by the application is not reflected in the parent process environment. For instance, a program cannot change the MS OS/2 environment values as the **set** command does.
- The Disk Transfer Address (DTA) is set to 80H (the default DTA in the program segment prefix). The program segment prefix contains file control blocks at 5CH and 6CH. MS OS/2 formats these blocks using the first two parameters that you typed when entering the command. If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field is not valid.
- An unformatted parameter area at 81H contains all the characters typed after the command (including leading and embedded delimiters), with the byte at 80H set to the number of characters. If you

type a less-than (<) or greater-than (>) symbol, or parameters on the command line, they do not appear in this area (nor do the filenames associated with them). Redirection of standard input and output is transparent to applications.

- Offset 6 (one word) contains the number of bytes available in the segment.
- Register AX indicates whether the drive specifiers (entered with the first two parameters) are valid, as follows:

  AL=FF if the first parameter contained an invalid drive specifier (otherwise, AL=00)

  AH=FF if the second parameter contained an invalid drive specifier (otherwise, AH=00)
- Offset 2 (one word) contains the segment address of the first byte of *unavailable* memory. Programs must not modify addresses beyond this point unless these addresses were obtained by allocating memory via Function Request 48H (Allocate Memory).

For executable (*.exe*) programs:

- DS and ES registers point to the program segment prefix.
- CS, IP, SS, and SP registers contain the values that icrosoft **link286** sets in the *.exe* image.

For executable (*.com*) programs:

- All four segment registers contain the segment address of the initial allocation block that starts with the control block of the program segment prefix.
- *.Com* programs allocate all user memory. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.
- The Instruction Pointer (IP) is set to 100H.
- The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

- The *.com* program places a word of zeros on top of the stack. Then by doing a RET instruction last, your program can exit to *command.com*. This method assumes, however, that you have maintained your stack and code segments.

Figure 11.6 illustrates the format of the program segment prefix. All offsets are in hexadecimal.

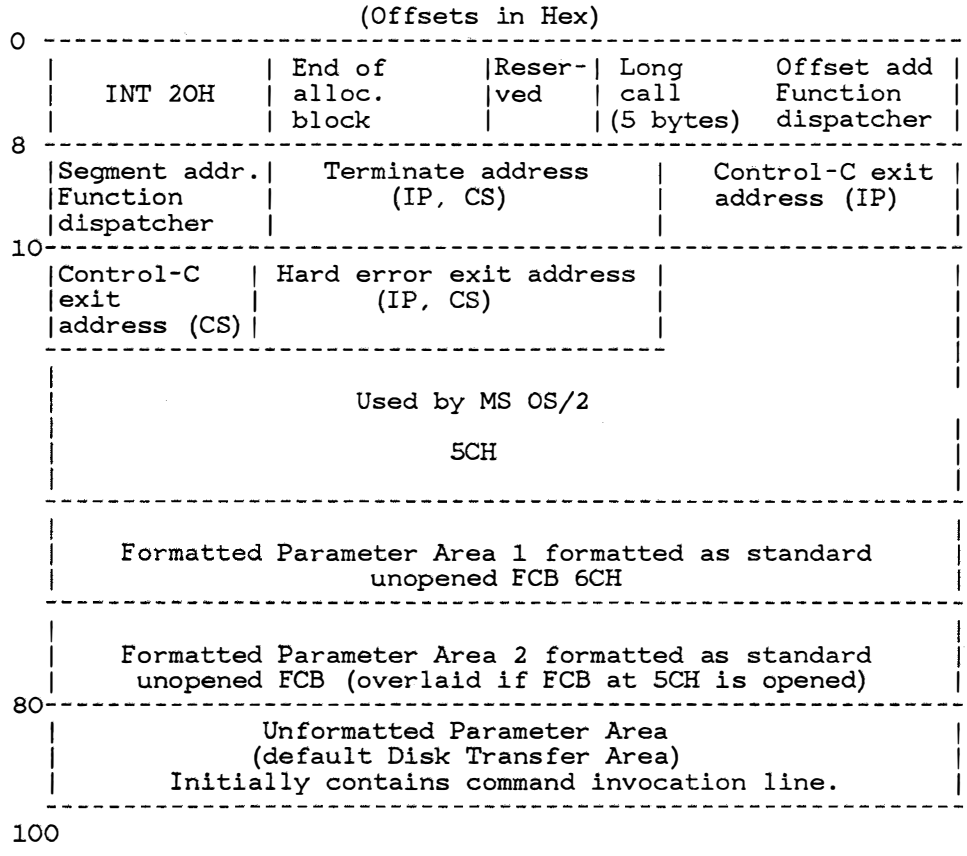


Figure 11.6 Program Segment Prefix

Important

Programs must not alter any part of the program segment prefix below offset 5CH.

11.7 Memory Management

MS OS/2 protected-mode memory management enables an application to allocate large real memory — beyond the 640K limit imposed by previous versions of DOS. Memory management is supported with both memory compaction and segment swapping.

11.7.1 The Protected-mode-only System

Figure 11.7 shows the memory layout of an MS OS/2 protected-mode-only system. In protected mode there is one LDT (*Local Descriptor Table*) per process—for *all* threads running under that process. MS OS/2 dynamic-link routines have LDT, as opposed to GDT (*Global Descriptor Table*), descriptors.

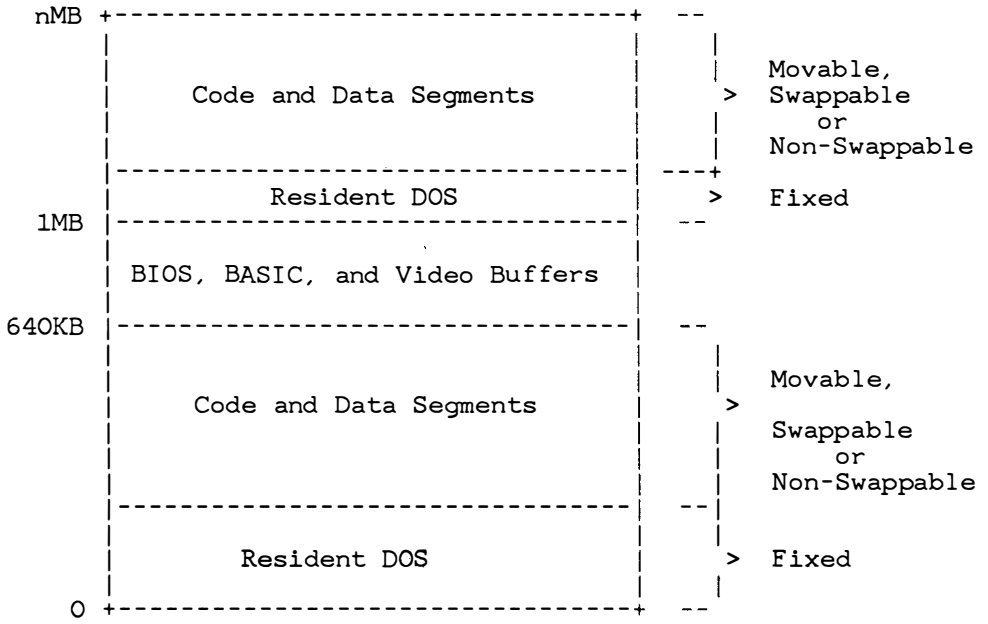


Figure 11.7 Protected-mode-only Memory Layout

11.7.2 The Protected/Compatibility-mode System

Figure 11.8 shows the memory layout of an MS OS/2 system running both protected-mode applications and a single, real compatibility-mode application. Protected-mode code and data segments are loaded above a boundary specified in *config.sys*. The following figure assumes the boundary is set at 640K:

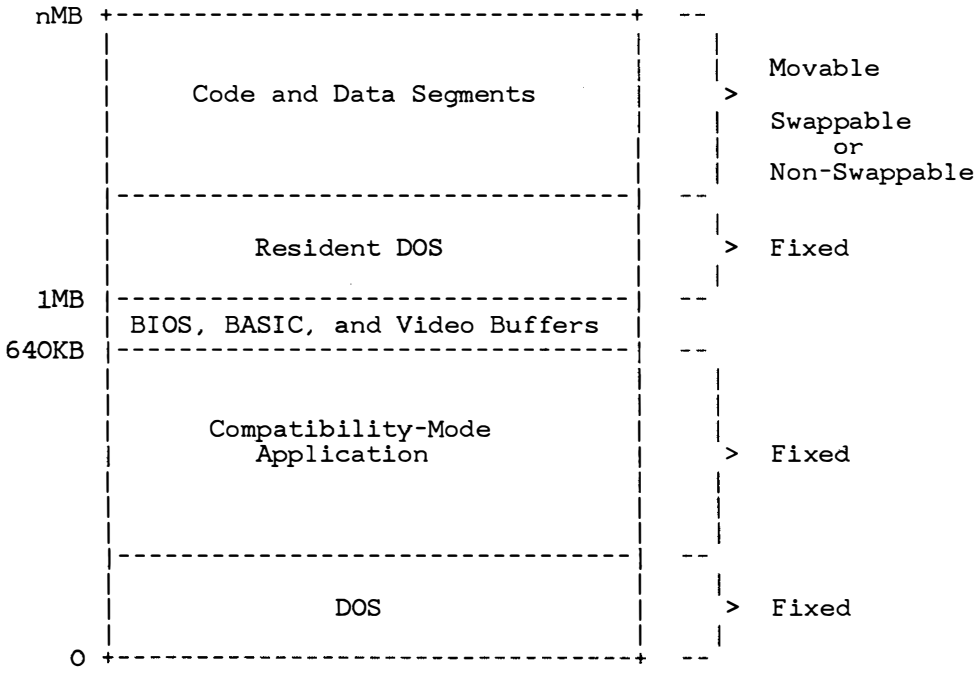


Figure 11.8 Protected/Compatibility-mode Memory Layout

11.7.3 A Summary of Memory Management System Calls

The following memory management interfaces are supported:

DosAllocSeg	Allocate Memory Segment
DosAllocShrSeg	Allocate Shared Memory Segment
DosGetShrSeg	Access Shared Memory Segment
DosGiveSeg	Gives Memory Segment
DosReallocSeg	Change Size of Segment
DosFreeSeg	Deallocate Segment
DosAllocHuge	Allocate Huge Memory Size

DosGetHugeShift	Return Shift Count
DosReallocHuge	Change Memory Allocation Size
DosCreateCSAlias	Create CS Alias
DosSubAlloc	SubAllocate memory within Segment
DosSubFree	Free memory suballocated within Segment
DosSubSet	Initialize or Set SubAllocated memory

Chapter 12
Message Service

12.1 Introduction

This chapter describes the MS OS/2 message function calls.

12.2 Message Services Calls

The message function calls are summarized as follows:

DosGetMessage	Get System Message
DosInsMessage	Insert Variable Text Strings in Message
DosPutMessage	Output Message to Handle

12.2.1 DosGetMessage

Purpose:

The **DosGetMessage** call retrieves a message from the specified system message file and inserts variable information into the body of the message.

Calling Sequence:

```
EXTRN DOSGETMESSAGE:FAR

PUSH@ OTHER IvTable          ; Table of variables to insert
PUSH  WORD  IvCount          ; Number of variables
PUSH@ OTHER DataArea         ; Buffer address to return message
PUSH  WORD  DataLength       ; Length of buffer
PUSH  WORD  MsgNumber        ; Number of the message
PUSH@ ASCIIZ filename        ; Message file name
PUSH@ WORD  MsgLength        ; Length of returned message
CALL  DOSGETMESSAGE
```

where:

IvTable is a pointer table. Each double-word far pointer points to an ASCIIZ or null-terminated DBCS string (variable insertion text). Zero to nine strings can be present.

IvCount is in the range from 0 to 9, the count of variable insertion text strings. If *IvCount* is zero, the next value on the stack, *IvTable*, is ignored.

DataArea is the storage area to which the system returns the requested message.

DataLength is the length, in bytes, of your storage area.

MsgNumber is the requested message number.

Filename is the pathname of the file in which the message can be found. This file is previously prepared by the **mkmsgf** utility.

MsgLength is the length of the message returned.

Returns:

IF (AX = 0)
    No error.

ELSE
    AX = Error code:

- Message file not found.
- Message text too long for buffer.
- Message number not found.

Comments:

If *IvCount* is greater than 9, **DosGetMessage** returns an error indicating that *IvCount* is out of range. If the numeric value of *x* in the *%x* sequence for *%1–%9* is less than or equal to **IvCount**, text insertion, by substituting for *%x*, is performed for all occurrences of *%x* in the message. Otherwise, text insertion is ignored and the *%x* sequence returns in the message unchanged. Text insertion is performed for all text strings defined by *IvCount* and *IvTable*.

Variable data insertion does not depend on blank character delimiters, nor are blanks inserted automatically. For warning and error messages, the message identifier (seven alphanumeric characters consisting of a three-character component identifier followed by a four-digit message number) followed by a colon and a blank character is returned to the caller as part of the message text. (**DosGetMessage** determines the type of message based on the message classification generated in the output file from the **mkmsgf** utility.)

DosGetMessage retrieves messages previously prepared by the **mkmsgf** utility to create a message file, or by **mkmsgseg** to bind a message segment to an *.exe* file. First, **DosGetMessage** attempts to retrieve the message from RAM in the message segment bound to the *.exe* program. If it cannot find this message, **DosGetMessage** retrieves the message from its message file on DASD (Direct Access Storage Device, such as a floppy disk or hard disk).

If no drive or path is specified in the filename, **DosGetMessage** uses the following scenario for the default drive and path:

- No drive or path specified

  The system root directory is used as the default. If this fails, the default drive and current directory are used.

- No drive specified, path specified

  The system boot drive is used as the default. If this fails, the default drive (if different) is used.

- Drive specified, no path specified

  The current directory is used.

Note

The message files should be in the system root directory; however, the user may choose to place them in another directory and use the **append** command to access them.

If **DosGetMessage** cannot retrieve a message because of a DASD hard error or a "file not found" condition, an error message will be returned.

The residency of the message in RAM (*.exe* bound) or on DASD is transparent to the caller and is handled by **DosGetMessage**. In either case, the message is referenced by message number and filename.

12.2.2 DosInsMessage

Purpose:

The **DosInsMessage** call inserts variable text string information into the body of a message.

Calling Sequence:

```
EXTRN DOSINSMESSAGE:FAR

PUSH@ OTHER IvTable      ; Table of variables to insert
PUSH  WORD  IvCount      ; Number of variables
PUSH  WORD  MsgInLength  ; Length of input message
PUSH@ OTHER DataArea     ; Buffer address to return message
PUSH  WORD  DataLength   ; Length of buffer
PUSH@ OTHERMsgInput      ; Address of input message
PUSH@ WORD  MsgLength    ; Length of returned message
CALL  DOSINSMESSAGE
```

where:

IvTable is a pointer table. Each double-word far pointer points to an ASCIIZ or null-terminated DBCS string (variable insertion text). Zero to nine strings can be present.

IvCount is in the range from 0 to 9, the count of variable insertion text strings. If *IvCount* is zero, the next value on the stack, *IvTable*, is ignored.

MsgInput is the address of the input message.

MsgInLength is the length of the input message.

DataArea is the storage area to which the system returns the updated message. If the message is too long to fit in the caller's buffer, as much of the message text as possible will be returned along with the appropriate error code.

DataLength is the length, in bytes, of the user's storage area.

MsgLength is the length of the message returned.

Returns:

IF (AX = 0)
    No error.

ELSE
    AX = Error code:

- Message file not found.
- Message text too long for buffer.
- Message number not found.

Comments:

The **DosInsMessage** utility differs from **DosGetMessage** in that it does not retrieve a message. It is useful when messages are loaded early before the actual insertion text strings are known.

If *IvCount* is greater than 9, **DosInsMessage** will return an error indicating that *IvCount* is out of range. A default message is also placed in the caller's buffer. If the numeric value of *x* in the *%x* sequence for *%1–%9*, is less than or equal to *IvCount*, text insertion, by substituting for *%x*, is performed for all occurrences of *%x* in the message. Otherwise, text insertion is ignored and the *%x* sequence is returned in the message unchanged. Text insertion is performed for all text strings defined by *IvCount* and *IvTable*.

Variable data insertion does not depend on blank character delimiters, nor are blanks inserted automatically.

12.2.3 DosPutMessage

Purpose:

The **DosPutMessage** utility outputs the message (in a buffer passed by a caller) to the specified handle.

Calling Sequence:

```
EXTRN DOSPUTMESSAGE:FAR

PUSH  WORD FileHandle          ; Handle of output file/device
PUSH  WORD MessageLength       ; Length of message buffer
PUSH@ OTHER MessageBuffer      ; Message buffer
CALL  DOSPUTMESSAGE
```

where:

MessageLength is the length of the message to be output.

MessageBuffer is the buffer that contains the message to be output.

Returns:

IF (AX = 0)
    No error.

ELSE
    AX = Error code:

- Invalid handle.

Comments:

The screen width is assumed to be 80 characters. But because this utility formats the buffer to prevent words from wrapping when displayed on the screen, if a word is about to span column 80, it is wrapped around to a new line at column 1. **DosPutMessage** assumes that the starting cursor position is column 1 when handling word wrapping. If the last character to be positioned on a line is a double-byte character that would be

bisected, this wrapping rule ensures that the character is not bisected.

Chapter 13
Mouse Services

13.1 Introduction

This chapter is the reference for all of the MS OS/2 mouse services function calls, which are summarized as follows.

13.1.1 Real-mode INT33 Mouse API Call Summary

The software INT 33H interface is provided only for real-mode support. The INT 33H Mouse API provides the following list of supported functions:

Function	Description
0	Mouse Installed Flag and Reset
1	Show Mouse Pointer
2	Hide Mouse Pointer
3	Get Mouse Pointer Position and Button Status
4	Set Mouse Pointer Position
5	Get Button Press Information
6	Get Button Release Information
7	Set Minimum and Maximum Horizontal Position
8	Set Minimum and Maximum Vertical Position
9	Set Graphics Pointer Shape
10	Set Text Pointer Shape
11	Read Mouse Motion Counters
12	Set User-Defined Subroutine Input Mask
13	Light Pen Emulation Mode ON
14	Light Pen Emulation Mode OFF
15	Set Mickey/Pixel Ratio
16	Conditional OFF
19	Set Double Speed Threshold

The function number and the function-specific parameters are passed to the mouse device driver in the four general purpose registers and the SI, DI

and ES registers:

- The AX register is always used to contain the requested function number.
- The BX, CX, and DX registers are used as needed for function-specific input parameters.
- SI and DI are used for function call 16.
- ES is used for function calls 9 and 12.

Upon returning from a software interrupt 33H function call, the general purpose registers contain return codes and/or mouse requested data items. The registers used are function-specific and detailed under each call description.

All input parameters are checked on function calls requiring parameters.

If an INT 33H function call is issued when the Mouse hardware/software is not properly initialized, the call will return an error code of 0 in AX.

The INT 33H Interface is based on the concept of a virtual display screen coordinates. All coordinates are input/output relative to a predefined range for rows and columns. The ranges are display-mode dependent. For a description of the virtual screen resolution by mode, see the section "Mouse - Virtual Screen Resolutions".

The mouse device driver maps the physical display resolution to the coordinate system of the virtual display screen independently of the physical display mode.

MS OS/2 supports a subset of the Microsoft PC/DOS INT33 mouse API. Display mode support is limited to CGA-compatible modes 0 through 6 and to monochrome mode 7.

The Microsoft INT33 mouse API is available only to those applications executing in real mode. Protected-mode applications must use the **Mou***xxx* mouse device interface.

The mouse device driver provides real-mode applications with an INT 33H interface to the pointing device hardware. The real-mode support is not equivalent to the MS OS/2 protected-mode support but, instead, preserves the Microsoft INT 33H mouse interface.

The real-mode mouse does not support:

- Device handles
- Monitor chains
- IOCtl direct function access
- MOU API function calls

All real-mode mouse functions are accessed via the software INT 33H interface. When a software interrupt 33H is detected, a real-mode mouse handler routine is invoked. All function-relevant information is supplied by the caller in the following seven registers: AX, BX, CX, DX, SI, ES, DI.

There are two states for the real-mode mouse support:

The MS OS/2 mouse device driver is loaded into the operating system. When this state is active, all real-mode mouse functions listed later in this section are available for user support.

The MS OS/2 mouse device driver attempted to load and either aborted due to some critical error encountered at the time the system was initialized, or the mouse device driver processed a deinstallation request. In this state, the real-mode mouse support is limited to INT33H, Function call 0.

13.1.2 Protected-mode Mouse API Call Summary

Call	Description
MouClose	Close Mouse Device
MouDeRegister	Deregister Mouse Subsystem
MouDrawPtr	Unmark Collision Area
MouFlushQue	Flush Mouse Event Queue
MouGetDevStatus	Get Mouse Status
MouGetEventMask	Get Current Event Mask
MouGetHotKey	Get System Hot Key Mouse Button
MouGetNumButtons	Get Number of Mouse Buttons
MouGetNumMickeys	Get Number of Mickeys-Per-Centimeter

MouGetNumQueEl	Get Current Number of Mouse Queue Elements
MouGetPtrPos	Get Mouse Pointer Position
MouGetPtrShape	Get Mouse Pointer Shape
MouGetScaleFact	Get Scale Factor
MouInitReal	Initialize Real-Mode Mouse Driver
MouOpen	Open Mouse Device
MouReadEventQue	Read Mouse Event Queue
MouRegister	Register Mouse Subsystem
MouRemovePtr	Mark Pointer Collision Area
MouSetEventMask	Set New Event Mask
MouSetHotKey	Set System Hot Key Mouse Button
MouSetPtrPos	Set Mouse Pointer Position
MouSetPtrShape	Set Mouse Pointer Shape
MouSetScaleFace	Set Scale Factors

13.1.3 Mouse Device Driver Interface Requirements

A new optional *config.sys* keyword in the statement **device**=*mouse.sys* will inform the mouse device driver whether mouse support is requested for protected mode, real mode, or both. The **mode**=*keyword* statement has the following format:

mode = P|R|B

In this statement, **mode** has one of the following values specifying the type of support:

Mode	Meaning
P	Protected-mode-only
R	Real-mode-only
B	Both

The case of the letters is insignificant.

The default type is Both. If the **mode**= *keyword* statement is omitted from the **device**=*mouse.sys* statement in the **config.sys**file, the system will support mice in both real and protected modes.

13.1.4 Mouse Device Driver Button Definitions

The real-mode button definition depends upon the number of buttons on the mouse device.

Button definitions are used in functions 3, 5, 6, and 12. Button number assignments are as follows:

Two-button mouse:

Bit	Mouse
1	Rightmost button
0	Leftmost button

Three-button mouse:

Bit	Mouse
2	Center button
1	Rightmost button
0	Leftmost button

Chapter 14
National Language Support

14.1 Introduction

National language support for this version of MS OS/2 includes these major features:

- National language support for system message files
- Country-dependent information
- Support for national keyboard layouts
- Double-byte character set (DBCS) enabling
- Programming interfaces for national language support
- Message Retriever for message files
- Utility commands

For languages that read from left-to-right, national language support is provided for displaying system messages to the user.

Country-dependent information is available on a per-country basis and includes the following:

- Time, date, and currency
- Lower- to uppercase character conversion tables
- Collating sequence for character sorting
- DBCS environmental vector for DBCS character determination
- Valid single-byte characters used in filenames

Keyboard support for different keyboard layouts is provided and selectable.

DBCS enabling supports DBCS-based national languages. The *Microsoft Operating System/2 Programmer's Reference* identifies the kernel functions for DBCS enabling of mixed single- and double-byte character strings. Full DBCS implementation and DBCS-based hardware support is not provided in this version of MS OS/2.

The MS OS/2 set of programming interfaces for national language support allow applications to use the country-dependent information just described. To access this information, applications do not need to change the current country code of the system. The current country code for the

system is the same for all screen groups and for both real and protected mode.

The Message Retriever utility provides a programming interface for user applications. This interface allows you to retrieve and display application messages from your customized message files.

Utility commands allow the user to select the keyboard layout and system country code.

This version of MS OS/2 does not support right-to-left national languages. Nor does it support multiple code-page fonts and code-page switching.

14.2 National Language Support Calls

The following dynamic-link calls allow an application, running in protected mode, to tailor its operation to the current country code or to change the current country code. The use of this information and the term *country code* are described elsewhere in this chapter.

These programming interfaces are also used by MS OS/2 to support the National Language requirements. The following programming interfaces are part of the DOS Family API:

DosCaseMap	Perform Case Mapping
DosGetCtryInfo	Get Country-Dependent Information
DosGetDBCSEv	Get DBCS Environmental Vector

These functions support access to country-dependent information. All country-dependent information is resident in one file named *country.sys*.

Chapter 15
Program Startup

15.1 Introduction

The rules controlling the content of various data areas and registers on program entry and termination are different for old programs than for new programs.

For old programs, the conventions of MS-DOS 3.*x* apply.

For programs linked with MS OS/2, the new conventions are as follows:

- There is no PSP.
- The argument list and environment list are passed in a segment (selector in AX).

The following registers will be set on program entry:

CS:IP	Points to the program's initial entry point as specified in the *.exe* header
SS:SP	Points to the stack specified in the *.exe* header
DS	Points to the automatic data segment specified in the *.exe* header
ES	0
AX	Environment segment handle
BX	Offset of the command line in the environment segment
CX	Size of the automatic data segment (0 = 65,536)
BP	0

15.2 Program Startup Calls

The two functions described in this chapter are designed specifically for program startup. These functions allow an application to tailor its operation to specific versions of MS OS/2.

DosGetEnv	Get Address of Process Environment String

DosGetVersion Get MS OS/2 Version Number

Chapter 16
Signals

16.1 Introduction

Signals are a mechanism that allows a process to intercept and deal with a variety of traps and external events. The signal facility allows a program to specify an "on condition" handler routine that is executed when the event occurs.

Following are two examples of events that can cause signal handlers to be executed:

- Pressing the CONTROL-BREAK key combination
- Pressing CONTROL-C
- Terminating a program via **DosKill**
- **DosFlagProcess** invoked

Each process may define a process-unique signal handler for any signal.

An incoming signal is handled in one of several ways:

- By the default action: this is typically IGNORE or TERMINATE PROCESS.
- If the process has specified a signal-handling address, Thread 1 (the original task thread) is diverted, in a "forced far-call" (analogous to a hardware interrupt), to the proper signal-handler address. Since an incoming signal represents a time-critical event, if Thread 1 is in the midst of a system call that will not complete quickly, that call will be aborted.

  The signal "interrupt" will take place immediately upon return from the MS OS/2 service call. "Slow" calls that may be so aborted are primarily device I/O calls. File system calls (disk open/close-/read/write) are not normally aborted.

  An application expecting to make non-emergency use of signals should reserve Thread 1—perhaps by having it block upon an externally-reserved RAM semaphore—and use another thread for program execution.

The two functions provided for signaling are as follows:

DosHoldSignal	Disable/Enable Signals
DosSetSigHandler	Define Routine to Handle Signal

Chapter 17
Tasking Calls

17.1 Introduction

In its simplest form, multitasking allows a user to operate several applications concurrently, with each application appearing to have the entire computer to itself. These applications are designed and coded in a somewhat different manner than they would be under MS-DOS.

Alternatively, an application may be designed so that its functions are divided among a collection of cooperating processes or performed by several execution threads within a single process.

Since applications dispatch these execution threads on a priority basis, MS OS/2 includes a scheduler, which ensures that threads of equal priority receive equal opportunity to execute. MS OS/2 also provides special functions for applications that require time-critical responses or that require multitasking. These functions

- initiate and terminate other processes and threads
- vary the dispatch priority of a process
- execute programs as separate processes
- coordinate execution among several processes
- communicate between several processes

This chapter describes those functions that are provided to vary process priority, to initiate and terminate processes, and to execute other programs as separate processes.

17.1.1 Definitions

A process represents an instance of an executing program together with the resources the program is using. Only a process may own resources, a thread may only access resources on behalf of its process. The thread is the unit of execution within a process and is not identifiable outside its process.

Where necessary in this chapter, you'll see terms such as "current thread," "waiting thread," or "parent thread." These terms are necessary to make perfectly clear that what is being discussed is one (of possibly many) thread of a process rather than the process itself.

To solve a particular problem, an application designer may effect multitasking by either multiple processes or multiple threads, depending on the problem.

- Multiple processes would be the proper choice, if the problem can be best solved by multitasking elements with a high degree of independence.

  Several independent processes may also be the best choice for applications in which the independence or function of the process model is required, yet the various programs of the application are tightly coupled with respect to interprocess communication or fixed, shared data areas.

- Multiple threads would be the better choice if the concurrent execution elements are to be started for only a short period of time. The reason for this choice is that the overhead required to start and end a thread is much less than it would be for a process.

  A thread would also be more suitable for problems in which multiple execution instances are needed, yet where each instance need not be externally identifiable and does not require resources distinct or separate from other instances.

17.1.2 Tasking Calls Summary

The tasking functions provided are

DosCreateThread	Create Another Thread of Execution
DosCWait	Wait for Child Process Termination
DosEnterCritSec	Enter Critical Section of Execution
DosExecPgm	Execute Program
DosExit	Exit Current Thread
DosExitCritSec	Exit Critical Section of Execution
DosExitList	Routine List for Process Termination
DosGetInfoSeg	Get Address of System Variables Segment
DosGetPrty	Get Process (or Thread) Priority
DosKillProcess	Terminate Process

DosPTrace	Interface for Program Debugging
DosResumeThread	Restart Thread
DosSetPrty	Set Process (or Thread) Priority
DosSuspendThread	Suspend Thread Execution

17.1.3 Resource Management

MS OS/2 provides resource management and tracks ownership at the process level. The resources that a process may own, or be granted controlled access to, are as follows:

- Files (and devices, if opened at the process level)
- Memory
- Pipes
- Queues
- System semaphores

When a process terminates, any resource that does not explicitly close or release is released automatically.

Chapter 18
Timer Services: Time/Date and Intervals

18.1 Introduction

The functions in this chapter are related to time of day and intervals of time. They provide the full range of "time primitives" necessary to use applications.

These timing functions are only accurate to within one or two clock ticks (at best), so an application may perceive that the time elapsed is longer than what was specified because of delays in getting the application resumed. All time values are in milliseconds and are rounded up to the next clock tick. The duration of the clock tick can be determined by using the **DosGetInfoSeg** function and examining the timer interval field in the GDT segment.

If it is necessary for an application to know how much time elapsed while waiting on a timer, a comparison can be made of the milliseconds field (in the GDT info segment) before the wait began and after the application resumed execution. The value in the milliseconds field will roll over every few weeks. If the application suspended prior to a rollover and awoke after a rollover then the elapsed time calculation must take that into account.

Under situations where interrupts are disabled for periods longer than the clock-tick interval, the milliseconds field may lose time. However, the time of day (hours, minutes, seconds), time in seconds since 1–1–70, and the date will remain accurate. If situations arise where interrupts may be disabled for long periods, the elapsed time should be computed in seconds—rather than milliseconds.

18.2 Time/Date/Interval Calls

MS OS/2 provides the following calls related to the time-of-day:

DosGetDateTime	Get Current Date and Time
DosSetDateTime	Set Date and Time

The following calls are provided for interval-related functions:

DosSleep	Delay Process Execution
DosTimerAsync	Start Asynchronous Time Delay
DosTimerStart	Start Periodic Interval Timer
DosTimerStop	Stop Interval or Asynchronous Timer

Chapter 19
Programming Hints

19.1 Introduction

This chapter describes recommended programming guidelines, procedures, and restrictions. These hints can help you write programs that will run in the MS OS/2 environment. In addition, by using these hints, you can ensure compatibility with future versions of MS OS/2.

The programming hints are organized in the following categories, with their section numbers in parentheses:

- 80286 Compatibility and Conformance (Section 19.2)
- Family API Considerations (Section 19.2.2)
- Compatibility-mode Considerations (Section 19.3). This category includes the following subcategories, all 3.*x* box only:

  System calls

  Interrupts

  Device management

  Memory management

  Process management
- File and Directory Management (Section 19.4)
- Miscellaneous (Section 19.5)

19.1.1 Old Programs

Old programs are defined as those programs created with MS-DOS 3.*x* (or earlier) linker. They are also called 3.*x*-box programs, or *compatibility-mode* programs. The following rules pertain to old programs in the MS OS/2 environment:

- Run in compatibility-mode
- Cannot execute in the background
- Must use the old-style INT 21H DOS interface
- Do not rely on undocumented MS-DOS interfaces

- Have I/O Privilege Level (IOPL)

19.1.2 Protected-mode Programs

Protected-mode programs are those programs linked with **link under MS OS/2. They are always loaded as protected-mode applications in systems capable of supporting protected-mode operation. The following rules apply to protected-mode programs:**

- **Cannot use the old-style INT 21H DOS interface**
- **Must use the new MS OS/2 API functions**
- **Can overcommit physical memory (data and code) through use of MS OS/2 memory management functions**
- **Cannot issue software interrupts**
- **Cannot process hardware interrupts**
- **Always reside above the protected-mode line (in a mixed-mode system)**
- **Must obey Intel 80286 segment manipulation rules**
- **Do not have IOPL without special requests to MS OS/2**

19.2 80286 Compatibility and Conformance

This section discusses MS OS/2 compatibility with applications from MS-DOS versions 3.*x*, 2.*x*, and 1.*x*. It also describes the differences between the 80286 and 8086/8088 processors, and addresses any special considerations that need to be made for code to run in the compatibility mode. Finally, it summarizes the program rules applicable to each class of MS OS/2 application program.

19.2.1 Differences Between the Processors

The Intel 80286 (286) processor is the processor in the IBM Personal Computer AT, and is the next processor in the 8086/8088 processor family. The 80286 processor differs from the 8086 and 8088 processors in the following ways:

- The 80286 includes extra instructions, such as "shift by count" and "push immediate."
- The 80286 supports both real mode and protected mode.
- It is much faster—approximately three times faster—than the 8088 in the IBM Personal Computer XT.

The real mode is intended to be compatible with the 8086 processor, but it does not *exactly* emulate an 8086 or 8088—there are some small but meaningful differences. So when stating that 8086 programs must be binary-compatible with the 80286, the question raised is, "compatible with the 80286 in real mode or in protected mode?" Ideally, the stronger condition, "compatibility with protected mode," should be met. When this condition is met, programs can take advantage of the flexibility and power provided by the MS OS/2 memory-management and dynamic-linking features.

19.2.2 Family API Considerations

Applications that use the Family API functions (so that they may execute in protected mode under MS OS/2 as well as in real mode under MS-DOS 3.x) must also consider the differences in the instruction sets of the 80286 and 8088/8086 processors. Some instructions that operate on the 80286 are undefined, and cannot be emulated, on the 8088. Specific guidelines apply to these instructions in applications that need to run on both machines.

To stay within these guidelines, an application can use one of two methods. First (and simplest) is the *avoidance* method—simply using the set of instructions common to both machines. Second is the test method, in which the program checks the processor type before an 80286 instruction is executed; the instruction is executed on an 80286 and emulated (by the application program) on an 8086/8088.

19.2.3 80286-8088 Compatibility Rules

Following is the list of rules to follow for 80286- and 8088-compatible operation. To run in protected mode, certain practices common to real-mode-only programs must be avoided.

- Do not depend on any segment overlap (or lack thereof).

  When a program loads a segment number into a segment register on an 8086, it is actually loading a displacement value. When a program loads a segment number into a segment register on the

80286, the 80286 uses the number to locate an entry in a master segment table, and then loads the displacement value for the segment from that table. On the 8086, a segment register contains the displacement; on the 80286, it contains an index into a table that contains the true displacement. Therefore, you cannot depend on segment overlap.

- Do not depend on any relationship between segment:offset combinations and physical memory.

  On the 8086, the segment value represents the high-order bits of the physical memory address. On the 286, however, one segment bears no particular relationship to the next; in fact, they may be far apart in physical memory.

- Place only valid segment numbers in segment registers. Do not use segment registers for scratch space.

  An 8086/286-compatible program must be written following a *true-segment* architecture. This means that the program makes no assumptions about the relationship between one segment and another, or between a segment and memory. It should only get its segment values from the linker, from the DOS operating system, or perhaps from its callers; it must never create or invent a segment number on its own. This means that segment registers cannot be used to hold arbitrary temporary values; they may hold only valid segment numbers.

- Never address beyond the allocated size of a segment.

  In protected mode, the master segment table entry (called the *segment descriptor*) contains a limit field. Any offset larger than this limit that is used to reference that segment generates a machine trap and that aborts the program.

  A program running in real mode may or may not "crash" the system, but it will always crash in protected mode because the 80286 protection hardware protects the system and other programs from the illegal reference. Furthermore, a segment offset in protected mode is limited to 65K, so adjusting the segment number merely points to a new segment instead of pointing further into the existing one. This means that if the program needs 120K, it must request it as two or more segments.

- Never mix code and data in one segment, and do not attempt to modify the contents of a code segment.

A restriction built into the 80286 does not allow writable code segments. One of the bits in the 286 segment descriptor indicates a CODE or DATA segment. Although the DS and ES registers may contain the selector of a CODE segment, the CS register may only contain selectors of CODE segments.

While a DATA segment's descriptor has a bit to indicate read/write or read-only access, a CODE segment's descriptor does not. As a consequence, only valid CODE segments may be placed in the CS register, and a program may not write into valid CODE segments, even if the segment selector is copied from the CS register to the DS or ES register.

- If you must manipulate I/O ports, use the appropriate dynamic-link routines.

- Do not use CLI.

  An 80286-compatible program cannot issue the CLI instruction because it causes a protection trap. An IRET instruction restores the previous value of the interrupt flag in real mode, but has no effect on the IFLG in protected mode. A software INT instruction does not disable interrupts in protected mode; it does in real mode, however.

- Do not use wrap-around segment offsets.

  The 8086 generates physical memory addresses by adding the offset value to (16 * segment-value), yielding a 20-bit result. Any overflow is ignored, so it is possible to wrap-around to low memory by using sufficiently large segment and offset values. For example, on a true 8086, the address FFF0:20 references the same location as the address 0000:10. The 80286, however, generates a physical address of 10000:10, or one megabyte plus 16. This is possible because the 80286 has 24 address lines, whereas the 8086 has only 20.

- Do not use the PUSH SP instruction.

  When executing the PUSH SP instruction, the 80286 pushes the SP value *before* the push instruction, while the 8086 pushes the value that SP will be *after* the push. This is an unusual sequence that does not appear in most code. If it should occur, you should use the following sequence:

```
MOV             AX,SP
PUSH            AX
```

- Do not use shift counts greater than 31.

  Shifts and rotate counts in the CL register are masked to five bits on the 80286; they remain eight bits wide on the 8086. Since the masked bits are all multiples of 16, most shifts and rotates with large counts produce the same result with a masked-off shift count. The exceptions are the rotate-with-carry instructions that involve 17 bits and can produce different results with shift counts greater than 31. Since larger shift counts waste CPU time, they should be avoided.

- Do not use IDIV operands that produce the most negative number.

  The 80286 can generate the "most negative number" as a quotient for the IDIV instruction. The 8086 generates a divide error exception. (This occurs for quotients of 8000H (WORD) and 080H (BYTE))

- Divide-trap handlers should not resume execution in the original code stream or they will have to detect and understand 80286/8086 differences.

  After a divide error trap, the 80286 will point at the divide instruction, including prefixes. The registers will be unchanged. The 8086 will point *after* the divide instruction and may change DX:AX or AH:AL, as appropriate.

- Do not use repeated (redundant) prefix bytes.

  The 80286 has an instruction-length limit of 10 bytes. This is longer than any possible instruction unless the prefix bytes are repeated.

- Never use undefined opcodes.

- Do not depend upon CPU speeds for any purpose.

  Various machines use CPUs of different speeds, so you should not rely upon the speed of the CPU for timing.

- Never examine or set explicit flag-register values; set and test the values only by using flag-specific instructions.

  The instruction PUSHF followed by POPF may change the contents of the flag register, since more flag bits are defined in the 80286 flag word. Programs should set and test flags using the standard instructions, not set or examine specific bit patterns via LAHF or SAHF.

- Do not single-step an INT instruction.

  The 8086 and 80286 processors work differently.

- Do not use POPF. Use the suggested POPFF macro.

  The POPF instruction does not work properly on existing 80286 processors. Specifically, if interrupts are off and a POPF is used to restore a flag word that also has interrupts off, an interrupt *may* be granted anyway. The POPF instruction should be replaced by the macro POPFF, defined as follows:

```
POPFF           MACRO
                LOCAL   A
                JMP     $+3
A               LABEL   NEAR
                IRET
                PUSH    CS
                CALL    A
                ENDM
```

  This macro uses PUSH CS and a short call to set up the stack so that an IRET can be used to restore the flags.

19.2.4 Hardware Interface

Direct interaction with the hardware is allowed in protected mode if a program is linked to run the IOPL segment. Normal segments in protected mode do not run at IOPL (I/O Privilege Level).

Direct access to the hardware cannot be prevented in real mode.

19.2.5 IBM PC/AT BIOS Interfaces

Several BIOS interfaces (using INT 15H) are not allowed under MS OS/2:

- Move block—high memory (87)

  Move a block of memory between low (640 kilobytes) and high (> 1 megabyte). Because this function interferes severely with normal operation of MS OS/2, it is not allowed.

- Processor to virtual mode (89)

  The swapping between protected and real modes is managed by MS OS/2. This function is not available to application programs.

- Multitasking (90 and 91)

  This function is provided by the operating system. Application programs (or subsystems) are not allowed to use their own variety of multitasking, since it is already provided by MS OS/2.

19.3 Compatibility-mode Programming Hints

Programs that must run in compatibility-mode under MS OS/2 must conform to certain programming conventions and limitations. The following section describes these conventions as they relate to the following subjects:

- Definition of compatibility-mode
- Compatibility-mode exceptions
- System calls
- Interrupts
- Device management
- Memory management
- Process management

19.3.1 Compatibility Mode: A Definition

The MS-DOS 3.2 Compatibility-mode of MS OS/2 is as follows:

- The compatibility-mode environment is that of MS-DOS 3.2 with the **share** command installed.
- Documented MS OS/2 services exist for compatibility mode.
- A ROM data area exists in compatibility mode.
- ROM services interrupts are accepted.
- Hardware interrupts (except for a real-time clock) are accepted.
- INT 28H (Spooler Interrupt) will be issued.

- The 3.*x* box is frozen in the background (can lose interrupts for the 3.*x* box).
- A *small* list of existing device drivers is supported.

  The only supported character device drivers are DI and old CON drivers. No old block device drivers are supported.
- No networking support is available in compatibility mode.
- No direct calls to ROM services (must use INT 10–1FH) are allowed.
- No calls are allowed to user code from device drivers entered through the kernel.
- No application may move the 8259 interrupt vectors (except TopView).
- INT 24H (Critical Error Handler Address) will never be issued in compatibility mode.

19.3.2 Compatibility-mode Exceptions

As described previously, all functions supported in MS-DOS 3.2 are supported in MS OS/2, with some exceptions, as follows:

- File sharing is always in effect.

  Restrictions imposed on applications in MS-DOS 3.2 by the **share** system utility are *always* in effect.
- Undocumented MS-DOS 3.2 function calls are not supported.
- MS-DOS 3.2 Network function calls are not supported
- The MS OS/2 version number is 10.0; version-bound applications will not run.
- The real-mode environment is frozen (suspended) while the real-mode application is in a background screen group. The process gets no task-time CPU service, nor does it receive any interrupts. An application that tracks the time of day by counting clock ticks will have an incorrect count when it returns to the foreground.

- Real-mode applications may "hook" any hardware interrupt vectors, except in the following cases:
  - — When the interrupt is the CMOS clock interrupt
  - — When the interrupt is already owned by an MS OS/2 device driver other than the keyboard interrupt

  If a real-mode application hooks a hardware vector that falls in one of these two categories, the real-mode application will be terminated.

- There are restrictions on which devices real-mode applications may directly manipulate:

  - — 8253 clock/timer chip

    Real-mode applications may reprogram this device at will.

  - — 8259 interrupt controller

    Real-mode applications may not reprogram the 8259 interrupt controller.

  - — Disk controller

    Real-mode applications may not reprogram the disk controller(s), although they may have direct access via INT 13H, INT 25H and INT 26H. INT 13H and INT 26H are not allowed for nonremovable media.

  - — DMA controller(s)

    Real-mode applications may not reprogram the Dynamic Memory Allocation (DMA) ports.

  - — COM/AUX/parallel port

    Real-mode applications may reprogram these devices at will.

    Using one of these ports while running an application in the compatibility mode makes the port unavailable to protected-mode processes. Similarly, if a protected-mode process uses one of these ports, it is unavailable to application running in compatibility mode.

- Not all old MS-DOS device drivers will run in the compatibility box. For more information, see *Microsoft Operating System/2 Device Drivers Guide.*

19.3.3 System Calls (Compatibility Mode Only)

- Do not use superseded system calls.

  Avoid using system calls that have been superseded by new calls, unless the program *must* maintain backward compatibility with versions of MS-DOS before 2.0.

- Avoid using system calls **01H–0CH** and **26H** (Create New PSP).

  Use the "tools" approach for reading and writing on standard input and output. Use **Function 4B00H** (Load and Execute Program) instead of **Function 26H** to execute a child process. Use file-sharing calls if more than one process is in effect.

- Use networking calls where appropriate.

  Some forms of IOCtl can be used only with Microsoft Networks.

- When selecting a disk with **Function 0EH** (Select Disk), treat the value returned in the AL register with care.

  The value in the AL register specifies the maximum number of logical drives; it does not specify which drives are valid.

19.3.4 Interrupts (3.*x* Box Only)

- Never explicitly issue INT 22H (Terminate Process Exit Address).

  Only the operating system should issue this interrupt. To change the terminate address, use **Function 35H** (Get Interrupt Vector) to get the current address and save it, then use **Function 25H** (Set Interrupt Vector) to change the INT 22H entry in the vector table to point to the new terminate address. Use INT 24H (Critical Error Handler Address) with care, since it must preserve the ES register.

- An INT 24H handler can issue only the **01H–0CH** system calls.

  Making any other calls destroys the MS-DOS stack and prevents successful use of the *Retry* or *Ignore* options.

- The SS, SP, DS, BX, CX, and DX registers must be preserved when using the *Retry* or *Ignore* options.

  When an INT 24H (Critical Error Handler Address) is received, always return using IRET to MS-DOS with one of the standard

responses. Programs that do not return with IRET from INT 24H leave the system in an unpredictable state until a function call other than **01H–0CH** is made. The *Ignore* option may leave incorrect or invalid data in internal system buffers.

- Avoid trapping INT 23H (CONTROL-C Handler Address) and INT 24H (Critical Error Handler Address). Do not rely on trapping errors via INT 24H as part of a copy protection scheme.

  These methods may not be supported in future operating system releases.

- A user program must never issue INT 23H (CONTROL-C Handler Address).

  Only the operating system may issue INT 23H.

- Save any registers that your program uses before issuing INT 25H (Absolute Disk Read) or INT 26H (Absolute Disk Write).

  These interrupts destroy all registers except for the segment registers.

- Avoid writing or reading an interrupt vector directly to or from memory.

  Use **Functions 25H** and **35H** (Set Interrupt Vector and Get Interrupt Vector) to set and get values in the interrupt table.

19.3.5 Device Management (3.*x* Box Only)

- Use installable device drivers.

  MS-DOS provides a modular device driver structure for the BIOS, allowing you to configure and install device drivers at boot time.

- Use buffered I/O.

  The device drivers can handle streams of data up to 64K bytes in length. To improve performance when sending a large amount of output to the screen, you can send the output with one system call.

- Programs that use direct console I/O via **Function 06H** and **07H** (Direct Console I/O and Direct Console Input) and that want to read CONTROL-C as data should ensure that CONTROL-C checking is off.

The program should ensure that CONTROL-C checking is off by using **Function 33H** (CONTROL-C Check).

- Be compatible with international support.

  To provide support for international character sets, MS-DOS recognizes all possible byte values as significant characters in filenames and data streams. Versions of MS-DOS before 2.0 ignored the high bit in MS-DOS filenames.

19.3.6 Memory Management (compatibility mode only)

- Use MS-DOS memory management features.

  The operating system keeps track of allocated memory by writing a memory control block at the beginning of each area of memory. Programs should use **Functions 48H** (Allocate Memory), **49H** (Free Allocated Memory), and **4AH** (Set Block) to release unneeded memory.

  Using this method allows for future compatibility.

- Use only allocated memory.

  Do not directly access memory that was not provided as a result of a system call. Do not use fixed addressing; instead, use only relative references.

  A program that uses memory that has not been allocated to it may destroy other memory control blocks or cause other applications to fail.

19.3.7 Process Management (3.*x* Box Only)

- Use **Function 4B00H** (Load and Execute Program) to load and execute programs.

  Function 4B00H is the preferred call to use when loading programs and program overlays. Using this call instead of hard-coding information about how to load an *.exe* file (or always assuming that your file is a *.com* file) isolates your program from changes in *.exe* file formats and future operating system releases.

- Use **Function 31H** (Keep Process) instead of INT 27H (Terminate But Stay Resident).

  Function 31H allows programs that are larger than 64K to terminate and stay resident.

- Programs should terminate using **Function 4CH** (End Process).

  Programs that terminate by any of the following methods must ensure that the CS register contains the segment address of the PSP (Program Segment Prefix). These methods terminate a program by

  — a long jump to offset 0 in the PSP

  — issuing an INT 20H with CS:0 pointing at the PSP

  — issuing an INT 21H with AH=0, CS:0 pointing at the PSP

  — a long call to location 50H in the PSP with AH=0

19.4 File and Directory Management

- Use the file management system provided by the operating system.

  Using the MS OS/2 file system ensures program compatibility with future versions of MS OS/2 and MS-DOS through compatible disk formats and consistent internal storage.

- Use file handles instead of FCBs.

  A handle is a 16-bit number that MS-DOS returns when a file is opened or created using **Functions 3CH**, **3DH**, **5AH**, or **5BH** (Create Handle, Open Handle, Create Temporary File, or Create New File).

  You should use these calls instead of the old file-related functions that use FCBs (File Control Blocks). This is because a file operation can simply pass its handle rather than maintaining FCB information. If you must use FCBs, be sure the program closes them and does not move them around in memory. Before issuing an Interrupt 20H (Program Terminate), Function 00H (Terminate Program), Function 4CH (End Process), or Function 0DH (Reset Disk), you should close all files that have changed in length. If you do not close a changed file, its length will not be recorded correctly in the directory.

- Close all files when they are no longer needed.

  In a multitasking or networking environment, closing unneeded files increases efficiency.

- Change disks only if all files on the disk are closed.

  Information in internal system buffers may be written incorrectly to a changed disk.

Locking Files

- Programs should not rely on being denied access to a locked region.

  To determine the status of a region, first attempt to lock it, then examine its error code.

- Programs should not close a file with a locked region, or terminate with an open file that contains a locked region.

  The result of this procedure is undefined. Programs that might be terminated by an INT 23H or INT 24H (CONTROL-C Handler Address or Critical Error Handler Address) should trap these interrupts and unlock any locked regions before exiting.

19.5 Miscellaneous

- Avoid timing dependencies.

  Various machines use CPUs of different speeds. Also, in a networking environment, programs that rely upon the speed of the clock for timing are *not* dependable.

- Use the documented interface to the operating system.

  If either the hardware or media change, the operating system can use the features without modification.

- Do not use the ROM support provided by the OEM (Original Equipment Manufacturer).

  If OEM changes or updates the ROM, the operating system can operate without modification.

- Do not directly address the video memory.

  You should always use the operating system interface to access video memory.

- Do not use undocumented system calls, interrupts, or features.

  These items may change or may not exist in future versions of the operating system. If you do use these features, your program will be highly nonportable.

- Use the MS OS/2 *.exe* format rather than the old program *.com* format.

  MS OS/2 *.exe* files are relocatable, but *.com* files are direct memory images that load at a specific place and have no room for additional control information. MS OS/2 *.exe* files also have headers that provide extended information to MS OS/2.

- Use the environment to pass information to and from applications.

  The environment allows a parent process to pass information to a child process, or to a separate process, with ease.

Chapter 20
MS OS/2 Utility Programs

20.1 Introduction

This chapter describes the MS OS/2 application program preparation utilities. The utilities included are

bind	Merges the MS OS/2 *.exe* file and resolves dynamic links for execution in an MS-DOS 3.*x* environment
implib	Creates an import library from a dynamic-link module
link	Links application code and library routines, and is described in the *Microsoft Operating System/2 Codeview and Utilities* manual
mkmsgf	Creates an MS OS/2 message file for use with MS OS/2 message facilities
msgbind	Modifies an MS OS/2 *.exe* file to contain RAM-based messages
helpmsg	Provides help information related to a warning or error message

20.2 The Bind Utility

Bind is an MS OS/2 utility that allows *.exe* files to be created that will run in protected mode or in an MS-DOS 3.*x* environment. **Bind** works by binding together the MS OS/2 *.exe* file with an MS-DOS 2.*x*/3.*x* *.exe* file and resolving the MS OS/2 style dynamic-link entries with true links. There are three elements to this tool:

bind	This tool merges the *.exe* files and resolves dynamic links.
loader	This tool loads the MS OS/2 *.exe* file when running MS-DOS 2.*x*/3.*x* and simulates the MS OS/2 startup conditions in an MS-DOS 3.*x* environment.
api.lib	This library simulates the MS OS/2 Application Program Interface in an MS-DOS 3.*x* environment.

20.2.1 Process Overview

Bind will function on any MS OS/2 *.exe* file. The user invokes **bind** with the MS OS/2 *.exe* file. The output is a new MS OS/2 *.exe* that is MS OS/2 compatible. A default library is used which contains the stub loader and the MS OS/2 API simulation routines. You can specify additional libraries to resolve other dynamic links. **Bind** has three steps:

1. Read in the dynamic-link entry points from the MS OS/2 *.exe* file. Ouput *extern def* records to a temporary *.obj* file for the dynamic-link names.
2. Run the linker using the temporary *.obj* module and *api.lib* to produce an MS-DOS 3.*x* *.exe* file. The *.exe* contains the loader and the API simulator routines in true link form.
3. Merge the MS OS/2 *.exe* file and the MS-DOS 3.*x* *.exe* file into a single file.

20.2.2 Command-line Format

Start the process by issuing the **bind** command. The general form of the **bind** command line is

Syntax:

bind *infile*[*.ext*] [*implib*] [*linklib*] [**–o** *outfile*[*.exe*]] [**–n** *@file*] [**–n** *name*[...]] [**–m** *mapfile*]

where:

Infile is the name of the MS OS/2 *.exe* file. The filename may contain a drive letter and pathname.

Implib is the names of the import libraries to be used in resolving the external references in the MS OS/2 *.exe* file to the entry points in the link-lib *.exe* file. There may be more than one *implib* specified. This file is optional unless IMPORT BY ORDINAL is used with the linker.

Linklib is the names of the bindings (like the MS OS/2 Family API) that are linked to the *.exe* file and loaded when the program runs in an MS OS/2 environment. The MS OS/2 Family API library is always included in the external reference resolution search. There may be more than one *linklib* specified.

–o *outfile* is the name of the bound *.exe* file (default is *infile*). The filename may contain a drive letter and pathname.

–n *name*[...] is the list of names to map to the **BadDynLink** system call.

–n [@*file*] is a file of names to map to the **BadDynLink** system call.

–m *mapfile* causes a link map to be generated for the MS-DOS 3.*x* environment of the *.exe* file. *Mapfile* is the destination of the link map. If no *mapfile* is specified, the destination of the link map is standard output.

The minimum parameter required is the name of the *.exe* file to bind. If no output filename is given, the new version will replace the old one. The user call specifies *.obj* and *.lib* files to be included in the link phase.

The **–n** option allows the declaration of names that will be mapped to the NotPresent API routine. If this routine is ever called at run time, it will abort the program with an error. So it is the responsiblity of the program to do version checks at run time and ensure that this routine is never called. If there are more names to be mapped than will fit on the command line, a file of names may be given via the @ character. This file contains a list of names, one name per line.

20.2.3 *.Exe* File Layout

MS OS/2 *.exe* files have two headers. The first header has an MS-DOS 3.*x* format. The second header has the MS OS/2 format. When the *.exe* file is run on an MS OS/2 system, it ignores the MS-DOS 3.*x* header and uses the MS OS/2 format. When when run under MS-DOS 3.*x*, the old header is used to load the file. Figure 20.1 shows the arrangement of the merged

headers:

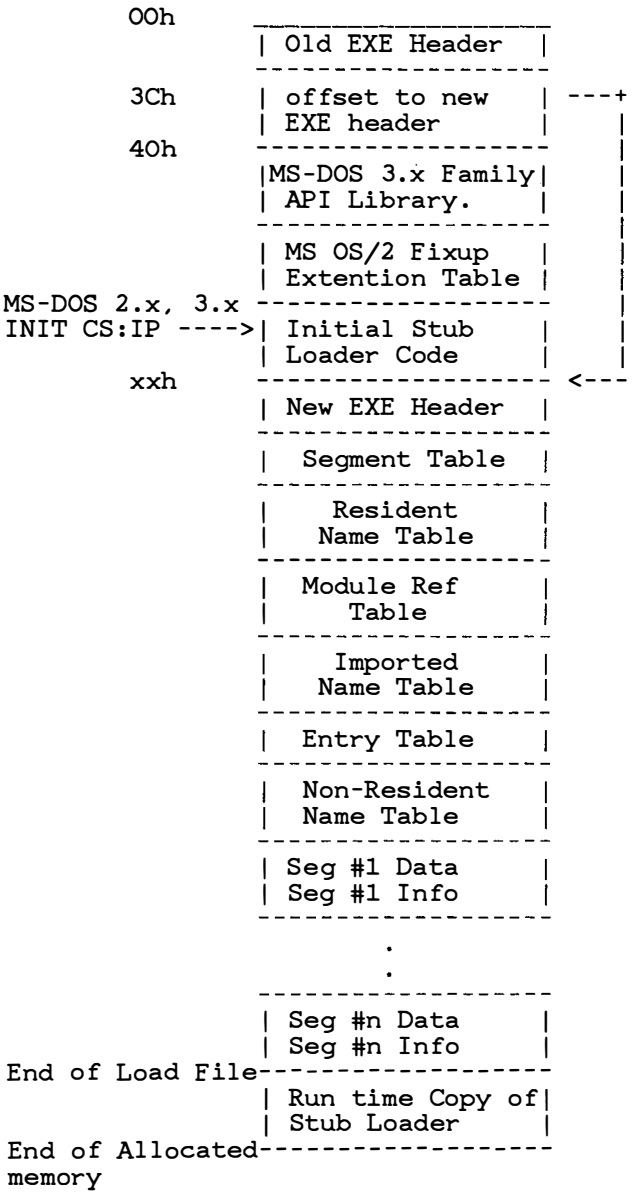
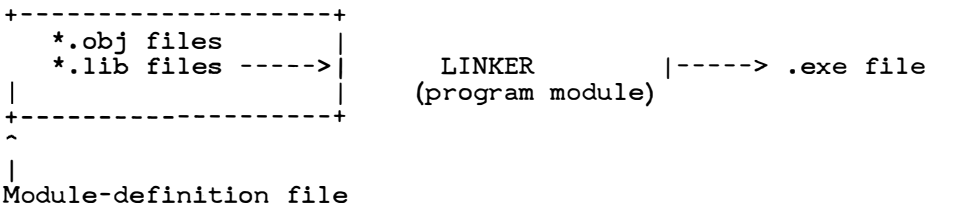

00h
Old EXE Header
3Ch
offset to new EXE header
40h
MS-DOS 3.x Family API Library.
MS OS/2 Fixup Extention Table
MS-DOS 2.x, 3.x
INIT CS:IP ---->
Initial Stub Loader Code
xxh
New EXE Header
Segment Table
Resident Name Table
Module Ref Table
Imported Name Table
Entry Table
Non-Resident Name Table
Seg #1 Data
Seg #1 Info
Seg #n Data
Seg #n Info
End of Load File
Run time Copy of Stub Loader
End of Allocated memory

Figure 20.1 Merged Headers in an MS OS/2 and MS-DOS 3.*x* file

20.3 The Implib Utility

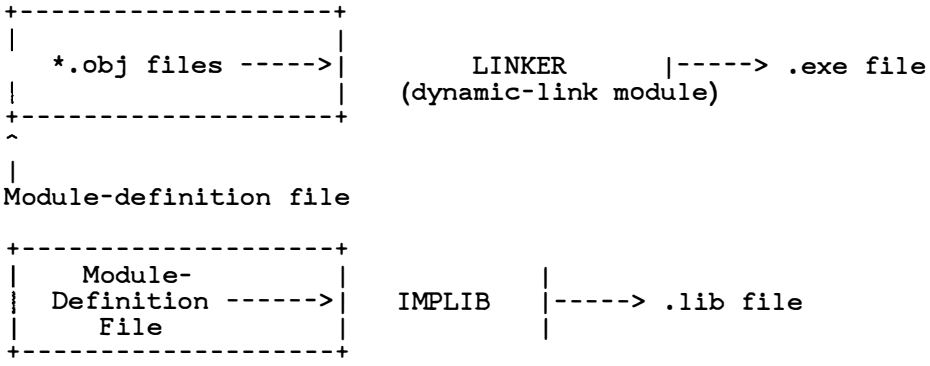
Implib is a utility program for creating dynamic-link modules that can be used by other applications developers. This section describes how to link applications and dynamic-link routines. You use the MS OS/2 utility **link** to link either an application or a group of dynamic-link routines. Whether an application or a group of dynamic-link routines is being linked, the output is an *.exe* file. If an application is being linked, the resulting *.exe* file is called a *program module.* If a group of dynamic-link routines is being linked, the resulting *.exe* file is called a *dynamic-link module.*

You create MS OS/2 program modules by linking your compiled source files with the **link** utility. **Link** takes your compiled source files, a list of other object file libraries, and an optional *module-definition file,* and creates a file that can be executed under MS OS/2. A module-definition file is an ASCII text file containing information about your application. **Link** uses this information to help build the *.exe* file.



You create MS OS/2 dynamic-link modules in two steps. First you link your compiled dynamic-link source files with the **link** utility. **Link** takes your compiled source files and a module-definition file and creates a dynamic-link module containing dynamic-link entry points that can be invoked under MS OS/2.

Next you use the **implib** utility to create a *.lib* file for your dynamic-link module. Applications developers can use the *.lib* file to resolve external references to your dynamic-link routines. Ordinary *.lib* files resolve external references by supplying the referenced object code. The *.lib* files built by **implib** resolve external references by supplying special records that contain pointers to the target dynamic-link modules and entry points. MS OS/2 binds an application to the dynamic-link routines it calls at load time.



Link can link code for execution in either MS-DOS 3.*x* (real mode) or MS OS/2 (protected mode). **Link** determines which form of executable file to produce by the presence of dynamic-link entries and definition files. If a dynamic-link entry resolves any reference, or if a definition file is requested, then an MS OS/2 executable file is produced; otherwise, an MS-DOS 3.*x* executable file is produced. It is also possible to produce executable files that use the Family API and that run in both real and protected modes.

The following sections describe how to create module definition files, how to use **link**, and how to use **implib**.

20.3.1 Creating Import Libraries

You can create an import library for use by other application developers in resolving external references to your dynamic-link module. The **implib** command creates the library, which is a *.lib* file that can be read by the MS OS/2 linker. The *.lib* file can be specified in the **link** command line with other libraries. Import libraries are recommended for all dynamic-link modules. Without this facility, external references to dynamic-link routines would have to be declared in an **IMPORTS** statement in the module-definition file for the application being linked. **Implib** is supported only in protected mode.

Syntax:

implib *implib-name mod-def-file* [*mod-def-file* ...]

where:

Implib-name is the name you wish the new import library to have.

Mod-def-file is the name of the one or more module-definition files for the MS OS/2 dynamic-link module.

Example:

The following command creates the import library named *mylib.lib* from the module-definition file *mylib.def*:

```
implib mylib.lib mylib.def
```

20.4 Using Module-definition Files

A module-definition file describes the name, size, format, functions, and segments of an application or library for Microsoft Windows. This file is required for Windows applications and libraries, and is also required for programs or libraries that run under MS OS/2.

A module-definition file contains one or more *module statements*. Each module statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the number and names of exported and imported functions. The module statements and the attributes they define are listed as follows:

Statement	Attribute
NAME	Module name
LIBRARY	Library name
CODE	Default attributes for the code segment
DATA	Default attributes for the data segment

SEGMENTS	Segment definitions
STACKSIZE	Local-stack size, in bytes
EXPORTS	Exported functions
IMPORTS	Imported functions
DESCRIPTION	One-line description of the module
STUB	Adds a DOS 3.*x* executable file to the beginning of the module
HEAPSIZE	Local-heap size, in bytes
PROTMODE	Specifies that the module runs only in MS-DOS protected mode.
OLD	Module for export ordinals

20.4.1 Module-definition Files for Windows Applications

A module-definition file for an application must contain a **NAME** statement defining the application's module name. Windows uses this name to identify the application. Although the **NAME** statement is the only required statement for a Windows application, most files contain additional statements, such as **DATA** and **CODE** statements, that define other characteristics of the application.

20.4.2 Module-definition Files for Windows Libraries

A module-definition file for a Windows library must contain a **LIBRARY** statement that defines the library's module name. Windows uses this name to identify the application. The file must also contain an **EXPORTS** statement that lists the functions that the library will export. Functions that are not listed in the **EXPORTS** statement cannot be accessed (unless the module-definition file for an application specifies one or more of the non-listed functions in an **IMPORTS** statement).

20.4.3 Module-definition Statements

This section describes each statement that can appear in a module-definition file and its use.

NAME

Syntax:

NAME [*appname*]

Remarks:

The **NAME** statement identifies the executable file as an application and optionally defines the name of the application. If this statement is included in the module-definition file, the **LIBRARY** statement cannot appear. If neither a **NAME** statement nor a **LIBRARY** statement appears in a module-definition file, the default is **NAME**; that is, the linker creates an application.

If an *appname* is given, it identifies the application when importing or exporting functions. If no *appname* is given, the name of the executable file, without a pathname or extension, is used as the application name.

Example:

The following example assigns the name "calendar" to the application being defined:

```
NAME calendar
```

LIBRARY

Syntax:

LIBRARY [*libraryname*]

Remarks:

The **LIBRARY** statement identifies the executable file as a library and optionally defines the name of the library. If this statement is included in a module-definition file, the **NAME** statement cannot also appear.

If a *libraryname* is given, it identifies the library. If no *libraryname* is given, the name of the library file without a pathname or extension is used.

If no **LIBRARY** statement appears in a module-definition file, the linker assumes that the module-definition file is defining an application. In this case, the name of the application is either the name specified in the **NAME** statement or (if no **NAME** statement appears) the name of the executable file without a pathname or extension.

Example:

The following example assigns the name "calendar" to the dynamic-link module being defined:

```
LIBRARY calendar
```

DESCRIPTION

Syntax:

DESCRIPTION '*text*'

Remarks:

The **DESCRIPTION** statement inserts the specified *text* into the application or library. This statement is useful for embedding source-control or copyright information in an application or library.

Example:

The following example inserts the text "Template Program" into the application or library being defined:

```
DESCRIPTION 'Template Program'
```

CODE

Syntax:

CODE [*load*][*shared*][*execute*][*privilege*]

Remarks:

The **CODE** statement defines the default attributes for code segments within the application or library.

The optional *load* keyword specifies when the segment will be loaded. It can be one of the following:

Keyword	Meaning
PRELOAD	The segment is loaded automatically (this is the default option if no **CODE** statement is included in the module-definition file).
LOADONCALL	The segment is loaded when it is accessed (this is the default option if a **CODE** statement is included without a *load* keyword).

The optional *shared* keyword specifies whether all instances of the program can share this code segment. It can be one of the following:

Keyword	Meaning
SHARED	The segment can be shared.
NONSHARED	The segment cannot be shared (this is the default option).

The optional *execute* keyword specifies whether the segment can be read as well as executed. It can be one of the following:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be executed and read (this is the default option).

The optional *privilege* keyword specifies whether the segment has I/O privilege (that is, whether it can access the hardware directly). If included, *privilege* must be **IOPL**, indicating that the segment has I/O privilege. By default, the segment does not have have I/O privilege.

Example:

The following example defines the module's code segment so that it is loaded only when it is accessed and so that it cannot be shared by more than one copy of the program:

```
CODE LOADONCALL NONSHARED
```

By default, the code segment can be executed and read, and has no I/O privilege.

DATA

Syntax:

DATA [*load*][*instance*][*shared*][*write*][*privilege*]

Remarks:

The **DATA** statement defines the default attributes for the data segments within the application or module.

The optional *load* keyword specifies when the segment will be loaded. It can be one of the following:

Keyword	Meaning
PRELOAD	The segment is loaded automatically (this is the default option if no **DATA** statement is included in the module-definition file).
LOADONCALL	The segment is loaded when it is accessed (this is the default option if a **DATA** statement is included without a *load* keyword).

The optional *instance* keyword describes the *automatic data segment*, the physical segment represented by the group name **DGROUP**, which is the data segment that contains the local stack and heap for the application. The *instance* keyword can be one of the following:

Keyword	Meaning
NONE	No automatic data segment is created.
SINGLE	All instances of the module share a single automatic data segment. In this case, the module is said to have "solo" data.
MULTIPLE	The automatic data segment is copied for each instance of the module. In this case, the module is said to have "instance" data (this is the default option).

The optional *shared* keyword specifies whether all instances of the program can share this data segment. It can be one of the following:

Keyword	Meaning
SHARED	The segment can be shared
NONSHARED	The segment cannot be shared (this is the default option).

Note

The linker makes the automatic data segment attribute—specified by an *instance* value of **SINGLE** or **MULTIPLE**—match the sharing attribute of the automatic data segment—specified by a *shared* value of **SHARED** or **NONSHARED**. Solo data—specified by **SINGLE**—forces shared data segments by default. Instance data—specified by **MULTIPLE**—forces nonshared data by default. Similarly, **SHARED** forces solo data, and **NONSHARED** forces instance data.

If you give a contradictory **DATA** statement such as `DATA SINGLE NONSHARED`, all segments in **DGROUP** are shared, and all other data segments are nonshared by default. If a segment that is a member of **DGROUP** is defined with a *sharing* attribute that conflicts with the automatic data type, a warning about the bad segment, and its flags are converted to a consistent sharing attribute; for example, the following

```
DATA SINGLE
SEGMENTS
_DATA CLASS 'DATA' NONSHARED
```

is converted to

```
_DATA CLASS 'DATA' SHARED
```

The optional *write* keyword specifies whether the segment can be written to. It can be one of the following:

Keyword	Meaning
READONLY	The segment can only be read from.
READWRITE	The segment can be read from or written to (this is the default option).

The *privilege* keyword specifies whether the segment has I/O privilege (that is, whether it can access the hardware directly). If included, *privilege* must be **IOPL**, indicating that the segment has I/O privilege. By default, the segment does not have have I/O privilege.

Example:

The following example defines the application's data segment so that it is loaded only when it is accessed and so that it cannot be shared by more than one copy of the program:

```
DATA LOADONCALL NONSHARED
```

By default, the data segment can be read and written; the automatic data segment is copied for each instance of the module; and the data segment has no I/O privilege.

SEGMENTS

Syntax:

SEGMENTS ['] *segname* ['] [CLASS ['*classname*']] [*minalloc*] [*segflags*]

Remarks:

The **SEGMENTS** statement defines the attributes of one or more segments in the application or library on a segment-by-segment basis. Usually, these segments are one or more segments that are also defined in an object module. Since the linker assigns segment positions in the order in which it finds segment definitions, this statement may also be used to specify segment order.

The *segname* option gives the name of the new segment. It may optionally appear enclosed in single quotation marks ('), which are required if the segment name is **CODE** or **DATA**.

The optional *classname* keyword specifies the class name of the segment. If you do not include a *classname*, the linker assumes that the class is **CODE**. Since the linker gives a type of **CODE** to any segment whose name ends in "CODE", the linker assumes that the segment has type **CODE** if the **CLASS** option is omitted.

The *minalloc* option is an integer that specifies the minimum number of bytes to be allocated for this segment.

The *segflags* option is any combination of the following flags:

Flag	Meaning
SHARED	The segment can be shared
NONSHARED	The segment cannot be shared.
PRELOAD	The segment is loaded automatically.
LOADONCALL	The segment is loaded when it is accessed.
READONLY	The segment can only be read from.
READWRITE	The segment can be read from or written to.
IOPL	The segment has I/O privilege.
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be executed and read.

If no *segflags* are given, the segment is assigned the default code or data flags for its segment type (code or data).

If at least one *segflag* is given, the default code and data flags do not apply. Any attribute that is not specified in the *segflags* is taken from the following list of default attributes:

NONSHARED
LOADONCALL
EXECUTEREAD (code) or **READWRITE** (data)
NONSHARED
no **IOPL**

HEAPSIZE

Syntax:

HEAPSIZE *number*

Remarks:

The **HEAPSIZE** statement specifies the number of bytes the application or library needs for its local heap. The application or library uses the local heap whenever it allocates local memory. The *number* must be an integer less than or equal to 65,536, the number of bytes in a single physical segment. If no **HEAPSIZE** statement is specified, the default local-heap size is zero.

Example:

The following example allocates 4096 bytes of local-heap space:

```
HEAPSIZE 4096
```

STACKSIZE

Syntax:

STACKSIZE *number*

Remarks:

The **STACKSIZE** statement performs the same function as the **/STACK** linker option. It specifies the number of bytes the application or library needs for its local stack. The module uses the local stack whenever it calls its own functions. The *number* must be an integer.

Example:

The following example allocates 4096 bytes of local-stack space:

```
STACKSIZE 4096
```

EXPORTS

Syntax:

EXPORTS
entryname[=*internalname*][@ *ordinal*][RESIDENTNAME][NODATA][*iopl*]
[*entryname*[=*internalname*][@ *ordinal*][RESIDENTNAME][NODATA][*iopl*]...]

Remarks:

The **EXPORTS** statement defines the names and attributes of the functions that will be exported to other modules, or of functions that run with I/O privilege.

The **EXPORTS** keyword marks the beginning of the definitions. It may be followed by up to 3072 export definitions, each on a separate line. You must give an export definition for each routine in an application or library.

The *entryname* specification, which defines the function name, is the name that other programs must use to access the exported function. The =*internalname* option defines the actual name of the export function if *entryname* is not the actual name. In export definitions for functions that run with I/O privilege, this option and the *iopl* option are the only valid options.

The @ *ordinal* parameter defines the function's *ordinal* value. In this parameter, *ordinal* is an integer that defines the location of the function's name in the module's string table. If this option is specified, the entry point can be invoked by name or by ordinal.

The optional keyword **RESIDENTNAME** specifies that the function's name must be resident at all times. It can only be used if an ordinal value is given.

The optional keyword **NODATA** specifies that the function is not bound to the global (shared) data segment. When the function is invoked, it uses the current data segment. If this keyword is *not* given, and if the module is a *singledata* library module, then the routine is assumed to use the global data segment and the first instruction in the routine's prolog must be

```
mov ax,#ds-value
```

The *iopl* option must be included for functions that execute with I/O

privilege. It is a numeric value that specifies the number of words that the function expects to be passed as parameters (this is the number of words that are copied from the caller's stack to the function's stack when the function is called). Routines that execute with I/O privilege are allocated a 512-byte stack. In export definitions for functions that run with I/O privilege, this option and the *entryname* specification are the only valid options.

Example:

The following **EXPORTS** statement defines three export functions: **SampleRead**, **StringIn**, and **CharTest**:

```
EXPORTS
        SampleRead = read2bin @1 8
        StringIn = str1 @2 4
        CharTest NODATA
```

SampleRead is known internally as **read2bin**; its name appears first in the module's string table, and it expects to be passed eight parameters. Similarly, **StringIn** is known internally as **str1**; its name appears second in the module's string table, and it expects to be passed four parameters. The **CharTest** function is defined so that it will not be bound to a specific data segment; when invoked, it will use the current data segment.

IMPORTS

Syntax:

IMPORTS [*internalname*=]*modulename*.[*entryname*|*entryordinal*] [[*internalname*=]*modulename*.[*entryname*|*entryordinal*]...]

Remarks:

The **IMPORTS** statement defines the names of the functions that will be imported for the application or library. Usually, **link** uses the import-library file (created by the **implib** utility) to resolve external references to functions. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

The **IMPORTS** keyword marks the beginning of the definitions. It may be followed by import definitions; the only limit on import definitions is that the total amount of space required for their names must be less than 64K. Each import definition must appear on a separate line.

If you include an *internalname*, it defines the name that the importing module actually uses to call the exported function.

The *modulename* is the name of the application or library that contains the function.

The optional *entryname* or *entryordinal* parameter specifies the function to be imported. The *entryname* is the actual function name; the *entryordinal* is the ordinal value of the function. If *entryordinal* is used, an *internalname* must be given.

Example:

The following **IMPORTS** statement defines three functions to be imported: **SampleRead**, **SampleWrite**, and an exported function with an ordinal value of **read.1**; it uses the internal name **Sample** to call the **SampleRead** and **SampleWrite** functions:

```
IMPORTS
        Sample.SampleRead
        Sample.SampleWrite
        read = Read.1
```

STUB

Syntax:

STUB *filename*

Remarks:

The **STUB** statement adds *filename*, an MS-DOS executable file, to the beginning of the application or library being created. (If **link** does not find this file in the current directory, it searches in the list of directories specified in the PATH environment variable.)

The specified file is invoked if the module is run under MS-DOS. For modules that cannot run under MS-DOS, the executable file can simply display a warning message and terminate.

Example:

The following example appends the MS-DOS executable file *sample.exe* to the beginning of the module:

```
STUB SAMPLE.EXE
```

Sample.exe is run if the module is run under MS-DOS.

PROTMODE

Syntax:

PROTMODE

Remarks:

The **PROTMODE** statement specifies that the module will run only in protected mode on the 80286 processor.

If this statement is not included in the module-definition file, the application can be run in either real or protected mode on the 80286.

20.5 The Message Utilities

MS OS/2 includes four message utilities that let you create and modify message files. These utilities are

msgbind
mkmsgf
helpmsg

Msgbind

Purpose:

The **msgbind** utility modifies an MS OS/2 executable file to contain RAM-based messages.

Syntax:

msgbind [*drive:*][*path*]*infile*[*.ext*]

Remarks:

The **msgbind** utility reads an input file that describes which executable files (*.exe*) files are to be modified. For each executable file, **msgbind** specifies which message files are to be scanned. For each message file, it specifies which messages are to be included in the executable file.

The file *infile*[*.ext*] is an ASCII file with the following format:

1. If a line begins with a right angle bracket (>), the text immediately following is the name of the MS OS/2 executable file to modify. All RAM messages following this line will be output to the specified executable file until either the end of the input file is reached *or* another line that begins with a right angle bracket is seen. The latter case allows you to use **msgbind** just once to update multiple executable files.
2. If a line begins with a left angle bracket (<), the text immediately following the bracket is the name of a file (sent as output from the **mkmsgf** utility) to read messages from. All messages numbers that follow are looked up in the specified message file and then copied to the current output executable file. The message number list is terminated either by the end of the input file, the occurrence of a new output specification, or the specification of a new input message file. The latter case allows messages from multiple message files to be included in the same executable file.
3. The *.exe* filename and the message filename allow for a drive and directory path to be named. If you don't specify a drive or directory path, the default values are the current drive and directory, respectively.

4. Otherwise, a line is assumed to begin with a message number (a three-character component ID and a four-digit message number).

The following is an example of the format of the input file:

```
>[d:][path]CMD.EXE
<[d:][path]DOSUTIL.MSG
DOS0100
DOS0123
DOS0245
>[d:][path]FORMAT.EXE
<[d:][path]DOSUTIL.MSG
DOS0001
DOS0006
<[d:][path]FORMAT.MSG
FMT0001
FMT0002
```

The files *dosutil.msg* and *format.msg* in this example represent the output message filenames from the **mkmsgf** utility. For more information about creating message files, see the **mkmsgf** utility in this chapter.

Mkmsgf

Purpose:

The **mkmsgf** utility creates a message file for use with MS OS/2 message facilities.

Syntax:

mkmsgf [*drive*:][*path*]*infile* [*.ext*] [*drive*:][*path*]*outfile*

Remarks:

Mkmsgf reads the input source message file, *infile*, that you specify, and creates an output file, *outfile*, that is usable by the Message Retriever in MS OS/2.

Optionally, **mkmsgf** also reads an input file of message profiles. This input file identifies messages from the source file that are to be bound to an *.exe* file message segment. The messages are then loaded as part of the *.exe* binary load image and are resident in RAM.

The file *message.msg* is a standard ASCII file with the following format:

1. Comments are allowed only at the beginning of the file before the line containing the three-character system component identifier. Comment lines are indicated by a semicolon (;) in the first position on the line. If a semicolon is found at the beginning of any line *after* the component ID line, it is treated as part of the message text.

2. The component ID line contains three characters that identify the system component that the message file is for.

3. Following the component ID line are the components messages. Each message is composed of a message header, followed by mixed single- and double-byte text.

   A message header is made up of the three characters of the component (from component ID line), a four digit number, a message classification character (see the following list), a colon (:), and a space. The message header must begin in the first position on the

line.

E	Error
H	Help
I	Information
P	Prompt
W	Warning

4. Messages may span multiple lines, but only one header is allowed for each message.

5. Message numbering may start at any number, but messages must be numbered sequentially. If a message number is not used, an empty entry must take its place in the text file.

6. If the character sequence *%0* is found at the end of a line, **mkmsgf** will not include the RETURN/LINEFEED in the output file for that message. This makes it possible to prompt for user input on the same line.

The file specified in *outfile* is the output file that will contain the indexed message file that the **DosGetMessage** call will use. The output file cannot have the same name as the input file.

Following is an example of a message file:

```
; This is an example of a
; message file for component DOS
; starting with three comment lines
DOS
DOS0100E: File not found
DOS0101E:
DOS0102E:
DOS0103E:
DOS0104I: %1 files copied
DOS0105E:
DOS0106W: Warning!  All data will be destroyed!
DOS0107E:
DOS0108E:
DOS0109E:
DOS0110P: Do you wish to apply these patches (Y or N)? %0
DOS0111E: Divide overflow
```

Note

Message DOS0110 prompts the user with the message, "Do you wish to apply these patches (Y or N)?" (The %0 supresses a new line so the user can answer on the same line.) A subsequent "read keyboard with echo" message will cause the response characters to be displayed immediately after the "(Y or N)" message.

The maximum number of messages that a message file can contain and be successfully built by **mkmsgf** is approximately 6,000 messages, depending upon the size of the **mkmsgf** data segments. It may be less than 6,000 messages, depending upon the amount of free memory space. (1000 messages require about 6K bytes of memory.)

Optionally, **msgbind** may be used to modify an executable *.exe* file to bind a segment of messages to it. See the **msgbind** utility for more details on how to do this.

To make use of the **helpmsg** utility, the output message files should follow certain naming conventions:

- The message file name should consist of the three-character component identifier with an extension of *.msg*.
- The help message file name should consist of the three-character component identifier followed by the "H" character (to indicate a help file) with an extension of *.msg*.

So for the component identifier "DOS," the message file name (the output file) would be *dos.msg*. Its corresponding help file would be *dosh.msg*.

For more information, refer to the **helpmsg** utility.

Helpmsg

Purpose:

The **helpmsg** utility provides help information related to a warning or error message.

Syntax:

helpmsg *number*

where:

number is the number of the help message for which you want additional information.

Remarks:

The message number consists of seven alphanumeric characters consisting of a 3-character component ID followed by a 4-digit message number. The **helpmsg** utility accepts the message number (without leading zeros) as a valid number. The component ID will default to the system's component ID; for example, "DOS."

Although the **helpmsg** command may be used for any displayed message (that has a message number), its intended use is to provide warning and error messages to help you identify possible causes and solutions to the condition.

The **helpmsg** utility displays the original message followed by its associated help message. If the original message requires parameter inserts, the **helpmsg** utility will insert three asterisks (***) for the variable data to indicate a parameter insert.

To retrieve the original message and the assocated help message based on the message number, the **helpmsg** utility requires that the message files be in the following format and one of the following directories:

The message file name should consist of the 3-character component ID with an extension of *.msg*

The help message file name should consist of the first 3 characters of the component ID followed by the "H" character (to indicate a help file) with an extension of *.msg*

This format allows applications to use the **helpmsg** utility with other message files.

The message file and the help message file should reside in one of the following directories:

- The system root directory
- The current directory
- A directory path listed in the APPEND statement

If these paths fail, **helpmsg** retries using the paths specified in the environment.

The following gives examples of sample input and sample output.

Sample error message output:

```
DOS0100: File not found
```

Sample input to the **helpmsg** utility:

```
HELPMSG DOS0100
```

The message filename for message number DOS0100 is *dos.msg* and the help message filename is *dosh.msg*.

Sample output to the **helpmsg** utility:

```
DOS0100: File not found
```

This message appears if the filename is incorrect or does not exist. If this message appears, you should check to make sure the filename is correct and then retry the command.

The **helpmsg** utility outputs the following message if no help text is available for the message number requested:

```
No help text available for message DOS 0100
```

Chapter 21
New Executable File Format

21.1 Introduction

This chapter contains general information about the contents of the executable file header format for MS OS/2. This file format, which provides support for true segmented programs and for the 80286 protected mode, is a superset of the MS-DOS 3.*x* *.exe* format. For MS OS/2 to recognize the new executable format, the existing *.exe* format is used with a slight modification:

> The word at offset 18H in the existing *.exe* file contains the relative byte offset to the relocation table.

If this offset is 40H, it indicates a new format *.exe* file. In addition, the long word at offset 3CH is the relative byte offset from the beginning of the file to the beginning of the new format's executable header. The remainder of the old format header describes a small *stub* program that will either print an error message l or bring in a new *.exe* loader. Old *.exe* files will continue using the MS-DOS 3.*x* file format.

21.2 Executable File Information

A new *.exe* file represents either an application or a module containing dynamic-link entry points. The following entry items are applicable to applications only:

- The segment:offset of the entry point.
- The segment number passed in DS on program invocation.
- The segment:offset of the stack.
- The stack size.

The *.exe* header keeps the following items on a per-segment basis:

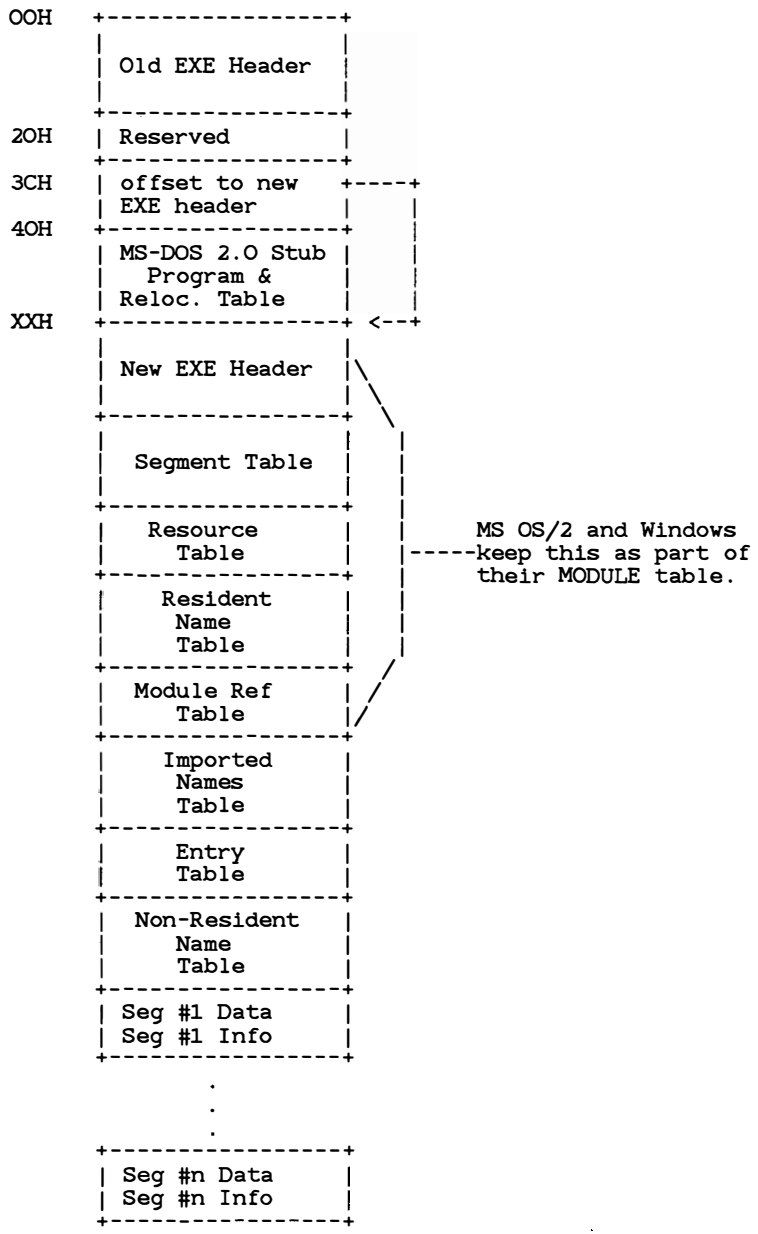
- It keeps code or data.
- If the segment is a code segment, it keeps whether the segment is preload versus demand load.
- If the segment is a data segment, the *.exe* header keeps whether the segment is preload versus demand load and whether the segment is read only or read/write.

- It keeps whether segment requires I/O privilege.
- It also keeps fix-up records for external references, including references resolved within this *.exe* file and those resolved via dynamic linking. These external references include module-name/entry-point name pairs and/or module-name/entry-point ordinal pairs. Each pair identifies a dynamic-link module (*.exe* file) and a dynamic-link routine within that module.
- The *.exe* header keeps fix up records to support 80287 emulation.
- And it keeps Debug information.

The *.exe* header keeps the following items on a per-FAR-CALL entry point:

- It keeps the entry-point name.
- It keeps whether the entry-point name is exported.
- It also keeps the segment:offset of the entry point.
- And finally, it keeps the number of parameters that must be copied from the caller's stack to the callee's stack when an entry point requiring I/O privilege is invoked.

Following is a diagram of the new *.exe* file format:



21.2.1 Behavior Bit and Header Information

The first portion of the *.exe* header contains the following information:

20H–3BH

Reserved.

3CH

DD This double word is the offset to the new executable header.

40H

This program, to which the MS-DOS 2.0 header points, either prints an error message or loads a new executable *.exe* loader.

21.2.2 The New *.exe* Header

***XX*H**

This offset denotes the beginning of the new executable header within the file header. It contains the following information:

DW Signature word.
N is the low-order byte.
E is the high-order byte.

DB Version number of the linker.

DB Revision number of the linker.

DW File offset of the Entry Table relative to the beginning of the new *.exe* header.

DW Number of bytes in the Entry Table.

21.2.3 Module Table Information

This marks the beginning of the area that MS OS/2 keeps as part of its module table:

DD CRC-32 of the entire contents of the file (with the following words taken as 00 during the calculation).

DW Flag word.

0000H = NOAUTODATA.

0001H = SINGLEDATA (Solo).

0002H = MULTIPLEDATA (Instance).

0004H = Runs in real mode.

0008H = Runs in protected mode (If 0CH is set, runs in either mode).

2000H = Errors detected at link time.

4000H = Non-conforming program (A valid stack is not maintained).

8000H = Library module.

The SS:SP information is invalid and CS:IP points to an initialization procedure called with AX equal to the module handle. This initialization procedure must execute a far return to the caller, with AX not equal to zero indicating success and AX equal to zero indicating failure to initialize. DS is set to the library's data segment if the SINGLEDATA flag is set. Otherwise, DS is set to the caller's data segment.

Only executable programs that have their SINGLEDATA flag set may be dynamically linked. If SINGLEDATA is set, MULTIPLEDATA must be cleared.

DW Segment number of the automatic data segment (index into the Segment Table).

Set to zero if the SINGLEDATA and MULTIPLEDATA flag bits are cleared.

DW Initial size (in bytes) of the dynamic heap added to the data segment.

This word equals zero if there is no local allocation.

DW Initial size (in bytes) of the stack added to the data segment.

This word equals zero if SS does not equal DS.

DD Segment number:offset of CS:IP.

DD Segment number:offset of SS:SP.

Segment number is an index into the module's segment table.

The first entry in the segment table is segment number 1.

If SS equals the automatic data segment and SP equals zero, the stack pointer is set to the top of the automatic data segment just below the additional heap area. This is shown as follows:

```
+-------------------------+
| additional dynamic heap |
+-------------------------+ <- SP
|    additional stack     |
+-------------------------+
|   loaded data segment   |
+-------------------------+ <- DS, SS
```

DW Number of entries in the Segment Table.

DW Number of bytes in the Non-Resident Name Table.

DW Offset of the Segment Table relative to the beginning of the new *.exe* header.

DW Offset of the Resource Table relative the to beginning of the new *.exe* header.

DW Offset of the Resident Name Table relative to the beginning of the new *.exe* header.

DW Offset of the Module Reference Table relative to the beginning of the new *.exe* header.

DW Offset of the Imported Names Table relative to the beginning of the new *.exe* header.

DD Offset of the Non-Resident Name Table relative to the beginning of the file.

DW Number of movable entry points.

DW Shift count of the logical sector alignment, log(base 2) of the segment sector size (default 9).

DB 12 dup(0) reserved.

21.2.4 Segment Table

N Segment Table Entries (1-based):

The first entry in the Segment Table is segment number 1.

DW *n*-byte logical sector offset to the contents of the segment data relative to the beginning of the file (zero means no file data).

DW Length of the segment in the file (in bytes) (zero means 64K bytes).

DW Flag word.

```
TYPE_MASK = 0007H       ; Segment type field
CODE = 0000H            ; Code segment type
DATA = 0001H            ; Data segment type
ITERATED = 0008H        ; Segment data is iterated
MOVABLE = 0010H         ; Segment is not fixed
PURE = 0020H            ; Segment can be shared
PRELOAD = 0040H         ; Segment is not demand loaded
ERONLY = 0080H          ; Execute only if code segment
                        ; Read only if data segment
```

```
RELOCINFO = 0100H       ; Set if segment has reloc records
DEBUGINFO = 0200H       ; Set if segment has debug info
SEGDPL = 0C00H          ; Reserved for 286 DPL bits
DISCARD = F000H         ; Discard priority
```

DW Minimum allocation size of the segment (in bytes) Total size of the segment (0 means 65,536).

21.2.5 Resource Table

DW Alignment shift count for resource data.

N Iterations of Record:

DW Type ID – Integer type if the high-order bit is set (8000H); otherwise, the offset to the type string, relative to the beginning of the Resource Table.

If the type ID equals zero, it marks the end of the resource records

DW Number of resources for this type

DD Reserved.

Number of resources/copies of the Resource Entry (8 bytes).

DW Offset to the contents of the resource data relative to the beginning of the file. The offset is in terms of alignment units specified at the beginning of the Resource Table.

DW Length of the resource in the file (in bytes)

DW Flag word.

```
MOVABLE = 0010H     ; Resource is not fixed.
PURE = 0020H        ; Resource can be shared.
PRELOAD = 0040H     ; Resource is not demand
                    ; loaded.
```

DW Resource ID—Integer type if the high-order bit is set (8000H); otherwise, the offset to the resource string, relative to the beginning of the Resource Table.

DD Reserved.

Resource type and name strings are stored at the end of the resource table. Note that these strings are *not* null-terminated.

DB Length of type or name, equals zero0 if end of resource table.

DB ASCII text of type or name; case-sensitive.

21.2.6 Module Reference Table

N Entries of the Form (1-based):

DW Offset within the Imported Names Table to the module name string.

21.2.7 Entry Table

N Bundles of Entry Definitions (1-based):

The ordinal value of an entry point is its ordinal within the Entry Table, counting the first entry as ordinal number 1. The loader must scan the bundles until it finds the one containing the entry point. The loader can then multiply the ordinal value by entry size to the index the proper entry.

The linker forms bundles in the densest manner it can, under the restriction that it cannot reorder entry points to improve bundling. The reason for this restriction is that other *.exe* files may refer to entry points within this bundle by their ordinal in the following table:

DB The number of entries in this bundle.

All records in one bundle are either movable or refer to the same fixed segment. This byte equals zero if no more bundles are in the Entry Table.

DB Segment indicator for this bundle.

000—Unused.

0FFH—Movable segment, segment number is in the 6-byte entry shown following.

Other—Segment number of the fixed segment.

If this is a movable segment, the entries are 6 bytes:

DB Flags.

0000 0001 - Set if the entry is exported.

0000 0010 - Set if the segment uses shared data segments.

INT 3FH.

DB Segment number.

DW Offset.

If this is a fixed segment, the entries are 3 bytes:

DB Flags.

0000 0001—Set if the entry is exported.

0000 0010—Set if the entry uses a global (shared) data segment.

"MOV AX,#DS-value" must be the first instruction in the prologue of this entry. This flag may be set only for SINGLEDATA library modules.

DW Offset.

21.2.8 Resident or Non-Resident Name Table (3 + *n* bytes)

The strings are case-sensitive and *not* null-terminated.

DB Length of string. This byte equals zero if no more are strings in the table.

DB ASCII text of string.

DW Ordinal number (index into Entry Table).

The first string in the Resident Name Table is the module name.

The first string in the Non-Resident Name Table is the module description.

21.2.9 Imported Names Table (1 + *n* bytes)

The strings are case-sensitive and *not* null-terminated.

DB Length of name. This byte equals zero if no more strings are in the table.

DB ASCII text of name.

21.2.10 Per-Segment Data

IF ITERATED

DW Number of iterations.

DW Number of bytes of data.

DB Data bytes.

ELSE

DB Data bytes.

IF RELOCINFO

DW Number of relocation items.

Relocation Item (8 bytes):

DB Source type (32-bit address, 16-bit segment, 16-bit offset).
SEGMENT = 02H.
FAR_ADDR = 03H.
OFFSET = 05H.
SOURCE_MASK = 07H.

DB Flags.
INTERNALREF = 00H.
IMPORTORDINAL = 01H.

IMPORTNAME = 02H.

TARGET_MASK = 03H.

ADDITIVE = 04H.

DW Offset.

The offset is within this segment of the source chain.

If the ADDITIVE flag is set, then add the target value to the source contents instead of replacing the source and following the chain. The source chain is a 0XFFFFH-terminated linked-list within this segment of all references to the target.

Target:

INTERNALREF.

DB Segment number for the fixed segment, or 0FFH, if movable.

DB 0

DW Index (that is, ordinal) into the Entry Table relative to its start, if and only if it is movable (if it points to the flag byte).

Offset into the segment, if it is fixed.

IMPORTNAME.

DW Index into the Module Reference Table.

DW Offset within the Imported Names Table to procedure name string.

IMPORTORDINAL.

DW Index into the Module Reference Table.

DW Procedure ordinal number.

IF DEBUGINFO.

DW Number of bytes of debug information.

Debug information.

Chapter 22
Microsoft Relocatable Object Module Formats

22.1 Introduction

This chapter presents the object record formats that define the relocatable object language for the 8086, 80186, and 80286 microprocessors. The 8086 object language is the output of all language translators that have an 8086 processor and that will be linked by the Microsoft linker. The 8086 object language is used for input and output for object language processors such as linkers and librarians, and is used in the XENIX, PC/MS-DOS, and MS OS/2 operating systems.

The 8086 object module formats let you specify relocatable memory images that may be linked together. These formats also allow efficient use of the memory mapping facilities of the 8086 family of microprocessors.

The following table lists the record formats (each described in this chapter) that Microsoft supports:

Table 22.1

Object Module Record Formats

Symbol Definition Records
Public Names Definition Record
Communal Names Definition Record
Local Symbols Record
External Names Definition Record
Line Numbers Record
Data Records
Logical Enumerated Data Record
Logical Iterated Data Record
T-Module Header Record
List of Names Record
Segment Definition Record
Group Definition Record
Fixup Record
Module End Record
Comment Record

Note

If an object module contains any undefined values, the behavior of the Microsoft linker is undefined. All undefined values should be considered reserved by Microsoft for future use.

22.1.1 Definition of Terms

The following terms are fundamental to 8086 relocation and linkage:

OMF - Object Module Formats

MAS - Memory Address Space

The 8086 MAS is one megabyte (1,048,576 bytes). Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

Module

A *module* is an "inseparable" collection of object code and other information produced by a translator.

T-module

A *T-module* is a module created by a translator, such as Pascal or FORTRAN.

The following restrictions apply to object modules:

- Every module should have a name. Translators provide default names (possibly filenames or null names) for T-modules if neither the source code nor the user specifies otherwise.
- Every T-module in a collection of linked modules should have a different name so that symbolic debugging systems can distinguish the various line numbers and local symbols. The Microsoft linker does not require or enforce this restriction.

Frame

A *Frame* is a contiguous region of 64K of memory address space (MAS), beginning on a paragraph boundary (i.e., on a multiple of 16 bytes) or on a selector on the 80286 processor. This concept is useful because the contents of the four 8086 segment registers define four (possibly overlapping) Frames; no 16-bit address in the 8086 code can access a memory location outside the current four Frames.

LSEG – Logical Segment

A *logical segment* (*LSEG*) is a contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither the size nor the location in MAS are necessarily determined during translation: the size, although partially fixed, may not be final because the linker may combine the LSEG when linking with other LSEGs, forming a single LSEG. So that it can fit in a Frame, an LSEG must not be larger than 64K. Thus, a 16-bit offset, from the base of a Frame that covers the LSEG, may address any byte in that LSEG.

PSEG – Physical Segment

This term is equivalent to Frame. Some prefer "PSEG" to "Frame" because the terms PSEG and LSEG reflect the "physical" and "logical" nature of the underlying segments.

Frame Number

Every Frame begins on a paragraph boundary. The "paragraphs" in MAS can be numbered from 0 through 65,535. These numbers, each of which defines a Frame, are called *Frame Numbers.*

Group

A *Group* is a collection of LSEGs defined at translation time, whose final locations in MAS have been constrained so that at least one Frame exists that covers (contains) every LSEG in the collection.

The notation "Gr A(X,Y,Z,)" means that LSEGs X, Y, and Z form a group named A. That X, Y, and Z are all LSEGs in the same group does not imply any ordering of X, Y, and Z in MAS, *nor does it imply any contiguity between X, Y, and Z.*

The Microsoft linker does not currently allow an LSEG to be a member of more than one group.

Canonic

On the 8086 processor, any location in MAS is contained in exactly 4096 distinct frames, but one of these frames can be distinguished because it has a higher Frame Number. This Frame is called the *canonic* Frame of the location. In other words, the canonic Frame of a given byte is the Frame chosen so that the byte's offset from that Frame lies in the range 0 to 15 (decimal).

For example, suppose FOO is a symbol defining a memory location. You would then refer to this Frame as the "canonic Frame of FOO." Similarly, if S is any set of memory locations, then a unique Frame exists that has the lowest Frame Number in the set of canonic frames of the locations in S. This unique Frame is called the canonic Frame of the set S. You might refer similarly to the canonic Frame of an LSEG or of a group of LSEGs.

Segment Name

LSEGs are assigned *Segment Names* at translation time. These names serve two purposes:

- During linking they play a role in determining which LSEGs are combined with other LSEGs.
- They are used in assembly source code to specify membership in groups.

Class Name

The translator may optionally assign *Class Names* to LSEGs during translation. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same Class Name.

The Microsoft linker applies the following semantics to Class Names: the Class Name "CODE", or any Class Name whose suffix is "CODE", implies that all segments of that class contain only code and may be considered read-only. Such segments may be overlaid if you specify the module containing the segment as part of an overlay.

Overlay Name

The linker may optionally assign an *Overlay Name* to LSEGs. The Overlay Name of an LSEG is ignored by Microsoft language linkers for version 3.00 and later languages, but the standard MS-DOS linker supports it.

Complete Name

The *Complete Name* of an LSEG consists of the Segment Name, Class Name, and Overlay Name. The linker combines LSEGs from different modules if their Complete Names are identical.

22.2 Module Identification and Attributes

A *module header record*, which provides a module name, is always the first record in a module. In addition to having a name, a module may represent a main program and may have a specified starting address. When linking multiple modules together, you should give only one module with the main attribute. If more than one main module appears, the first takes precedence.

In summary, modules may or may not be main and may or may not have a starting address.

22.2.1 Definition of a Segment

A module is a collection of object code defined by a sequence of records that a translator produces. The object code represents contiguous regions of memory whose contents the linker determines during translation. These regions are LSEGs. A module defines the attributes of each LSEG. The *segment definition* record (SEGDEF) is responsible for maintaining all LSEG information (name, length, memory alignment, etc.). The linker requires the LSEG information when you combine multiple LSEGs and

when it establishes segment addressability. The SEGDEF records must follow the first header record.

22.2.2 Addressing a Segment

The 8086 addressing mechanism provides segment base registers from which you may address a 64K-byte region of memory (a Frame). There is one *code segment* base register (CS), two *data segment* base registers (DS, ES), and one *stack segment* base register (SS).

The possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would be the case in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that will fit in one memory frame. Then all the LSEGs may be addressed using the same base register value. This addressable unit is a Group and has been defined earlier in Section 7.2, "Definition of Terms."

To establish addressability of objects within a Group, you must explicitly define each Group in the module. The *group definition* record (GRPDEF) lists constituent segments by their Segment Names.

The GRPDEF records within a module must follow all SEGDEF records because GRPDEF records will reference SEGDEF records in defining a Group. The GRPDEF records must also precede all other records except header records, which the linker must process first.

22.2.3 Defining a Symbol

The Microsoft linker supports three different types of records belonging to the class of symbol definition records. The types are *public names definition* records (PUBDEFs), *communal names definition* records (COMDEFs), and *external names definition* records (EXTDEFs). You use these record types to define globally visible procedures and data items and to resolve external references.

22.2.4 Indices

"Index" fields appear throughout this chapter. An *index* is an integer that selects a particular item from a collection of items; for example: Name Index, Segment Index, Group Index, External Index, Type Index, etc.

Note

An index is normally a positive number. The index value zero is reserved, and may carry a special meaning depending on the type of index (for example, a Segment Index of zero specifies the "Unnamed" absolute pseudo-segment; a Type Index of zero specifies the "Untyped type.")

In general, indices must assume values that are quite large (that is, much larger than 255). Nevertheless, a great number of object files contain no indices with values greater than 50 or 100. Therefore, indices are encoded in one or two bytes, as required.

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, the index is a number between 0 and 127, occupying one byte. If the bit is 1, the index is a number between 0 and 32K–1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

22.3 Conceptual Framework for Fixups

A *fixup* is a modification to object code that achieves address binding that a translator requested and a linker performed.

Note

This is the linker's definition of fixup. Nevertheless, the linker can modify object code (make a "fixup") that does not conform to this definition. For example, binding code to either hardware or software floating-point subroutines is a modification to an operation code, which is treated as an address. The previous definition of fixup is not intended to disallow or discourage modifications to the object code.

8086-family translators need four kinds of data to specify a fixup:

- The place and type of a Location to be fixed up.
- One of two possible fixup modes.
- A Target, which is the memory address that Location must refer to.
- A Frame defining a context in which the reference takes place.

Location - There are five types of Locations: a *pointer*, a *base*, an *offset*, a *hibyte*, and a *lobyte*.

The vertical alignment of the following figure illustrates four points (remember that the high-order byte of a word in 8086 memory is the byte with the higher address):

- A *base* is the high-order word of a *pointer* (the linker doesn't care whether the low-order word of the pointer is present).
- An *offset* is the low-order word of a *pointer* (the linker doesn't care whether the high-order word follows).
- A *hibyte* is the high-order half of an *offset* (the linker doesn't care whether the low-order half precedes).
- A *lobyte* is the low-order half of an *offset* (the linker doesn't care whether the high-order half follows).

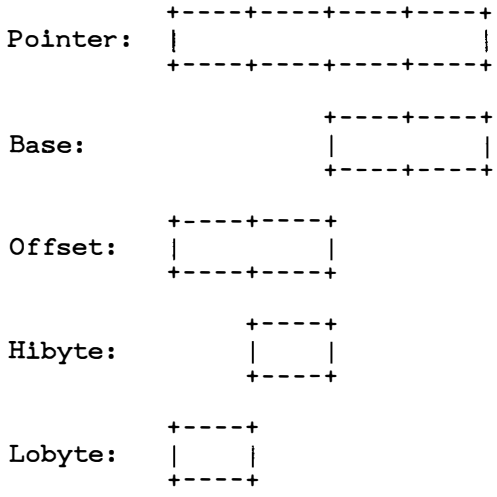


Figure 22.1 Location Types

A Location is specified by two kinds of data: the Location type, and where the Location is.

The Location type is specified by the LOC subfield in the FIXUP record's LOCAT field; where the location is specified by the Data Record offset subfield in the FIXUP record's LOCAT field.

Mode - The Microsoft linker supports two kinds of fixups: *self-relative* and *segment-relative.*

Self-relative fixups support the 8-bit and 16-bit offsets used in CALL, JUMP, and SHORT-JUMP instructions. Segment-relative fixups support all other addressing modes of the 8086.

Target - The Target is the location in MAS that the linker references. (More explicitly, the linker considers the Target the lowest byte in the object that it is referencing.) The linker specifies a Target by one of six methods. There are three "primary" methods and three "secondary" ones. Each primary method of specifying a Target uses two kinds of data: an Index Number *X*, and a displacement *D*.

(T0) *X* is a Segment Index. The Target is the *D*th byte in the LSEG that the Segment Index identifies.

(T1) *X* is a Group Index. The Target is the *D*th byte in the LSEG that the Group Index identifies.

(T2) *X* is an External Index. The External Index identifies the External Name that (eventually) gives the address of a byte. The *D*th byte following this byte is the Target.

Each secondary method of specifying a Target uses only one item of data — the Index Number *X*; this assumes an implicit displacement equal to zero.

(T4) *X* is a Segment Index. The Target is the 0th (first) byte in the LSEG that the Segment Index identifies.

(T5) *X* is a Group Index. The Target is the 0th (first) byte in the LSEG in the specified group located (eventually) lowest in MAS.

(T6) *X* is an External Index. The Target is the byte whose address is the External Name that the External Index identifies.

The following nomenclature describes a Target:

Target: SI(*Segment Name*), *displacement*	[T0]
Target: GI(*Group Name*), *displacement*	[T1]
Target: EI(*Symbol Name*), *displacement*	[T2]
Target: SI (*Segment Name*)	[T4]
Target: GI (*Group Name*)	[T5]
Target: EI (*Symbol Name*)	[T6]

The following examples illustrate how this notation is used:

Target: SI(CODE), 1024	The 1025th byte in the segment "CODE."
Target: GI(DATAAREA)	The location in MAS of a group called "DATAAREA."
Target: EI(SIN)	The address of the external subroutine "SIN."
Target: EI(PAYSCHEDULE), 24	The 24th byte following the location of an external data structure called "PAYSCHEDULE."

Frame

Every 8086 memory reference is to a location contained within a Frame. This Frame is designated by the content of a segment register. For the linker to form a correct, usable memory reference, it must know what the Target is, and to which Frame the reference is being made. Thus, every fixup specifies such a Frame, in one of six methods. Some methods use data, *X*, which is in Index Number, as above. Other methods require no data.

The five methods of specifying Frames are as follows:

(F0) *X* is a Segment Index. The Frame is the canonic Frame of the LSEG that the Segment Index defines.

(F1) *X* is a Group Index. The Frame is the canonic Frame defined by the group (that is, the canonic Frame defined by the LSEG in the group located (eventually) lowest in MAS).

(F2) *X* is an External Index. The Frame is determined when the linker finds the External Name's public definition. There are three cases:

- (F2a) The linker defines the symbol relative to some LSEG, and there is no associated Group. The linker also specifies the LSEG's canonic Frame.
- (F2c) Regardless of how the linker defines the symbol, there is an associated Group. And the linker specifies the canonic Frame of the Group. (The Group Index subfield of the PUBDEF record specifies the Group.)

(F4) No *X*. The Frame is the canonic Frame of the LSEG that contains Location.

(F5) No *X*. The Target determines the Frame. There are three cases:

- (F5a) The Target specifies a Segment Index: in this case, the Frame is determined as in (F0).
- (F5b) The Target specifies a Group Index: in this case, the Frame is determined as in (F1).
- (F5c) The Target specifies an External Index: in this case, the Frame is determined as in (F2).

The nomenclature that describes frames is similar to the above nomenclature for targets.

Frame: SI (*Segment Name*)	[F0]
Frame: GI (*Group Name*)	[F1]
Frame: EI (*Symbol Name*)	[F2]
Frame: *Location*	[F4]
Frame: *Target*	[F5]
Frame: None	[F6]

For an 8086 memory reference, the Frame specified by a self-relative reference is usually the canonic Frame of the LSEG that contains the Location. Also, the Frame specified by a segment-relative reference is the canonic Frame of the LSEG that contains the Target.

22.3.1 Self-relative Fixup

A self-relative fixup works as follows: Location implicitly defines a memory address — namely, the address of the byte following Location (because at the time of a self-relative reference, the 8086 IP (Instruction Pointer) is pointing to the byte following the reference).

For 8086 self-relative references, if either the Location or the Target is outside the specified Frame, the linker gives a warning. Otherwise, there is a unique 16-bit displacement that, when added to the address implicitly defined by Location, yields the relative position of the Target in the Frame.

If the Location is an offset, the linker adds the displacement to Location (modulo 65,536) and reports no errors.

If the Location is a lobyte, the displacement must be within the range {–128:127}; otherwise, the linker gives a warning. The linker adds the displacement to Location (modulo 256).

If the Location is a base, pointer, or hibyte, it is unclear what the translator intended, so the linker's action is undefined.

22.3.2 Segment-relative Fixup

A segment-relative fixup operates as follows: a non-negative 16-bit number, FBVAL, is defined as the Frame Number of the Frame or selector value that the fixup specifies. For protected-mode applications, FBVAL should be the same as the selector of the Target. A signed 20-bit number, FOVAL, is defined as the distance from the base of the Frame to the Target. If this signed 20-bit number is less than 0 or greater than 65,535, the linker reports an error. Otherwise, the linker uses FBVAL and FOVAL to fix up Location in the following fashion:

- If the Location is a pointer, the linker adds FBVAL (modulo 65,536) to the high-order word of pointer, and adds FOVAL (modulo 65,536) to the low-order word of pointer.
- If the Location is a base, the linker adds FBVAL (modulo 65,536) to the base and ignores FOVAL.
- If the Location is an offset, the linker adds FOVAL (modulo 65,536) to the offset and ignores FBVAL.
- If the Location is a hibyte, the linker adds (FOVAL/256) (modulo 256) to the hibyte and ignores FBVAL. (The division indicated is integer division; that is, the linker discards the remainder.)
- If the Location is a lobyte, the linker adds (FOVAL modulo 256) (modulo 256) to the lobyte and ignores FBVAL.

22.4 Record Sequence

A object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. The following syntax shows the valid record ordering necessary to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

Note

The syntactic description language used in the following syntax is defined in *WIRTH: CACM*, November 1977, vol. 20, no. 11, pp. 822–823. The character strings represented by capital letters are not literals but identifiers, and are further defined in the record format section.

```
object file  = tmodule

tmodule      = {THEADR | LHEADR} seg-grp   {component}   modtail

seg_grp      = {LNAMES} {SEGDEF} {EXTDEF | GRPDEF}

component    = data | debug_record

data         = content_def | thread_def |
                    PUBDEF | EXTDEF | COMDEF | LOCSYM

debug_record = LINNUM

content_def  = data_record {FIXUPP}

thread_def   = FIXUPP (containing only Thread fields)

data_record  = LIDATA | LEDATA

modtail      = MODEND
```

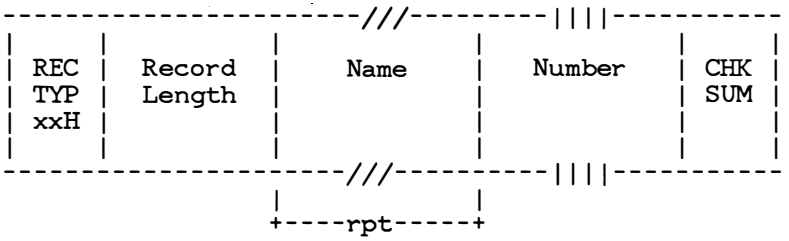
The following rules apply:

- A FIXUPP record always refers to the previous Data record.
- All LNAMES, SEGDEF, GRPDEF, and EXTDEF records must precede all records that refer to them.
- Comment records may appear anywhere in a file, except as the first or last record in a file or module, or within a content_def.

22.5 Introducing the Record Formats

The following pages present diagrams of record formats in schematic form. Here is a sample record format that illustrates the various conventions:

22.5.1 Sample Record Format (SAMREC)



The Title and Official Abbreviation

At the top of the figure is the name of the record format described, with its official abbreviation. To promote uniformity among various programs, including translators and debuggers, use the abbreviation in both code and documentation. The record format abbreviation is always six letters.

The Boxes

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes each represent two bytes. The wide boxes with three slashes in the top and bottom represent a variable number of bytes, one or more, depending upon content. The wide boxes with four vertical bars in the top and bottom represent four-byte fields.

RECTYP

The first byte in each record contains a value between 0 and 255, indicating the record type.

Record Length

The second field in each record contains the number of bytes in the record, exclusive of the first two fields, where a field is a 16-bit number—a low byte followed by a high byte.

Name

Any field that indicates a name has the following internal structure: the first byte contains a number between 0 and 127, inclusive, indicating the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to a subset of the ASCII character set.

Number

A four-byte number field represents a 32-bit unsigned integer, where the first eight bits (least-significant) are stored in the first byte (lowest address), the next eight bits are stored in the second byte, and so on.

Repeated or Conditional Fields

Some portions of a record format contain a field or series of fields that may be repeated one or more times. Such portions are indicated by the "repeated" or "rpt" brackets below the boxes.

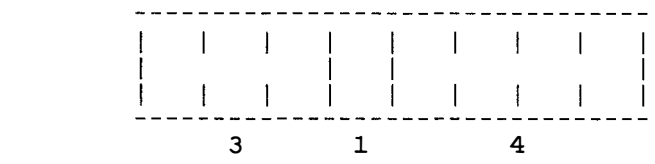
Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

CHKSUM

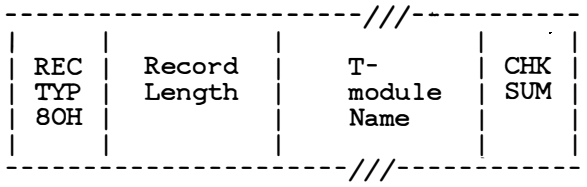
The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals zero.

Bit Fields

Sometimes descriptions of contents of fields are at the bit level. Boxes with vertical lines drawn through them represent bytes or words; the vertical lines indicate bit boundaries. Thus, the following byte representation has three bit fields of three, one, and four bits.



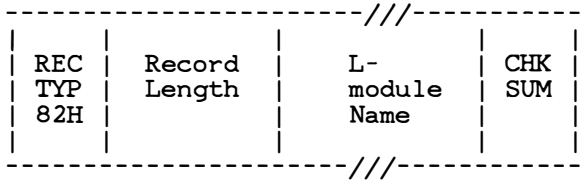
22.5.2 T-module Header Record (THEADR)



T-module Name

The T-module Name is a name for the T-module.

22.5.3 L-module Header Record (LHEADR)

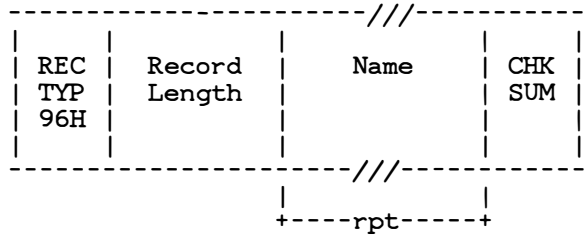


L-module Name

The L-module Name is a name for the L-module.

Every module output from a translator must have a T-module or L-module header record. The linker requires a THEADR or LHEADR record to come first in the module and ignores any others. The LHEADR record is identical to the THEADR record, except it has a record type of 82H.

22.5.4 List of Names Record (LNAMES)



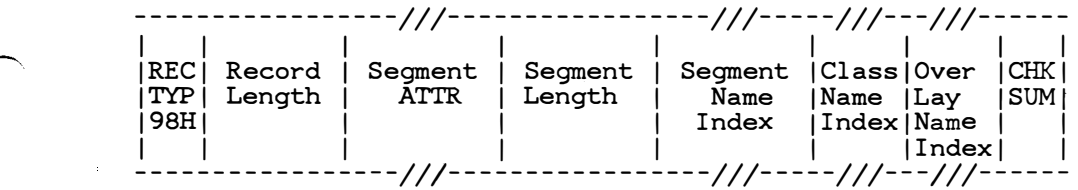
This record contains a list of names that the following SEGDEF and GRPDEF records may use as the names of Segments, Classes, and/or Groups.

The order of LNAMES records in a module and the order of names within each LNAMES record imply a mapping of these names to numbers: 1, 2, 3, etc. These numbers are used as "Name Indices" in the Segment Name Index, Class Name Index, and Group Name Index fields of the SEGDEF and GRPDEF records.

Name

This repeatable field provides a Name, which may have zero length.

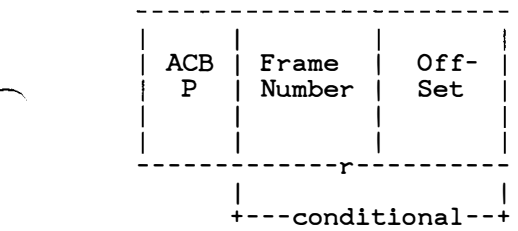
22.5.5 Segment Definition Record (SEGDEF)



Segment Index values 1 through 32,767, which are used in other record types to refer to specific LSEGs, are defined implicitly by the sequence in which SEGDEF records appear in the object file.

SEG ATTR

The SEG ATTR field provides information on various attributes of a Segment, and has the following format:



The ACBP byte contains four numbers: the A, C, B, and P attribute specifications. This byte has the following format:

A (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

A=0 SEGDEF describes an absolute LSEG.

A=1 SEGDEF describes a relocatable, byte-aligned LSEG.

A=2 SEGDEF describes a relocatable, word-aligned LSEG.

A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.

A=4 SEGDEF describes a relocatable, page(256-byte)-aligned LSEG.

If A=0, the Frame Number and Offset fields are present. With the Microsoft linker, you may use absolute segments for addressing only; for example, to define the starting address of a ROM and to define Symbolic Names for addresses within the ROM. The linker ignores any data that belongs to an absolute LSEG, and issues a warning if absolute segments are defined for a program that runs in protected mode.

C (Combination) is a 3-bit subfield that specifies the Combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,3,4,5,6, or 7) that indicates how the segment can be combined. One way to interpret this attribute is to consider how two LSEGs are combined.

For example, suppose that X and Y are LSEGs, and that Z is the LSEG resulting from the combination of X and Y. Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY. Now, to accommodate the alignment attribute of Y, let G be the length of any gap required between the X and Y components of Z. Then, let LZ denote the length of the (combined) LSEG Z; let dx (0<=dx<LX) be the offset in X of a byte, and similarly, let dy be the offset (of a byte) in Y.

The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X, and to dy in Y.

Table 22.2
Combination Attribute Example

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	"Public"
5	LX+LY+G	dx	dy+LX+G	"Stack"
6	MXY	dx	dy	"Common"

Table 7.2 has no lines for C=0, C=1, C=3, C=4, or C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated the same as C=2.

B (Big) is a 1-bit subfield, which, if set to 1, indicates that the Segment Length is exactly 64K (65,536 bytes). In this case, the Segment Length field must contain zero.

The P field must always be zero.

The Frame Number and Offset fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. Offset is in the range between 0 and 15, inclusive. If you want an offset value larger than 15, you should adjust the Frame Number.

Segment Length

The Segment Length field gives a segment's length in bytes. This length may be zero. If it is, the linker does not delete the segment from the module. The Segment Length field is only large enough to hold numbers from 0 to 64K–1, inclusive. To give the segment a length of 64K, you must use the B attribute bit in the ACBP field (see SEG ATTR in this section).

Segment Name Index

The Segment Name is a name that a programmer or translator assigns to the segment, for example: CODE, DATA, TAXDATA, MODULENAME_CODE, or STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES record.

Class Name Index

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has a length of zero. The purpose of a Class Name is to let the programmer define a "handle" to order the LSEGs in MAS, for example: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name by indexing into the list of names provided by the LNAMES record.

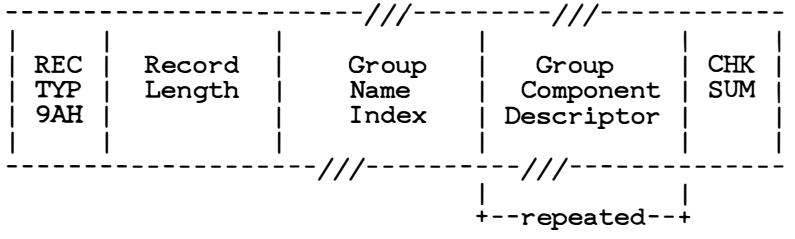
Overlay Name Index

The Overlay Name is a name that the translator and/or the linker, at the programmer's request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name by indexing into the list of names provided by the LNAMES record.

Note

Microsoft language linkers (versions 3.00 and later) ignore the Overlay Name Index, but the standard MS-DOS linker supports it.

22.5.6 Group Definition Record (GRPDEF)



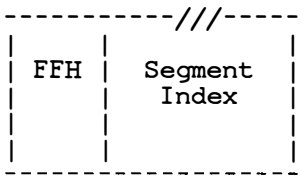
Group Name Index

The linker may reference a collection of LSEGs with the Group Name. Most importantly, when the LSEGs are eventually fixed in MAS, a Frame must exist that "covers" every LSEG of the Group.

The Group Name Index field provides the Group Name by indexing into the list of names provided by the LNAMES record.

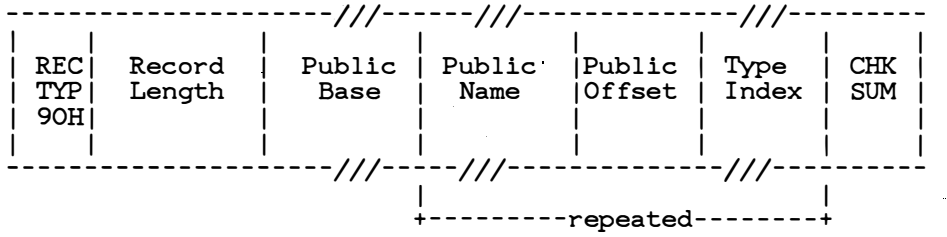
Group Component Descriptor

Each Group Component Descriptor has the following format:



The first byte of the Descriptor contains 0FFH; the Descriptor contains one field, which is a Segment Index that selects the LSEG described by a preceding SEGDEF record.

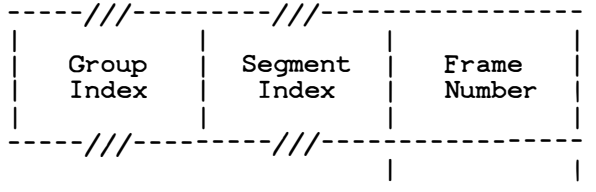
22.5.7 Public Names Definition Record (PUBDEF)



This record provides a list of one or more Public Names. For each name, three kinds of data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

Public Base

The Public Base has the following format:



+conditional+

The Group Index field has a format given earlier, and provides a number between 0 and 32,767, inclusive. A nonzero Group Index associates a group with the public symbol, and is used as described in Section 7.8, "Conceptual Framework for Fixups," case (F2c). A zero Group Index indicates that there is no associated group.

The Segment Index field has a format given earlier, and provides a number between 0 and 32,767, inclusive.

A nonzero Segment Index selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a nonnegative displacement (given by a Public Offset field) from the first byte of the selected LSEG. Also, the Frame Number field must be absent.

A Segment Index of 0 means that the defined symbols are absolute, and the absolute addresses of the symbols are the values in the Public Offset field. The Group Index is ignored.

The Frame Number is present only if the Segment Index is zero. The linker ignores this Frame Number.

A nonzero Group Index selects some group. This group is taken as the "frame of reference" for references to all public symbols defined in this record. That is, the linker performs the following actions:

- The linker converts any fixup of the form:

  Target: EI(P)

  Frame: Target

  (where "P" is a public symbol in this PUBDEF record) to a fixup of the form:

  Target: SI(L),d

  Frame: GI(G)

  where "SI(L)" and "d" are provided by the Segment Index and Public Offset fields. (The "normal" action would have the frame specifier in the new fixup be the same as in the old fixup: Frame: Target.)
- When the linker converts the value of a public symbol, as defined by the Segment Index, Public Offset, and (optionally) Frame Number fields, to a {base,offset} pair, the base part is the base of the indicated group.

A Group Index of zero selects no group. The linker does not alter the Frame specification of fixups referencing the symbol, and takes, as the base part of the absolute value of the public symbol, the canonic Frame of the Segment (either LSEG or PSEG) determined by the Segment Index field.

Public Name

The Public Name field gives the name of the object whose location in MAS the linker makes available to other modules. The name must contain one or more characters.

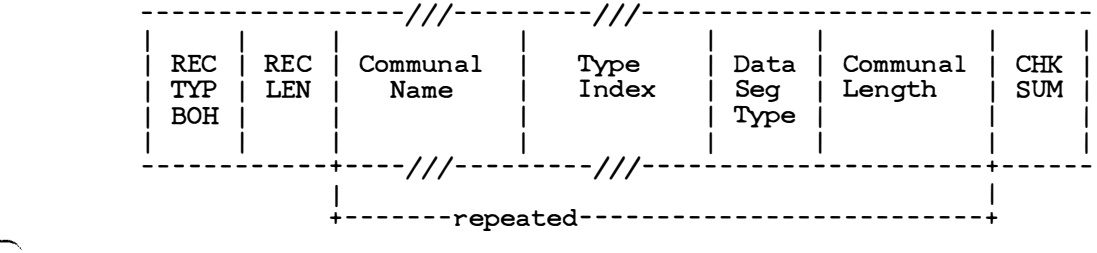
Public Offset

The Public Offset field is a 16-bit value. It is either the offset of the public symbol with respect to an LSEG (if the Segment Index > 0), or the offset of the public symbol with respect to the specified Frame (if the Segment Index $= 0$).

Type Index

The Type Index field identifies a single preceding TYPDEF (Type Definition Record), which contains a descriptor for the type of entity represented by the public symbol. The linker ignores this field.

22.5.8 Communal Names Definition Record (COMDEF)



This record provides a list of one or more Communal Names, which define communal variables. A communal variable is an unitialized public variable whose final size and location are not fixed during compiling.

Communal variables are similar to FORTRAN common blocks in that if you are linking object modules that each declare a communal variable, then the size of that variable is the largest of the declared variables. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char    foo[4];         /* In file a.c */
char    foo[1];         /* In file b.c */
char    foo[1024];      /* In file c.c */
```

If the objects produced from *a.c*, *b.c*, and *c.c* are linked together, the linker allocates 1024 bytes for the character array "foo."

Note

This record requires that a COMENT record from the Microsoft Extension class appear before it in the object module.

Communal Name

This field gives the communal variable name, and must contain one or more characters.

Communal Names are treated as External Names when an External Name is requested elsewhere in the module. Communal Names are ordered together with External Names for the purpose of referring to an External Name by its index. (See the description of the EXTDEF record later in this section for more details on External Names.)

Type Index

This field is ignored by the Microsoft linker.

Data Segment Type

The Data Segment Type field is a single byte that describes the data segment in which the communal variable resides. It can contain the following values:

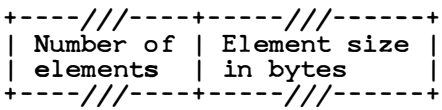
62H (NEAR) = the communal variable is in the default data segment.
61H (FAR) = the communal variable is not in the default data segment.

Communal Length

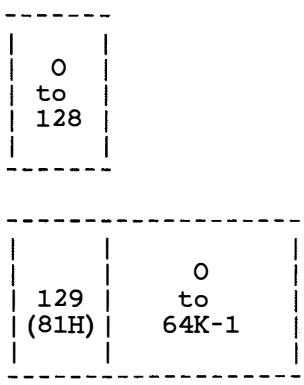
The Communal Length field describes the length of the communal variable according to its data segment type.

If its value is NEAR (62H), the field represents the length in bytes.

If its value is FAR (61H), the field represents:



The format of all the length fields is as follows:



132 (84H)	0 to 16M-1

136 (88H)	-2G-1 to 2G-1

The first format (single byte), containing a value between 0 and 127, represents the number given.

The second format, with a leading byte containing 129, represents the number contained in the following two bytes.

The third format, with a leading byte containing 132, represents the number contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents the number contained in the following four bytes with its sign extended if necessary.

Link-time Semantics:

A PUBDEF matching a communal variable definition overrides the communal variable definition. Two communal variable definitions match if the names in their definitions match. If two matching definitions disagree whether a communal variable is NEAR or FAR, the linker assumes the variable is NEAR.

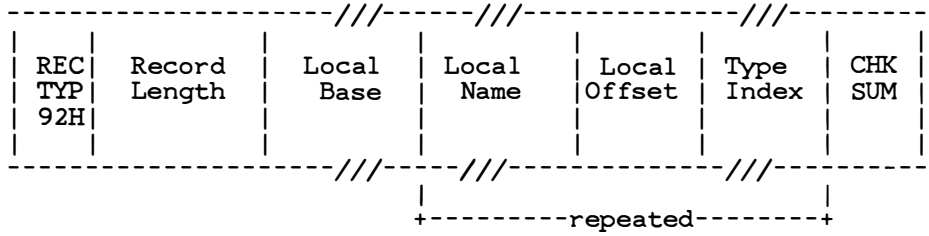
If the variable is NEAR, then its size is the largest of the sizes specified for it. If the variable is FAR, the linker issues a warning if the array element sizes conflict. If these sizes don't conflict, the variable's size is the element size multiplied by the largest number of elements specified. In addition, the sum of the sizes of all NEAR variables must not exceed 64K bytes, and the sum of the sizes of all FAR variables must not exceed the size of the machine's addressable memory space.

HUGE Communal Variables:

A FAR communal variable that is larger than 64K bytes (a HUGE communal variable) resides in segments that are contiguous (on an 8086) or that have consecutive selectors (on an 80286). No other data items reside in the segments occupied by a HUGE communal variable.

If the linker finds matching HUGE and NEAR communal variable definitions, it issues a warning message, since it is impossible for a NEAR variable to be larger than 64K bytes.

22.5.9 Local Symbols Record (LOCSYM)



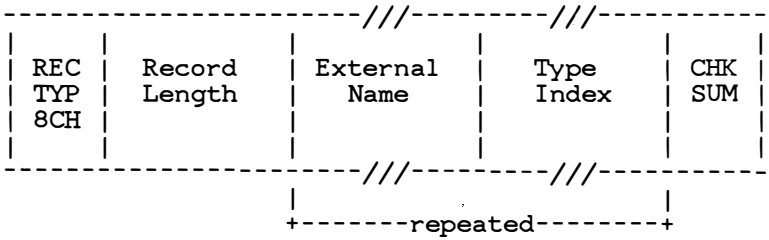
This record provides information for the definition of a local symbol, one that is visible only within the module in which it is defined.

The form and meaning of each of the fields is identical to those in the PUBDEF record.

Note

The LOCSYM record requires that a COMENT record from the Microsoft Extensions class appear before it in the object module. Also, it is only recognized by Microsoft language linkers later than version 3.07.

22.5.10 External Names Definition Record (EXTDEF)



This record provides a list of External Names, and for each name, the type of object it represents. The linker assigns to each External Name the value provided by an identical Public Name or Local Name (if such a name is found).

External Name

This field provides the external object name, which must have nonzero length.

Including a Name in an External Names record is an implicit request to link the object file to a module containing the same name declared as a public symbol, unless the name is defined as a local symbol within the same module as the EXTDEF. This request determines whether the External Name is referenced within some FIXUPP record in the module.

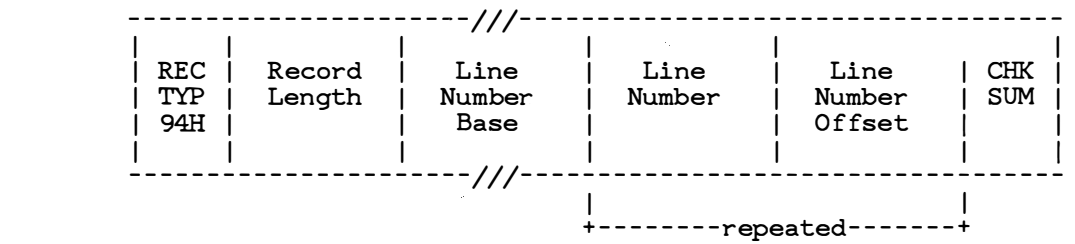
The order of EXTDEF records in a module and the order of External Names within each EXTDEF record, together with COMDEF records and Communal Names, implies a mapping on the set of all External Names requested by the module; for example: 1, 2, 3, etc. So to refer to a particular External Name, the linker uses these numbers as "External Indices" in the Target Datum and/or Frame Datum fields of FIXUPP records.

External indices may not reference forward. For example, an External Definition Record defining the *k*th object must precede any record referring to that object with index *k*.

Type Index

This field is ignored by the Microsoft linker, except for linker versions earlier than 3.05, and for object modules lacking the COMENT record from the Microsoft Extensions class, in which case, refer to Section 7.13 "Microsoft Type Representation for Communal Variables."

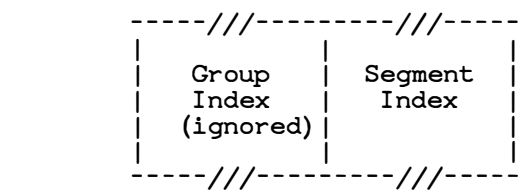
22.5.11 Line Numbers Record (LINNUM)



This record allows a translator to relate a line number in source code to the corresponding line in translated code.

Line Number Base

The Line Number Base has the following format:



The Segment Index determines the location of the first byte of code corresponding to some source line number.

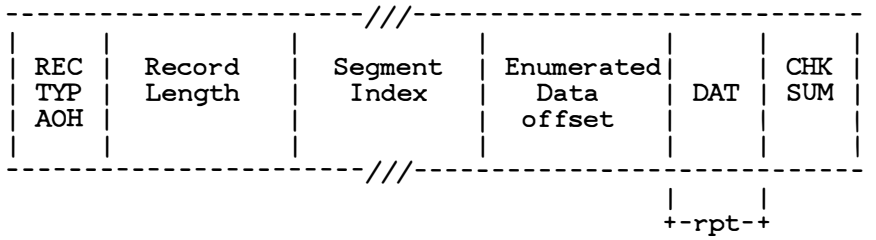
Line Number

This field provides a binary line number between 0 and 32,767, inclusive. If the high-order bit is not zero, the number is considered undefined.

Line Number Offset

The Line Number Offset field is a 16-bit value, which is the offset of the line number with respect to an LSEG (if the Segment Index > 0).

22.5.12 Logical Enumerated Data Record (LEDATA)



This record provides contiguous data from which the linker may construct a portion of an 8086 memory image.

Segment Index

This field, which must be nonzero, specifies an index relative to the Segment Definition Records that precede the LEDATA record.

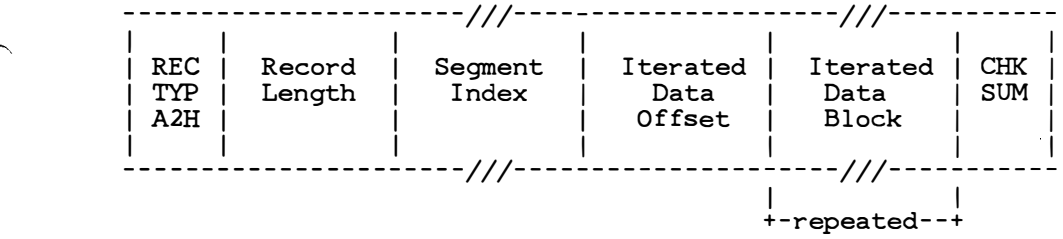
Enumerated Data Offset

This field specifies an offset that is relative to the base of the LSEG specified by the Segment Index. The field also defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

DAT

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

22.5.13 Logical Iterated Data Record (LIDATA)



This record provides contiguous data from which the linker may construct a portion of an 8086 memory image.

Segment Index

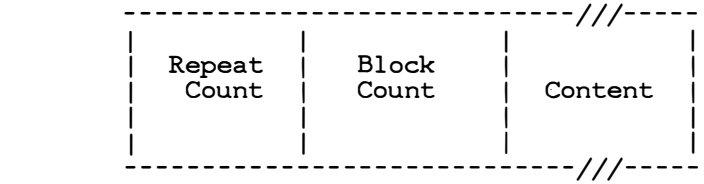
This field, which must be non-zero, specifies an index that is relative to the SEGDEF records that precede the LIDATA record.

Iterated Data Offset

This field specifies an offset that is relative to the base of the LSEG specified by the Segment Index. It also defines the relative location of the first byte in the Iterated Data Block. Successive data bytes in the Iterated Data Block occupy successively higher locations of memory.

Iterated Data Block

This repeated field is a structure specifying the repeated data bytes. It has the following format:



Note

The linker cannot handle LIDATA records whose iterated Data Blocks are larger than 512 bytes.

Repeat Count

This field specifies the number of times to repeat the Content portion of this Iterated Data Block. Repeat Count must be nonzero.

Block Count

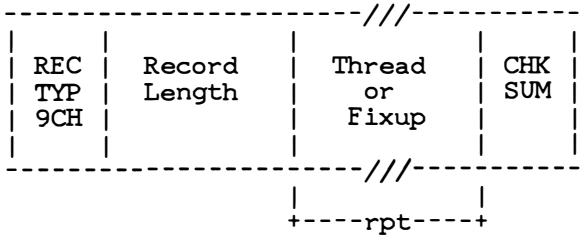
This field specifies the number of Iterated Data Blocks in the Content portion of this Iterated Data Block. If this field has a value of zero, the Content portion of the Iterated Data Block is interpreted as data bytes. If the field is nonzero, the Content portion is interpreted as that number of Iterated Data Blocks.

Content

This field may be interpreted in one of two ways, depending on the value of the previous Block Count field.

If Block Count is zero, this field is a one-byte count followed by the indicated number of data bytes. But if Block Count is nonzero, this field is interpreted as the first byte of another Iterated Data Block.

22.5.14 Fixupp Record (FIXUPP)

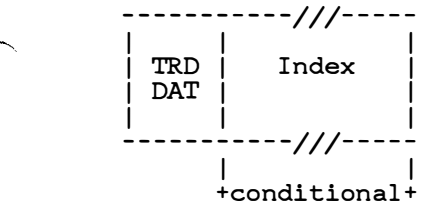


This record specifies zero or more fixups. Each fixup requests a modification (fixup) to a Location within the previous Data record. A data record may be followed by more than one fixup record that refers to it. Each fixup is specified by a Fixup field that specifies four kinds of data: a Location, a Mode, a Target, and a Frame. The Frame and Target may be specified completely within the Fixup field, or by reference to a preceding Thread field.

A Thread field specifies a default Target or Frame that subsequently may be referred to. Eight Threads are provided—four for Frame specification and four for Target specification. Once a Thread has specified a Target or Frame, any Fixup fields that follow (in the same or following FIXUPP records) may refer to that Target or Frame until another Thread field with the same type (Target or Frame) and Thread Number (0–3) appears (in the same or in another FIXUPP record).

Thread

Thread is a field with the following format:



The TRD DAT (Thread Data) subfield is a byte with the following internal structure:

The D subfield is one bit and defines the type of Thread being used. If D=0, this bit defines a Target Thread, and if D=1, it defines a Frame Thread.

Method is a 3-bit subfield containing a number between 0 and 3 (if D=0) or a number between 0 and 6 (if D=1).

If D=0, then Method = (0, 1, 2, 4, 5, 6) mod 4, where 0, 1, 2, 4, 5, 6 indicate methods T0, T1, T2, T4, T5, and T6 of specifying a Target. Thus, Method indicates the kind of Index or Frame Number required to specify the Target, without indicating whether the Target is specified by a primary or secondary method.

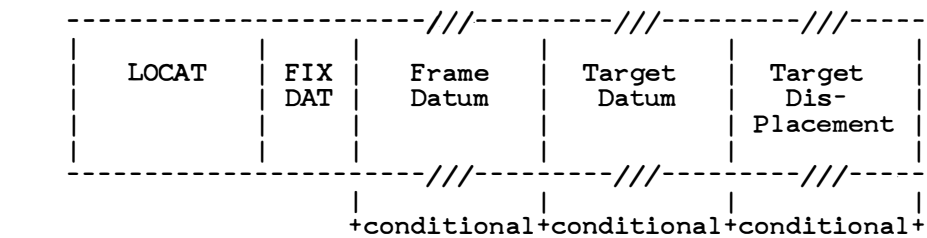
If D=1, then Method = 0, 1, 2, 4, 5 corresponds to methods F0, F1, F2, F4, F5 of specifying a Frame. Here, Method indicates the kind (if any) of Index required to specify the Frame.

THRED is a number between 0 and 3, and associates a Thread Number to the Frame or Target defined by the Thread field.

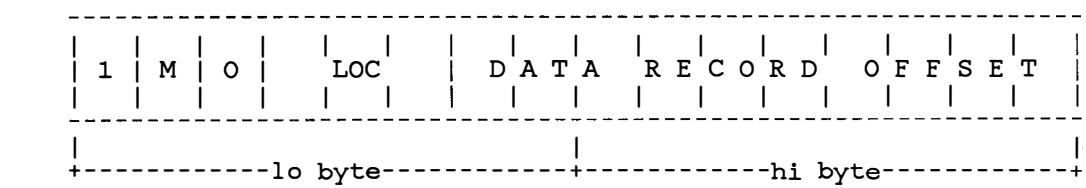
Index contains a Segment Index, Group Index, or External Index depending on the specification in the Method subfield. If Method specifies F4 or F5, this subfield will not be present.

Fixup

Fixup is a field with the following format:



LOCAT is a byte pair with the following format:



M is a 1-bit subfield that specifies the mode of the fixups: self-relative (if M=0) or segment-relative (if M=1).

Note

Self-relative fixups may not be applied to LIDATA records.

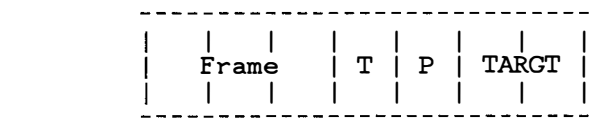
LOC is a 3-bit subfield indicating that the byte(s) in the preceding Data record to be fixed up are a lobyte (if LOC=0), an offset (if LOC=1), a base (if LOC=2), a pointer (if LOC=3), a hibyte (if LOC=4), or a "loader-resolved" offset (if LOC=5). Any other values in LOC are invalid.

The Data Record Offset is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of Location (the actual bytes being fixed up) within the preceding Data record. The Data Record Offset is relative to the first byte in the data fields in the Data records.

Note

If the preceding Data record is an LIDATA record, it is possible for the Data Record offset value to designate a location within a Repeat Count subfield or a Block Count subfield of the Iterated Data field. Such a reference is an error. The linker's action on such a malformed record is undefined.

FIX DAT is a byte with the following format:



F is a 1-bit subfield that specifies whether the frame for this Fixup is specified by a Thread (if F=1) or explicitly (if F=0).

Frame is a number interpreted in one of two ways, as indicated by the F bit. If F is zero, Frame is a number between 0 and 5 and corresponds to methods F0,...,F5 for specifying a Frame. If F=1, then Frame is a Thread Number (0–3). It specifies the Frame most recently defined by a Thread field that defined a Frame Thread with the same Thread Number. (Note

that the Thread field may appear in the same FIXUPP record, or in an earlier one.)

T is a 1-bit subfield that specifies whether the Target specified for this fixup is defined by reference to a Thread (T=1), or is given explicitly in the FIXUP field (T=0).

P is a 1-bit subfield that indicates whether the target is specified by a primary method (requires a Target Displacement, if P=0) or by a secondary method (requires no Target Displacement, if P=1). Since a Target Thread does not have a primary/secondary attribute, the P bit is the only field that specifies the Target specification attribute.

TARGT is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, T1, T2 or T4, T5, T6, depending on the value of P (where P is interpreted as the high-order bit of T0, T1, T2, T4, T5, or T6). When a Thread specifies the Target (if T=1), then Target specifies a Thread Number (0–3).

Frame Datum is the "referent" portion of a Frame specification, and is a Segment Index, Group Index, or External Index. The Frame Datum subfield is present only when the Frame is not specified by a Thread (if F=0) or explicitly by methods F4, F5, or F6.

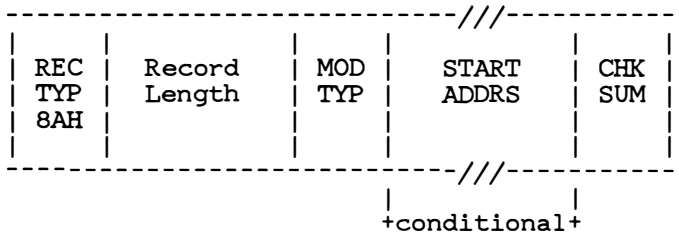
Target Datum is the "referent" portion of a Target specification, and is a Segment Index, Group Index, or External Index. The Target Datum subfield is present only when a Thread does not specify the Target (if T=0).

Target Displacement is the 2-byte displacement required by "primary" methods of specifying Targets. This 2-byte subfield is present if P=0.

Note

All these methods are described in Section 7.8, "Conceptual Framework for Fixups."

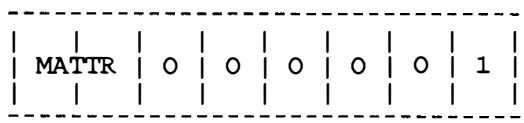
22.5.15 Module End Record (MODEND)



This record serves two purposes. It denotes the end of a module and indicates whether the module that just ended specifies an entry point to begin execution. If it does not, the linker specifies the execution address.

MOD TYP

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:

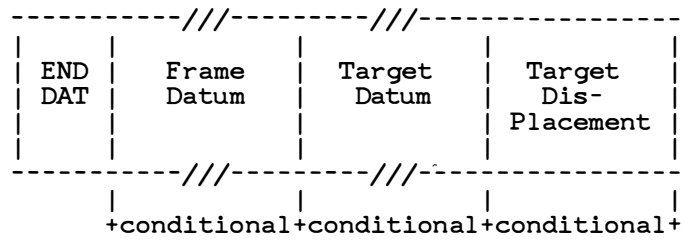


MATTR is a 2-bit subfield that specifies the following module attributes:

MATTR	Module Attribute
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

the following format:

START ADDRS



The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in the same manner as mapping any other logical address to a physical address, as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, Frame Datum, Target Datum, and Target Displacement fields in the FIXUPP record. Only primary fixup methods are allowed. Frame method F4 is not allowed.

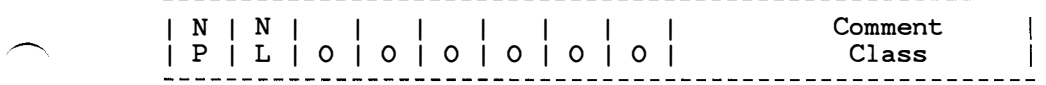
22.5.16 Comment Record (COMENT)

This record allows translators to include comments in object text.

Comment Type

This field indicates the type of comment that this record carries, allowing you to structure comments for processes that selectively act on comments.

The format of this field is as follows:



The NP (NOPURGE) bit, if set to 1, indicates that this comment cannot be purged by object file utility programs that can delete Comment records.

The NL (NOLIST) bit, if set to 1, indicates that the text in the Comment field should not appear in the listing file of object file utility programs that can list object Comment records.

The Comment Class subfield is a byte defined as follows:

0	Language Translator Comment (obsolete). If the comment field contains one of the strings "MS PASCAL" or "FORTRAN 77," then the comment record enables the **dsallocation** switch on the Microsoft linker.
156(9CH)	DOS Version. The Comment field contains a 2-byte integer that specifies the DOS level number.
157(9DH)	Memory Model. Indicates the memory model of the object module. The Comment field contains a single byte with the values s, m, l, or h, for small, medium, large, or huge, respectively. This comment record is used only by the Microsoft XENIX linker.
158(9EH)	Force Segment Ordering. Causes the linker to use a special segment ordering for executable files. This comment record has the same effect as giving the **dosseg** switch to Microsoft language versions of the linker.
159(9FH)	Library Specifier.
129(81H)	Library Specifier (obsolete). Specifies a library to add to the Microsoft linker's library search list. The Comment field contains the name of the library. Note that unlike all other name

specifications, the library name is not prefixed with its length but is determined by the record length. The **nodefaultlibrarysearch** switch causes the linker to ignore these comment records. The 159(9FH) class record is ignored by XENIX versions of the Microsoft linker.

160(A0H) Dynamic Link Record. For details, see Section 7.14 "Microsoft Extensions for Dynamic Linking."

161(A1H) Microsoft Extensions.

Indicates that the object module contains extensions to the original Microsoft adaptation of the INTEL Relocatable Object Module Format, such as the COMDEF and LOCSYM records.

Comment

This field provides the commentary information.

22.6 Microsoft Type Representations for Communal Variables – Obsolete Method

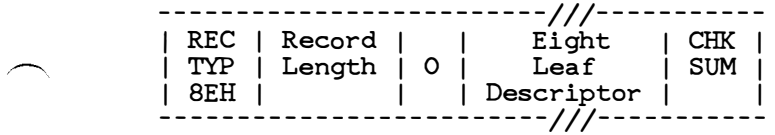
---

Note

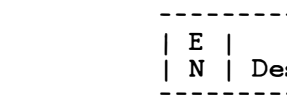
Object modules not containing the COMENT record from the Microsoft Extensions class can represent communal variable definitions only with the obsolete method described here. Also, Microsoft language linkers earlier than version 3.05 can recognize this method only. The newer method uses the Communal Variable Definitions (COMDEF) record.

A communal variable is defined in the object text by an EXTDEF (External Definition) record and the Type Definition record (TYPDEF) to which it refers.

The TYPDEF of a communal variable has the following format:

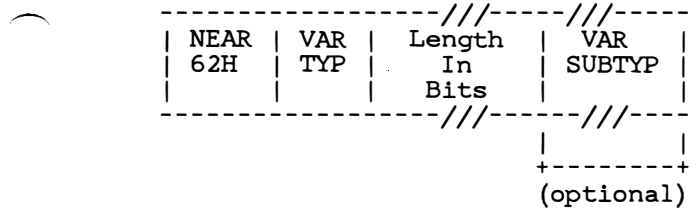


The Eight Leaf Descriptor field has the following format:

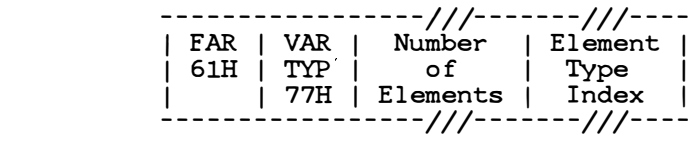


The EN bitfield specifies whether the next eight leaves in the Leaf Descriptor field are EASY (if EN = 0) or NICE (if EN = 1). This byte is always zero for TYPDEFs of communal variables.

The Leaf Descriptor field has one of the following two formats. The format for communal variables in the default data segment (NEAR variables) is as follows:



The VARTYP (Variable Type) field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The linker ignores the VAR SUBTYP field (if one exists). The format for communal variables not in the default data segment (FAR variables) is as follows:



The VARTYP field must be ARRAY (77H). The Length in Bits field specifies the Number of Elements, and the Element Type Index is an index to a previously defined TYPDEF whose format is that of a NEAR communal variable.

The format for the Length in Bits or Number of Elements fields is the same as the format of the Communal Length field of the COMDEF record.

Link-time Semantics:

All EXTDEFs referencing a TYPDEF of one of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected.

For more information, see "Link-time Semantics" under the Communal Names Definition Record (COMDEF) in Section 7.5.8 of this manual.

22.7 Microsoft Extensions for Dynamic Linking

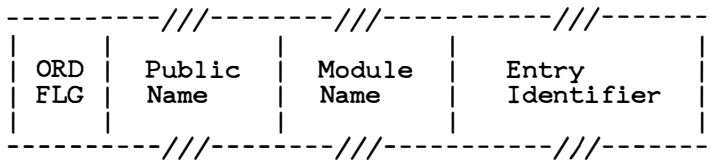
A COMENT record with a class value of 160 (A0H), represents a dynamic-link record, whose data is contained in the Comment field. The first byte of the Comment field contains a value that indicates the type of dynamic-link record.

The valid types are:

Import Definition Record

Defines an imported name for use by versions of the Microsoft linker for Microsoft Windows and MS OS/2. This record is equivalent to an Import definition in the module definitions file for the linker.

The record is contained in the Comment field and has the following format:



Ordinal Flag

This is a single byte field with any nonzero value; meaning that the import is identified by ordinal number and zero, meaning by name.

Public Name

The Public Name field contains the name by which the imported entry is publicly known and that resolves the external reference this record generates. This name is added to the Imported Names table.

Module Name

This field contains the name of the executable module containing the function.

Entry Identifier

This field contains the identity of the entry for the imported name according to the value of the Ordinal Flag:

- If the Ordinal Flag is identified by name, the Entry Identifier is the name of the entry within the module where it is defined. If the name is null, the public name is used.
- If the Ordinal Flag is identified by ordinal, then the Entry Identifier is a 16-bit ordinal value for the entry.

Chapter 23
MS OS/2 Disk Allocation

The MS OS/2 area is formatted as follows:

- Reserved area - variable size
- First copy of File Allocation Table - variable size
- Additional copies of File Allocation Table - variable size (optional)
- Root directory - variable size
- File data area - variable size

Space for a file in the data area is not preallocated, but allocated one *cluster* at a time. A cluster consists of one or more consecutive sectors (the number of sectors in a cluster must be a power of 2). The cluster size is determined at format time. All clusters for a file are "chained" together in the File Allocation Table (FAT). MS OS/2 normally keeps a second copy of the FAT for consistency, unless your system has reliable storage such as a virtual RAM disk. Should the disk develop a bad sector in the middle of the first FAT, MS OS/2 can use the second, avoiding loss of data due to an unreadable FAT.

23.1 MS OS/2 Disk Directory

The **format** command builds the root directory for all disks. The root directory's location on the disk and the maximum number of entries are dependent on the media. Since MS OS/2 regards all subdirectories as files, there is no limit to the number of files that these directories may contain.

All directory entries are 32 bytes in length and are in the following format (note that byte offsets are in hexadecimal):

0–7 Filename. Eight characters, left-aligned and, if necessary, padded with blanks. The first byte of this field indicates the file status, as follows:

- 00H The directory entry has never been used. This is used to limit the length of directory searches, for improved performance.
- 05H Indicates that the first character of the filename contains an E5H character.

2EH The entry is for a directory. If the second byte is also 2EH, the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are spaces, and the cluster field contains the cluster number of this directory.

E5H The file was used, but has since been erased.

Any other character is the first character of a filename.

8–0A Filename extension.

0B File attribute. The attribute byte is mapped as follows (values are in hexadecimal):

01 File is marked read-only. An attempt to open the file for writing using the Open Handle system call (Function Request 3DH) results in an error code being returned. This value can be used in programs along with the other attributes in this list. Attempts to delete the file with the Delete File system call (13H) or Delete Directory Entry (41H) will also fail.

02 Hidden file. The file is excluded from normal directory searches.

04 System file. The file is excluded from normal directory searches.

08 The entry contains the volume label in the first 11 bytes, but contains no other usable information (except date and time of creation). The entry may exist only in the root directory.

10 The entry defines a subdirectory, and is excluded from normal directory searches.

20 Archive bit. The bit is set to "on" whenever the file has been written to and closed.

Note

The system files (*io.sys* and *msdos.sys*) are marked as read-only, hidden, and system files. Files can be marked hidden when they are created. Also, you may change the read-only, hidden, system, and archive attributes through the Get/Set File Attributes system call (Function Request 43H).

0C–15 Reserved.

16–17 The time the file was created or last updated. The hours, minutes, and seconds are mapped into two bytes as follows (bit 7 on the left, 0 on the right):

```
Offset 17H
| H | H | H | H | H | M | M | M |

Offset 16H
| M | M | M | S | S | S | S | S |
```

where:

H is the binary number of hours (0–23).

M is the binary number of minutes (0–59).

S is the binary number of two-second increments.

18–19 The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 19H
| Y | Y | Y | Y | Y | Y | Y | M |

Offset 18H
| M | M | M | D | D | D | D | D |
```

where:

Y is the year, in the range 0–119 (1980–2099).

M is the month, in the range 1–12.

D is the day of the month, in the range 1–31.

1A–1B Starting cluster. The number of the first cluster in the file. Note that the first cluster for data space on all disks is cluster 002. The cluster number is stored with the least significant byte first.

Note

For details about converting cluster numbers to logical sector numbers, see Sections 1.5.1 and 1.5.2.

1C–1F File size in bytes. The first word of this four-byte field is the low-order part of the size.

23.2 File Allocation Table (FAT)

The following information is for system programmers who wish to write installable device drivers. This section explains how MS OS/2 allocates disk space for a file by using the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver is then responsible for locating the logical sector on the disk. Programs *should* use the MS OS/2 file management function calls for accessing files; programs that access the FAT are not guaranteed to be upwardly-compatible with future releases of MS OS/2.

The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk. For disks containing more than 4085 (note that 4085 is the correct number) clusters, a 16-bit FAT entry is used.

The first two FAT entries are reserved. However, the device driver may use the first byte as a FAT ID byte for determining media.

The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). The operating system does not always sequentially write (on the disk) files in the data area. Instead, the system allocates the data area one cluster at a time, skipping over clusters it has already allocated. The first free cluster following the last cluster allocated for that file is the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient use of disk space, because if

you erase old files, you can free enough clusters, which the operating system can then allocate for new files.

Each FAT entry contains three or four hexadecimal characters, depending on whether it is a 12-bit or 16-bit entry:

(0)000	The cluster is unused and available.
(F)FF7	The cluster contains a bad sector if it is not part of any cluster chain. MS OS/2 will not allocate such a cluster. So for its report, the **chkdsk** command counts the number of bad clusters.
(F)FF8–FFF	Indicates the last cluster of a file.
(X)XXX	Any other characters that are the cluster number of the next cluster in the file. The number of the first cluster in the file is in the file's directory entry.

The FAT always begins on the first sector after the reserved sectors. If the FAT is larger than one sector, the sectors are contiguous. The operating system usually writes two copies of the FAT to preserve data integrity. MS OS/2 reads the FAT into one of its buffers, whenever needed (open, read, write, etc.). The operating system also gives this buffer a high priority to keep it in memory as long as possible.

23.2.1 How to Use the FAT (12-bit FAT Entries)

To get the starting cluster of a file, examine its directory entry (in the FAT). Next, to locate each subsequent cluster of the file, follow these steps:

1. Take the cluster number just used and multiply it by 1.5 (each FAT entry is 1.5 bytes in length).

   The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

2. Use a MOV instruction to move the word at the calculated FAT offset into a register.

3. If the last cluster used was an even number, keep the low-order 12 bits of the register by using the AND operator with 0FFFH and the register. If the last cluster used was an odd number, keep the high-order 12 bits by using a SHR instruction to shift the register four bits to the right.

4. If the resulting 12 bits are 0FF8H–0FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by **debug**), follow these steps:

1. Subtract 2 from the cluster number.
2. Multiply the result by the number of sectors per cluster.
3. To this result add the logical sector number of the beginning of the data area.

23.2.2 How to Use the FAT (16-bit FAT entries)

To get the starting cluster of a file, examine its directory entry (in the FAT). Then, to find the next file cluster, follow these steps:

1. Take the cluster number last used and multiply it by 2 (each FAT entry is two bytes).
2. Use a MOV WORD instruction to move the word at the calculated FAT offset into a register.
3. If the resulting 16 bits are 0FFF8–0FFFH, no more clusters are in the file. Otherwise, the 16 bits contain the number of the next cluster in the file.

23.3 MS OS/2 Standard Disk Formats

For multi-sided media, you should arrange the clusters on an MS OS/2 disk to minimize head movement. MS OS/2 then allocates all the space on one track (or cylinder) before moving to the next. It uses the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on, until it has used all the sectors on all the heads of the track.

The formats in Tables 23.1 and 23.2 are standard and should be readable in the appropriate standard drive.

Table 23.1
MS OS/2 Standard Disk Formats

Disk Size in inches	5-¼				8		
Tracks/side	40	40	40	40	77	77	77
WORD bytes/sector	512	512	512	512	128	128	024
BYTE sectors/allocation unit	1	1	2	2	4	4	1
WORD reserved sectors	1	1	1	1	1	4	1
Byte no. FATs	2	2	2	2	2	2	2
WORD root directory entries	64	64	12	12	68	68	92
WORD no. sectors	320	360	640	720	2002	2002	616
BYTE media descriptor	FE	FC	FF	FD	FE*	FD	FE*
WORD sectors/FAT	1	2	1	2	6	6	2
WORD sectors/track	8	9	8	9	26	6	8
WORD no. heads	1	1	2	2	1	1	2
WORD no. hidden sectors	0	0	0	0	0	0	0

*The two media descriptor bytes that are the same for 8" disks (FEH) are not misprints. To establish whether a disk is single- or double-density, a read of a single-density address mark should be made. If an error occurs, the media is double-density.

Table 23.2
MS OS/2 Standard Disk Formats

Disk Size in inches	3-½ or 5-¼				5-¼
Tracks/side	80	80	80	80	80
WORD bytes/sector	512	512	512	512	512
BYTE sectors/allocation unit	2	2	2	2	1
WORD reserved sectors	1	1	1	1	1
BYTE no. FATs	2	2	2	2	2
WORD root dir entries	64	64	112	112	224
WORD no. sectors	640	720	1280	1640	2400
BYTE media descriptor	FA	FB	F8	F9	F9
WORD sectors/FAT	1	2	2	3	7
WORD sectors/track	8	9	8	9	15
WORD no. heads	1	1	2	2	2
WORD no. hidden sectors	0	0	0	0	0

Appendix A
Categorical List of Functions

The categories listed in this appendix are somewhat arbitrary; however, they are intended to group related functions into smaller, more manageable subsets.

Note

The function names are shown in upper- and lowercase letters to make them easy to read. When used in a program, the names must be entered in uppercase letters only.

A.1 Device I/O Functions

Name	Description
DosBeep	Generate Sound from Speaker
DosCLIAccess	Request CLI/STI Privilege
DosDevConfig	Get Device Configuration
DosDevIOCtl	I/O Control for Devices
DosPortAccess	Request Port Access
KbdCharIn	Read Character-Scan Code
KbdClose	Close Logical Keyboard
KbdDeRegister	Deregister Keyboard Subsystem
KbdFlushBuffer	Flush Keystroke Buffer
KbdFreeFocus	Free Keyboard Focus
KbdGetFocus	Get Keyboard Focus
KbdGetStatus	Get Keyboard Status
KbdOpen	Open Logical Keyboard
KbdPeek	Peek at Character-Scan Code
KbdRegister	Register Keyboard Subsystem

KbdSetFgnd	Set Foreground Keyboard Priority
KbdSetStatus	Set Keyboard Status
KbdShellInit	Initialize Keyboard Shell
KbdStringIn	Read Character String
KbdSynch	Synchronize Keyboard Access
KbdXlate	Translate Keyboard Scan Code
VioDeRegister	Deregister Video Subsystem
VioEndPopUp	Deallocate a Popup Display Screen
VioGetAnsi	Get ANSI State
VioGetBuf	Get Logical Video Buffer
VioGetConfig	Get Video Configuration
VioGetCurPos	Get Cursor Position
VioGetCurType	Get Cursor Type
VioGetFont	Get Font Selector
VioGetMode	Get Display Mode
VioGetPhysBuf	Get Physical Video Buffer
VioGetState	Get Video State
VioModeUndo	Restore Mode Undo
VioModeWait	Restore Mode Wait
VioPopUp	Allocate Popup Display Screen
VioPrtSc	Print Screen
VioPrtScToggle	Toggle Print Screen
VioReadCellStr	Read Character-Attribute String
VioReadCharStr	Read Character String
VioRegister	Register Video Subsystem
VioSavReDrawUndo	Undo Screen Save Redraw
VioSavReDrawWait	Wait for Screen Save Redraw
VioScrLock	Lock Screen

VioScrollDn	Scroll Screen Down
VioScrollLf	Scroll Screen Left
VioScrollRt	Scroll Screen Right
VioScrollUp	Scroll Screen Up
VioScrUnLock	Unlock Screen
VioSetAnsi	Set ANSI On or Off
VioSetCurPos	Set Cursor Position
VioSetCurType	Set Cursor Type
VioSetFont	Set Video Font
VioSetMode	Set Display Mode
VioSetState	Set Video State
VioShowBuf	Display Logical Buffer
VioWrtCellStr	Write Character-Attribute String
VioWrtCharStr	Write Character String
VioWrtCharStrAtt	Write Character String with Attribute
VioWrtNAttr	Write *n* Attributes
VioWrtNCell	Write *n* Character-Attributes
VioWrtNChar	Write *n* Characters
VioWrtTTY	Write TTY String

A.2 Device Monitor Functions

Name	Description
DosMonClose	Close Connection to MS OS/2 Device Monitor
DosMonOpen	Open Connection to MS OS/2 Device Monitor
DosMonRead	Read Input from Monitor Stream

DosMonReg	Register Set of Buffers as Monitor
DosMonWrite	Write Output to Monitor Stream

A.3 Dynamic Linking

Name	Description
DosFreeModule	Free Dynamic-link Module
DosGetModHandle	Get Dynamic-link Module Handle
DosGetModName	Get Dynamic-link Module Name
DosGetProcAddr	Get Dynamic-link Procedure Address
DosGetResource	Get Resource Segment Selector
DosLoadModule	Load Dynamic-link Module

A.4 Error Handling

Name	Description
DosErrClass	Classify Error Code
DosError	Enable Hard-error Processing
DosSetVec	Establish Handler for Exception Vector
DosSystemService	DOS System Process Services

A.5 Family API Program Execution Control Functions

Name	Description
BadDynLink	Bad Dynamic Link

DosGetMachineMode	Return Current Processor Mode

A.6 File I/O Functions

Name	Description
DosBufReset	Commit File Buffers/Directory Info
DosChdir	Change Current Directory
DosChgFilePtr	Change File Read/Write Pointer
DosClose	Close File Handle
DosDelete	Delete File
DosDupHandle	Duplicate File Handle
DosFileLock	Manage File Locks
DosFindClose	Close Find Handle
DosFindFirst	Find First Matching File
DosFindNext	Find Next Matching File
DosMkdir	Make Subdirectory
DosMove	Move File
DosNewSize	Change File Size
DosOpen	Open File
DosPhysicalDisk	Support Partitionable Disk
DosQCurDir	Query Current Directory
DosQCurDisk	Query Current Disk
DosQFHandState	Query File Handle State
DosQFileInfo	Query File Information
DosQFileMode	Query File Mode
DosQFSInfo	Query File System Information
DosQHandType	Query Handle Type

DosQVerify	Query Verify Setting
DosRead	Read from File
DosReadAsync	Read from File Asynchronously
DosRmdir	Remove Subdirectory
DosScanEnv	Scan Environment Segment
DosSearchPath	Search Path for Filename
DosSelectDisk	Select Default Drive
DosSetFHandState	Set File Handle State
DosSetFileInfo	Set File Information
DosSetFileMode	Set File Mode
DosSetFSInfo	Set File System Information
DosSetMaxFH	Set Maximum File Handles
DosSetVerify	Set/Reset Verify Switch
DosWrite	Write to File Synchronously
DosWriteAsync	Write to File Asynchronously

A.7 Interprocess Communication Functions: Pipes, Queues, and Semaphores

Name	Description
DosCloseQueue	Close Queue
DosCloseSem	Close System Semaphore
DosCreateQueue	Create Queue
DosCreateSem	Create System Semaphore
DosFlagProcess	Set Process External Event Flag
DosMakePipe	Create Pipe
DosMuxSemWait	Wait for One of *n* Semaphores to be Cleared
DosOpenQueue	Open Queue

DosOpenSem	Open Existing System Semaphore
DosPeekQueue	Peek Queue
DosPurgeQueue	Purge Queue
DosQueryQueue	Query Size of Queue
DosReadQueue	Read from Queue
DosSemClear	Clear Semaphore
DosSemRequest	Request Semaphore
DosSemSet	Set Semaphore Owned
DosSemSetWait	Set Semaphore and Wait for Next Clear
DosSemWait	Wait for Semaphore to be Cleared
DosWriteQueue	Write to Queue

A.8 Memory Management Functions

Name	Description
DosAllocHuge	Allocate Huge Memory
DosAllocSeg	Allocate Segment
DosAllocShrSeg	Allocate Shared Segment
DosCreateCSAlias	Create Code Segment Alias
DosFreeSeg	Free Segment
DosGetHugeShift	Get Shift Count
DosGetSeg	Get Access to Segment
DosGetShrSeg	Access Shared Segment
DosGiveSeg	Give Access to Segment
DosLockSeg	Lock Segment in Memory
DosMemAvail	Get Size of Largest Free Memory Block
DosReAllocHuge	Change Huge Memory Size

DosReAllocSeg	Change Segment Size
DosSubAlloc	Suballocate Memory within Segment
DosSubFree	Free Memory Suballocated within Segment
DosSubSet	Initialize or Set Allocated Memory
DosUnLockSeg	Unlock Segment

A.9 Message Functions

Name	Description
DosGetMessage	Get System Message
DosInsMessage	Insert Variable Text Strings in Message
DosPutMessage	Output Message Text to Handle

A.10 Mouse API Functions

Name	Description
MouClose	Close Mouse Device for Current Screen Group
MouDeRegister	Deregister Mouse Subsystem
MouDrawPtr	Release Screen Area for Device Driver Use
MouFlushQue	Flush Mouse Event Queue
MouGetDevStatus	Get Mouse Status
MouGetEventMask	Get Current Mouse Event Mask
MouGetHotKey	Get System Hot Key Button
MouGetNumButtons	Get Number of Buttons
MouGetNumMickeys	Get Number of Mickeys-per-centimeter
MouGetNumQueEl	Get Current Number of Mouse Queue Elements

MouGetPtrPos	Get Mouse Pointer Coordinates
MouGetPtrShape	Get Mouse Pointer Shape
MouGetScaleFact	Get Mouse Scaling Factors
MouInitReal	Initialize Real-mode Mouse Device Driver
MouOpen	Open Mouse Device
MouReadEventQue	Read Mouse Event Queue
MouRegister	Register Mouse Subsystem
MouRemovePtr	Reserve Screen Area for Application Use
MouSetDevStatus	Set Mouse Driver Status Flags
MouSetEventMask	Set New Event Mask
MouSetHotKey	Set System Hot Key Button
MouSetPtrPos	Set Mouse Pointer Coordinates
MouSetPtrShape	Set Mouse Pointer Shape and Size
MouSetScaleFact	Set Mouse Scaling Factors
MouShellInit	Initialize Mouse Shell
MouSynch	Synchronize Mouse Driver Access

A.11 National Language Support Programming Interface Functions

Name	Description
DosCaseMap	Perform Case-mapping
DosGetCollate	Get Collating Sequence
DosGetCP	Get Process Code Page
DosGetCtryInfo	Get Country-dependent Information
DosGetDBCSEv	Get DBCS Environmental Vector
DosPFSActivate	Activate Font
DosPFSCloseUser	Close Font User Instance

DosPFSInit	Initialize Code Page and Font
DosPFSQueryAct	Query Active Font
DosPFSVerifyFont	Verify Font
DosSetCP	Set Process Code Page
KbdCustCP	Install Custom Translate Table
KbdGetXT	Get Loaded Translate Table ID
KbdSetXT	Set Translate Table
VioGetCP	Get Code Page
VioSetCP	Set Code Page

A.12 Program Startup Functions

Name	Description
DosGetEnv	Get Address of Process Environment String
DosGetVersion	Get DOS Version Number

A.13 Signal Functions

Name	Description
DosHoldSignal	Disable/Enable Signal
DosSetSigHandler	Set Signal Handler

A.14 Tasking Functions

Name	Description
DosCreateThread	Create Another Thread of Execution
DosCWait	Wait for Child Process Termination
DosEnterCritSec	Enter Critical Section of Execution
DosExecPgm	Execute Program
DosExit	Exit Program
DosExitCritSec	Exit Critical Section of Execution
DosExitList	Maintain Routine List for Process Termination
DosGetInfoSeg	Get Address of System Variables Segment
DosGetPrty	Get Process Priority
DosKillProcess	Terminate Process
DosPTrace	Interface for Program Debugging
DosResumeThread	Restart Thread
DosSelectSession	Select Foreground Session
DosSetPrty	Set Process Priority
DosSetSession	Set Session Status
DosStartSession	Start Session
DosStopSession	Stop Session
DosSuspendThread	Suspend Thread Execution

A.15 Timer Functions

Name	Description
DosGetDateTime	Get Current Date and Time

DosSetDateTime	Set Current Date and Time
DosSleep	Delay Process Execution
DosTimerAsync	Start Asynchronous Time Delay
DosTimerStart	Start Periodic Interval Timer
DosTimerStop	Stop Interval or Asynchronous Timer

A.16 Real-mode Mouse Functions

Name	Description
INT 33H Function 0	Install Flag and Reset
INT 33H Function 1	Show Pointer
INT 33H Function 2	Hide Pointer
INT 33H Function 3	Get Position and Button Status
INT 33H Function 4	Set Pointer Position
INT 33H Function 5	Get Button Press Information
INT 33H Function 6	Get Button Release Information
INT 33H Function 7	Set Minimum and Maximum Horizontal Position
INT 33H Function 8	Set Minimum and Maximum Vertical Position
INT 33H Function 9	Set Graphics Pointer Block
INT 33H Function 10	Set Text Pointer
INT 33H Function 11	Read Mouse Motion Counters
INT 33H Function 12	Set User-defined Subroutine
INT 33H Function 13	Turn On Light Pen Emulation
INT 33H Function 14	Turn Off Light Pen Emulation
INT 33H Function 15	Set Mickey/Pixel Ratio
INT 33H Function 16	Turn Off Conditional

INT 33H Function 19	Set Dbl Speed Threshold
INT 33H Function 20	Swap User-defined Subroutine
INT 33H Function 21	Get Mouse State Storage Requirements
INT 33H Function 22	Save Mouse Driver State
INT 33H Function 23	Restore Mouse Driver State