Microsoft® Operating System/2

Device Driver Kit

Guide

Microsoft Corporation

Document No. 060060010-100-000-0887

Contents

1 **Introduction 1**

1.1 About This Guide 3
1.2 Contents of the Device Driver Kit 3
1.3 About the Hardware 4
1.4 What You Need to Know 4
1.5 Simplifying the Development Process 5
1.6 Documentation Changes 5
1.7 Using This Guide 5
1.8 Notational Conventions 6

2 **Hardware and Software Requirements 7**

2.1 Introduction 9
2.2 Development System 9

3 **Using the DOS-resident Debugger 11**

3.1 Introduction 15
3.2 Setting Up the DOS-resident Debugger 15
3.3 Using the DOS-resident Debugger 16
3.4 Symbolic Debug Support 16
3.5 Debugger Command Format 17
3.6 Debugger Commands 23
3.7 Debugging with DBPRT Statements 56
3.8 Debugging with Map Files 59

4 **Automated Tests 63**

4.1 Introduction 65
4.2 Setting Up the Tests 65

5 **Sample Device Drivers 77**

5.1 Introduction 79
5.2 General Development Approach 79
5.3 Sample Serial Device Driver 82
5.4 Sample Disk Device Driver 109

A DOS-resident Debugger Error Messages 147

B References 149

B.1 Applicable Documents 149
B.2 Microsoft Documents 150

C Overview of Automated Tests 151

Glossary 176

Chapter 1
Introduction

1.1 About This Guide 3
1.2 Contents of the Device Driver Kit 3
1.3 About the Hardware 4
1.4 What You Need to Know 4
1.5 Simplifying the Development Process 5
1.6 Documentation Changes 5
1.7 Using This Guide 5
1.8 Notational Conventions 6

1.1 About This Guide

This guide explains how to use the Microsoft® Operating System 2 (MS® OS/2) Device Driver Kit in conjunction with the MS OS/2 SDK to design, develop, and test device drivers for MS OS/2. The hardware and software requirements are explained in this guide. A step-by-step description of the development process is provided to guide you through the following operations:

- Setting up your development system
- Creating device drivers
- Debugging device drivers
- Testing device drivers

When mentioned in this guide, DOS refers to MS OS/2, unless otherwise specified.

1.2 Contents of the Device Driver Kit

The following components make up the Device Driver Kit:

MS OS/2 Device Driver Guide	Documentation describing how to design, develop, and test device drivers for MS OS/2.
Binary Distribution Disks	A set of distribution disks containing the system files necessary to support device driver development. These disks also contain the source code for the sample device drivers described in this guide, as well as two debugging versions of MS OS/2 to aid in the development of device drivers.

1.3 About the Hardware

The 80286 and 80386 microprocessors use a *segmented memory* architecture, in which physical memory is divided into individual segments. Each of these segments is of arbitrary size, up to and including 64K bytes, and permits different types of access. The 80286 and 80386 incorporate a memory management unit on chip to support their segmented scheme, so the software required for the memory management scheme is identical on all 80286- and 80386-based computers.

Intel has published several books on the architecture and operation of the 80286 and 80386 microprocessor chips. See Appendix B, "References," for further information.

1.4 What You Need to Know

This guide is for use by ISVs who want to write device drivers to support particular hardware not supported by standard device drivers. To write a device driver for a new hardware environment, you must be completely familiar with the target hardware. In particular, you need to know how the peripheral device controllers work, how memory is laid out, and how the interrupt controller is used.

In addition, you should thoroughly read and understand the user's and programmer's documentation provided in the MS OS/2 SDK, particularly the *MS OS/2 Device Driver Guide*, which describes the architecture, API, and DOS services available to device drivers. A thorough understanding of these concepts is a prerequisite to writing device drivers.

This guide was written for readers who have a good understanding of

- systems programming principles
- 80286 assembly language
- 80286 machine architecture
- the programming environment

If you are unfamiliar with any of these items, see Appendix B, "References," for a list of additional material that may be helpful in this development process.

1.5 Simplifying the Development Process

To simplify the development process, Microsoft provides the source code for sample working device drivers. This code supports devices compatible with the IBM Personal Computer AT (IBM PC AT) or other computers that are 100 percent compatible with the IBM PC AT. The devices supported by these drivers include a hard disk, floppy disk, and serial adapter. You may be able to use these device drivers to substantially reduce your development time and costs.

1.6 Documentation Changes

Any changes or additions to this guide appear in the *readme.doc* file on Distribution Disk 1. Read this file and make the appropriate changes or additions to the documentation before proceeding.

1.7 Using This Guide

This guide contains the following chapters and appendixes:

Chapter 1, "Introduction," is a general overview of this guide, the Device Driver Kit, and the hardware and software required to run MS OS/2.

Chapter 2, "Hardware and Software Requirements," describes the hardware and software requirements for both your development and target systems.

Chapter 3, "Using the DOS-resident Debugger," describes procedures for debugging your device driver.

Chapter 4, "Automated Tests," describes some procedures for testing your device driver.

Chapter 5, "Sample Device Drivers," describes the sample device driver source code provided on the distribution disks, a general device driver development approach, and sample debugging sessions for each of the sample device drivers.

Appendix A, "DOS-resident Debugger Error Messages," describes the system information and error messages that MS OS/2 generates.

Appendix B, "References," lists additional documents that may be useful in the development process.

Appendix C, "Overview of Automated Tests," provides a flowchart overview for each automated device driver test.

The Glossary defines key terms and expressions used in this guide.

1.8 Notational Conventions

This guide uses the following notational conventions:

Bold	MS OS/2 commands, data types, and options appear in bold type. Bold also sets off sample command lines, keywords, system calls, functions, and literal portions of syntax statements.
Italics	Filenames, variables, and directories appear in italics. The first mention of a technical term or an emphasized term also appears in italics.
`Monospace`	Sample portions of programs and system-generated text appear in monospace type. Monospace type simulates the type that appears on your screen.
SMALL CAPITALS	Names of keys and key sequences appear in small capital letters.
CAPITALS	Environment variables, register names, and logical operators appear in capital letters.

Chapter 2
Hardware and Software Requirements

2.1 Introduction 9
2.2 Development System 9
2.2.1 Required Hardware 9
2.2.2 Optional Hardware 9
2.2.3 Required Software 10

2.1 Introduction

This chapter describes the hardware and software needed to modify and run MS OS/2.

Much of the terminology used in this chapter is specific to the Intel 80286 microprocessor, its architecture, and the hardware components that it uses. To learn more about these terms, refer to the documents listed in Appendix B, "References."

2.2 Development System

The MS OS/2 Device Driver Kit is designed for use with an IBM PC AT or compatible computer as the development system.

2.2.1 Required Hardware

The following hardware components make up the minimum acceptable hardware configuration for the development system:

- an IBM PC AT or compatible computer
- one megabyte of memory
- a formatted 20-megabyte hard disk
- a 5.25-inch, 96 tracks-per-inch (tpi) floppy disk drive

2.2.2 Optional Hardware

The following hardware component is supported, and may be useful during the development process:

- a printer for producing listings

2.2.3 Required Software

The following software products are required for the development process, and are included as part of the MS OS/2 SDK:

- Microsoft C Compiler
- Microsoft Macro Assembler (MASM)
- Microsoft LINK

Chapter 3
Using the DOS-resident Debugger

3.1 Introduction 15
3.2 Setting Up the DOS-resident Debugger 15
3.3 Using the DOS-resident Debugger 16
3.4 Symbolic Debug Support 16
3.5 Debugger Command Format 17
3.5.1 Command Parameters 18
3.5.2 Binary and Unary Operators 20
3.6 Debugger Commands 23
3.6.1 ? – Display Help Menu 26
3.6.2 ? *expression* – Display Expression 27
3.6.3 .? – Display External Commands 29
3.6.4 .a – Display Physical Memory 29
3.6.5 .d – Display Data Structures 31
3.6.6 .h – Display Memory Manager Handle 31
3.6.7 .p – Display Current Process 32
3.6.8 bc – Clear Breakpoints 32
3.6.9 bd – Disable Breakpoints 32
3.6.10 be – Enable Breakpoints 33
3.6.11 bl – List Breakpoints 33
3.6.12 bp – Set Breakpoints 34
3.6.13 c – Compare Memory 35
3.6.14 d – Display Memory 35
3.6.15 db – Display Bytes 36
3.6.16 dd – Display Doublewords 37
3.6.17 dg – Display GDT 37

3.6.18 di – Display IDT 38
3.6.19 dl – Display LDT 39
3.6.20 dt – Display TSS 39
3.6.21 dw – Display Words 40
3.6.22 e – Enter Byte 40
3.6.23 f – Fill Memory 42
3.6.24 g – Go 42
3.6.25 h – Hexadecimal Arithmetic 43
3.6.26 i – Input Byte 44
3.6.27 j – Conditional Execute 44
3.6.28 lg – List Groups 45
3.6.29 lm – List Map 45
3.6.30 ln – List Near 46
3.6.31 ls – List Symbols 46
3.6.32 m – Move Memory 46
3.6.33 o - Output to Port 47
3.6.34 p – Program Trace 47
3.6.35 r – Display Registers 48
3.6.36 s – Search Bytes 50
3.6.37 t – Trace Instructions 51
3.6.38 u – Unassemble Bytes 52
3.6.39 vc – Clear Interrupt Vector 52
3.6.40 vl – List Interrupt Vector 53
3.6.41 vs – Set Interrupt Vector 54
3.6.42 w – Change Map 54
3.6.43 x – Reboot Target System 55
3.6.44 z – Execute Default Command String 55
3.6.45 zl – Display Default Command String 55
3.6.46 zs – Change Default Command String 55
3.7 Debugging with DBPRT Statements 56
3.7.1 Notes on Using DBPRT Statements 58

3.7.2 Examples of Using DBPRT Statements 58
3.8 Debugging with Map Files 59
3.8.1 Reading a Map File 60
3.8.2 Example Debug Session 60

3.1 Introduction

The DOS-resident debugger is used to test and debug your operating system code. The debugger is built into MS OS/2 and interacts with the operator through a serial port and a terminal. Before you can begin debugging your code, you need a special version of MS OS/2 that contains the DOS-resident debugger.

3.2 Setting Up the DOS-resident Debugger

Before you can use the DOS-resident debugger, you must perform the following steps:

1. Delete the *os2dos.com* file from your MS OS/2 bootable disk (that is, a copy of the boot disk provided with the Binary Adaptation Kit).
2. Open and close the floppy disk drive door to ensure that DOS reuses the same space on the floppy disk previously occupied by *os2dos.com.*
3. Copy the *os2dosd.com* file from your distribution disk to the boot disk as *os2dos.com.*

The DOS-resident debugger is now installed on your boot disk.

3.3 Using the DOS-resident Debugger

To use the DOS-resident debugger, first connect your target system COM2 Port to a terminal running at the following recommended settings:

9600 baud
 No parity
  8 data bits
  1 stop bit

Note that the debugging terminal must be connected to COM2 of your target system.

Next, boot the disk you prepared in Section 3.2, "Setting Up the DOS-resident Debugger," on the target system.

The DOS-resident debugger begins by executing an **r** (**display registers**) command, which displays all of the 80286 microprocessor's registers and flags. It then prompts you with a number sign (#), indicating protected mode (a greater-than sign [**>0**] indicates real mode).

For example:

```
MS OS/2 DOS-resident debugger
Copyright 1986, Microsoft
AX=0698  BX=2008  CX=2C18  DX=18AB  SP=1B7A  BP=00FF  SI=0020  DI=10CD
IP=0450  CS=18B0  DS=1BE8  ES=0DA8  SS=0048  NV UP DI PL NZ NA PO NC
GDTR=01BE80 3687  IDTR=01F508 03FF  TR=0010  LDTR=0028 IOPL=3 MSW=PM
18B0:0450 C3             RET
#
```

3.4 Symbolic Debug Support

The DOS-resident debugger supports symbolic debugging. When a symbol file (generated with the **mapsym** program, version 4.0 or greater) is loaded for a base or supplemental device driver, the debugger can use any of the public symbols as part of an expression.

To create a symbol file, perform the following steps:

1. Generate a map file during the linking process. Then, run the map through the **mapsym** program by typing the following command:

   mapsym *mapfile.map*

   where *mapfile.map* is the name of the map file created during the link process. Running the map file through **mapsym** generates a symbol file with the extension **.sym**.

2. **Base device drivers only.** The symbol file for base device drivers must be in the root directory of the boot drive and have the same name as the device driver, but with the **.sym** extension (for example, *clock01.sym*).
3. **Supplemental device drivers only.** The symbol file for supplemental device drivers must be in the same directory as the device driver file and have the same name as the device driver but with the **.sym** extension (for example, *com.sym*).

There can be more than one symbol file loaded at one time, but only one symbol file can be active at once. (For information on controlling and listing the active map file, see the **w** (**change map**) and **lm** (**list map**) commands later in this chapter.)

The following message

Symbols Linked (*xxxxxxxx*)

appears when a symbol file has been successfully loaded.

xxxxxxxx is the symbol map name listed with the **lm** command.

3.5 Debugger Command Format

Each DOS-resident debugger command consists of one or two letters, usually followed by one or more parameters. When typing these commands and parameters, you can use any combination of uppercase and lowercase letters.

Note

Throughout this chapter, an uppercase *L* is used to specify the length option when used as part of the *range* parameter in commands. This is done to avoid confusion with the number 1.

If a syntax error occurs in a DOS-resident debugger command, the debugger reprints the command line and indicates the error with a caret (^) and the word *Error*, as in the following example:

```
A100
^ Error
```

Pressing CONTROL-C at the debug terminal while the target system is executing halts execution of the target system and starts the DOS-resident debugger. At every timer interrupt, the debugger looks for a CONTROL-C. If the debugger finds one, it takes control from the target system.

Pressing CONTROL-S freezes a DOS-resident debugger display; CONTROL-Q restarts the display. CONTROL-S and CONTROL-Q are ignored if the target system is executing code.

Note

If you are using a terminal emulator for your debugging terminal, you may see irregular screen displays. These displays occur because terminal emulation programs store characters in a buffer since they cannot display the characters as fast as they receive them from the debugger. When you press CONTROL-C, the DOS-resident debugger immediately stops sending characters. However, the terminal emulator continues to dump its buffer, giving the impression that CONTROL-C does not work. To avoid any problems this might cause, a real terminal can be used for debugging.

3.5.1 Command Parameters

You can separate DOS-resident debugger command parameters with delimiters (spaces or commas), but the only required delimiter is between two consecutive hexadecimal values. The following commands are equivalent:

```
dCS:100 110
d CS:100 110
d,CS:100,110
```

Note

Selector is the term used to indicate the value in a segment register while in protected mode. *Segment* is the equivalent in real mode. Although the following discussion uses *selector*, the discussion applies to *segments* as well.

The following list describes the parameters you can use with DOS-resident debugger commands:

Parameter	Definition

addr	An address parameter that can be represented in one of two forms. The first form is a two-part designation containing either an alphabetic selector register or a four-digit selector address plus an offset value. The second form is a physical address using the % operator. You can omit the selector name or selector address, in which case the default selector is DS. This default selector is used for all commands except **g**, **p**, **t**, and **u**. The default selector for these commands is CS. All numeric values are hexadecimal. Example addresses include
	`CS:0100`
	`04BA:0100`
	Note that a colon is required between the selector name (whether numeric or alphabetic) and the offset value. The selector portion is treated as a selector or segment as appropriate for the current processor mode (protected or real) unless specifically overridden by the # or & operator (see Section 3.5.2, "Binary and Unary Operators").

byte	A two-digit hexadecimal value.
cmds	An optional set of DOS-resident debugger commands to be executed with the **bp (set breakpoints)** or **j (conditional execute)** commands.
dword	An eight-digit (four-byte) hexadecimal value. A *dword* is most commonly used as a physical address.
expression	A combination of parameters and operators that evaluates to an 8-, 16-, or 32-bit value. *Expressions* can be used as values in any command. An *expression* can combine any symbol, number, or address with any of the binary and unary operators (see Section 3.5.2, "Binary and Unary Operators").
group-name	Specifies the name of a group that contains the map symbols you want to display.
list	A series of byte values or a *string*. *List* must be the last parameter on the command line. Following is an example of the **f** (**fill**) command using a *list*: `fCS:100 42 45 52 54 41`
map-name	Specifies the name of a symbol map file.

range	Contains two addresses (*addr addr*) or one address, an L, and a value (*addr* **L** *word*, where *word* is the number of items on which the command should operate; **L** 80 is the default). Sample *ranges* include
	`CS:100 110`
	`CS:100 L 10`
	`CS:100`
	The limit for *range* is 10000 (hexadecimal). To specify a *word* of 10000 using only four digits, type *0000* or *0*.

register-name	Specifies the name of a microprocessor register.

string	Any number of characters enclosed in single (' ') or double (" ") quotation marks. If the quotation marks must appear within a *string*, you must use two sets of quotation marks. For example, the following strings are legal:
	`"This 'string' is okay."`
	`"This ""string"" is okay."`
	However, the following strings are illegal:
	`"This "string" is not okay."`
	`'This 'string' is not okay."`
	The ASCII values of the characters in the string are used as a list of byte values.

word	A four-digit (two-byte) hexadecimal value.

3.5.2 Binary and Unary Operators

Table 3.1 lists the binary operators and the order of precedence in which they can be used in DOS-resident debugger commands.

Table 3.2 lists the unary operators and the order of precedence in which they can be used in DOS-resident debugger commands.

Table 3.1

Binary Operators

Binary Operator	Meaning	Precedence
( )	Parentheses	Highest
:	Address binder	
*	Multiplication	
/	Integer division	
MOD	Modulo (remainder)	
+	Addition	
–	Subtraction	
>	Greater-than relational operator	
<	Less-than relational operator	
>=	Greater-than/equal-to relational operator	
<=	Less-than/equal-to relational operator	
==	Equal-to relational operator	
!=	Not-equal-to relational operator	
AND	Bitwise Boolean AND	
XOR	Bitwise Boolean exclusive OR	
OR	Bitwise Boolean OR	
&&	Logical AND	
||	Logical OR	Lowest

Table 3.2

Unary Operators

Unary Operator	Meaning	Precedence
&(seg)	Interpret address using segment value	Highest
#(sel)	Interpret address using selector value	
%(phy)	Interpret address as a physical value	
–	Two's complement	
!	Logical NOT operator	
NOT	One's complement	
SEG	Segment address of operand	
OFF	Address offset of operand	
BY	Low-order byte from given address	
WO	Low-order word from given address	
DW	Doubleword from given address	
POI	Pointer (four bytes) from given address	
PORT	One byte from given port	
WPORT	Word from given port	Lowest

3.6 Debugger Commands

The DOS-resident debugger commands are summarized in Table 3.3 and described in detail in the remainder of this chapter. A vertical bar (¦) separating two parameters indicates that either parameter is acceptable. Parameters enclosed in brackets ([]) are optional. Ellipses (...) indicate that the preceding item can be repeated.

Table 3.3

DOS-resident Debugger Commands

Command	Function
?	Displays a help menu
? *expression*¦"*string*"	Evaluates and displays an expression or string
.?	Displays external commands help menu
.a[a][*addr*¦*range*]	Displays physical memory
.d[*data-struc-name*][*addr*]	Displays DOS data structure
.h[a][*handle*¦*range*]	Displays memory manager handle
.p	Displays current process
bc *list*¦ *	Clears breakpoints
bd *list*¦ *	Disables breakpoints
be *list*¦ *	Enables breakpoints
bl	Displays all breakpoints
bp[*n*] *addr* [*passcnt*] ["*cmd(s)*"]	Creates breakpoints
c *range addr*	Compares memory locations
d [*addr*¦*range*]	Displays memory location(s) in ASCII and hexadecimal format
db[*addr*¦*range*]	Displays memory location(s) in bytes

Table 3.3 *(continued)*

Command	Function
dd[*addr*¦*range*]	Displays memory location(s) as doublewords
dg[**a**][*range*]	Displays entries in the Global Descriptor Table (GDT)
di[**a**][*range*]	Displays entries in the Interrupt Descriptor Table (IDT)
dl[**a**][*range*]	Displays entries in the Local Descriptor Table (LDT)
dt[*addr*]	Displays Task State Segment (TSS)
dw[*addr*¦*range*]	Displays memory location(s) in words
e*addr* [*list*]	Enters bytes in memory
f*range list*	Fills memory location(s) with data from list
g[=*start-addr* [*addr...*]]	Executes the code currently in memory
h*word word*	Performs hexadecimal arithmetic
i *word*	Inputs and displays a byte from a specified 16-bit port address
j *expression* ["*cmd(s)*"]	Executes optional commands if expression is TRUE
lg	Lists the selector and the name of each group in the active map
lm	Lists symbol files currently loaded and displays which one is active
ln[*addr*]	Lists the symbol nearest to the specified address

Table 3.3 *(continued)*

Command	Function
ls*group-name*	Lists the symbols in the group
m*range addr*	Moves block of memory to different memory location
o *word byte*	Writes a byte to a specified 16-bit port address
p[=*start-addr*][*count*]	Executes specified instruction and displays registers and flags
r[*register-name*][=*word*]	Displays the contents of one or more CPU registers
s*range list*¦"*string*"	Searches for specified list of bytes or ASCII character string
t[=*addr*] [*word*]	Executes an instruction and displays registers and flags
u[*range*]	Disassembles and displays code
w*map-name*	Changes the active map file
vc[r¦p]*byte,...*	Clears trapping of the specified internal interrupt and exception vector
vl[r¦p]	Displays internal hardware interrupt and exception vectors being trapped
vs[r¦p]*byte,...*	Passes control (traps) to debugger when specified internal interrupt or exception is executed
x	Reboots the target system
z	Executes the default command string
zl	Displays the default command string
zs "*string*"	Changes default command to command(s) specified in *string*

3.6.1 ? – Display Help Menu

The **?** command displays a list of commands and syntax recognized by the DOS-resident debugger. The syntax for **?** is

?

Typing *?* at the command prompt results in the following display:

```
BC [<list> | *] - clear breakpoint(s)
BD [<list> | *] - disable breakpoint(s)
BE [<list> | *] - enable breakpoint(s)
BL - list breakpoint(s)
BP[<n>] <addr> [<passcnt>] ["<bp cmd(s)>"] - set a breakpoint
C <range> <addr> - compare bytes
D [<addr> <range>] - display memory
DB [<addr> <range>] - display memory in bytes
DW [<addr> <range>] - display memory in words
DD [<addr> <range>] - display memory in dwords
DG [A] [<range>] - display GDT entries
DI [A] [<range>] - display IDT entries
DL [A] [<range>] - display LDT entries
DT [<addr>] - display TSS
E <addr> [<list>] - enter memory
F <range> <list> - fill memory
G [=<addr> [<addr> ...]] - go (begin execution)
H <word> <word> - hex arithmetic
I <word> - input from port
J <expr> [<cmds>] - execute <cmds> if <expr> is true (non-zero)
LG - list groups in active map
LM - list linked maps
LN [<addr>] - list near symbols
LS <group name> - list symbols in named group
M <range> <addr> - move memory
O <word> <byte> - output to port
P [=<addr>] [<word>] - program step
R [<reg>] [[=] <word>] - register
S <range> <list> - search for byte
T [=<addr>] [<word>] - trace execution
U [<range>] - unassemble code
VL [R | P] - list interrupt vectors set
VS [R | P] <byte> ... - set an interrupt vector
VC [R | P] <byte> ... - clear an interrupt vector
W - changes active map file
X - reboot target machine
Z - execute the default command
ZL - list the default command
ZS - change the default command
. - external commands, execute ".?" for help
? - display help menu ? <expr> | "string" - display expression or string
<range> = [<addr>] [<word>] | [<addr>] [L <word>]
<addr> = [& | #] [<word>:]<word> | %<dword>
<list> = <byte> <byte> ... | "string"
<binary ops> = : * / MOD + - < > >= <= AND XOR OR && ||
<unary ops> = & (seg) # (sel) % (phy) ! NOT SEG OFF BY WO DW POI PORT WPORT
```

3.6.2 ? *expression* – Display Expression

The ? *expression* command displays the value of a specified expression or string. The syntax for ? *expression* is

? *expression*| "*string*"

where *expression* is any combination of numbers, addresses, and operators and *string* is a sequence of characters enclosed in single or double quotation marks.

An expression is first evaluated and then displayed in hexadecimal, decimal, octal, and binary format. The debugger also displays an ASCII character representation of the evaluated expression, a physical address interpretation, and whether the expression evaluated to a Boolean TRUE or FALSE.

Strings enclosed in quotation marks are echoed to the screen.

The expression evaluator provides support for three types of addresses: real mode (%selector:offset), protected mode (#selector:offset), and physical address (%dword).

The &, #, and % characters override the current address type, allowing selectors to be used in real mode, segments to be used in protected mode, and so on. The % character converts other addresses to physical addresses. For example:

```
%(#001F:0220)
```

This example will look up selector 1F's physical address in the current LDT and add 0220 to it.

The following keywords can be used with the **?** *expression* command:

Keyword	**Description**
reg	Returns the value of *reg*, where *reg* is one of the following registers: AX, BX, CX, DX, SI, DI, BP, DS, ES, SS, CS, SP, or IP
FLG	Returns the value of the flags
GDTB	Returns the value of the GDT base as a physical address
GDTL	Returns the value of the GDT limit
IDTB	Returns the value of the IDT base as a physical address
IDTL	Returns the value of the IDT limit
TR	Returns the value of the TR register
LDTR	Returns the value of the LDTR register
MSW	Returns the value of the MSW register

The @ character can be used with any of the register names to ensure that the expression evaluator interprets the name as a register instead of a symbol (for example, **@ax** is the same as **ax**)

To display the value of the expression *DS:SI + BX*, type

```
? ds:si+bx
```

The debugger returns a display similar to the following:

```
2038:4278 540557944T 401604117OOQ 0100001001111000Y 'X' %0245F8 TRUE
```

To display the value of an arithmetic expresion such as *3*4*, type

```
? 3*4
```

The debugger returns the following display:

```
OCH 12T 14Q 00001100Y '.' %00000C TRUE
```

The **?** *string* command is useful with breakpoint commands. You can use it to announce a breakpoint number as in the following example:

```
bp1 100 "r;d 200;? 'BP 1 REACHED'"
```

3.6.3 .? – Display External Commands

The **.?** command displays a list of external commands. These commands are part of the debugger, but are specific to the environment in which the debugger is running. The syntax for **.?** is

.?

Typing *.?* at the command prompt results in the following display:

```
.A[A] [<addr>] - display physical arena or an entry
.D <name> [<addr>] - display a DOS data struc: SFT, DPB, BPB, VPB, DEV, REQ
.H[A] [<handle>] | [<range>] - display handle table or entry
.P - prints process status
? prints help message
```

3.6.4 .a – Display Physical Memory

The **.a** command displays the physical memory arena at a specified physical address. The syntax for **.a** is

.a [a] [*addr*|*range*]

where the second **a** is an option that displays the physical arena information, but suppresses the owner information; *addr* specifies an arena address; *range* specifies a range of arena addresses.

If no options are specified, the entire arena is displayed.

The default address type is physical, so a % symbol is not always necessary. If other address types are used, you must specify the appropriate override selector (& or #).

The command displays information such as the size of the arena, the owner, and what process (MTE or PTDA) owns the segment. It also displays the PID of the process and, if it is available, the name of the module that owns the segment. Code segments normally only have the module name and no process ID. If the segment is an MTE, PTDA, or LDT, the command displays the object name, the process ID (PTDA), and the module name if possible.

The following list describes the owners that the **.a** command can display:

Owner	Description
FREE	Free memory
DevHlp	Allocated by the **DevHlpAllocPhys** call
MSHARE	Shared memory
GIVESEG	Give segmentable memory
BDEVICE	Base device driver memory
IDEVICE	Installable device driver memory
3XBOX	Memory owned by the 3.*x* box
SYSTEM	System owned memory
DOSEXE	Initial memory arena where MS OS/2 started (includes all the MS OS/2 code and data segments)
MTE	Module table entry
PTDA	Per task data arena
LDT	Local descriptor table
ROM	ROM BIOS area in the arena

If the display just has a PID or module name, then this piece of memory is owned by that process or module.

The **.a** command is useful when you need to find out who owns a selector because of a GP (*General Protection*) fault in some unknown LDT segment. For example:

```
.a %(cs:0) -20H
```

This command displays the owner of the code segments. (The "cs" can be replaced with any selector.) It converts the selector into a physical address and subtracts the size of the arena header so that the resulting address points to the segment's physical arena header. The display is similar to the following:

```
Address  Sig  Flag  Hand  Owner  Size      Owner     Module
%27A600  47   0300  0610  FADE   00005380  dosexe
```

3.6.5 .d – Display Data Structures

The **.d** command displays DOS data structures. The syntax for **.d** is

.d *data-struc-name* [*addr*]

where *data-struc-name* is the name of the data structure you want to display and *addr* is an optional address where the structure is located.

The following are data structures the **.d** command can display:

Name	Description
SFT	System file table entry
VPB	Volume parameter block
DPB	Disk parameter block
DEV	Device driver header
REQ	Device driver request packet

For example, to display the current device driver header, type

```
.d dev
```

In response, the DOS-resident debugger produces a display similar to the following:

```
  DevNext:  06B9:0233
  DevAttr:  0001
 DevStrat:  5988
   DevInt:  0828
 NumUnits:  00
DevProtCS:  20A8
DevProtDS:  8088
DevRealCS:  FFFF
DevRealDS:  FFFF
```

3.6.6 .h – Display Memory Manager Handle

The **.h** command displays the virtual memory manager's handle. The syntax for **.h** is

.h[**a**] [*handle*| *range*]

where **a** is an option that displays all of the handles rather than just the ones currently being used. If no *range* is specified, the entire table is displayed.

For example, the following displays the handle at 05A8:

.H 05A8

In response, the debugger produces a display similar to the following:

```
Handle      Addr          LockCnt     Sel       Flags
05A8        %11C560       00          0290      1011
```

3.6.7 .p – Display Current Process

The **.p** command displays the priority, status, block ID, and name of the process or thread. The syntax for **.p** is

.p

For example, typing *.p* produces a display similar to the following:

```
Slot  ThdNum  State  Priority  BlockID   Pid  Ppid  PTDA    Name
2     1       void   0         00000000  0     0    2764A0
```

3.6.8 bc – Clear Breakpoints

The **bc** command removes one or more defined breakpoints. The syntax for **bc** is

bc *list* ¦ *

where *list* is any combination of integer values in the range 0–9. If you specify *list*, the DOS-resident debugger removes the specified breakpoints. If you specify *, all breakpoints are cleared.

For example, to remove breakpoints 0, 4, and 8, type

bc 0 4 8

To remove all breakpoints, type

bc *

3.6.9 bd – Disable Breakpoints

The **bd** command *temporarily* disables one or more breakpoints. The syntax for **bd** is

bd *list* ¦ *

where *list* is any combination of integer values in the range 0–9. If you specify *list*, the DOS-resident debugger disables the specified breakpoints. If you specify *, all breakpoints are disabled.

For example, to disable breakpoints 0, 4, and 8, type

```
bd 0 4 8
```

To disable all breakpoints, type

```
bd *
```

To restore breakpoints disabled by the **bd** command, use the **be** (**enable breakpoints**) command.

3.6.10 be – Enable Breakpoints

The **be** command restores (enables) one or more breakpoints that have been temporarily disabled by a **bd** (**disable breakpoints**) command. The syntax for **be** is

be *list* | *

where *list* is any combination of integer values in the range 0–9. If you specify *list*, the DOS-resident debugger enables the specified breakpoints. If you specify *, all breakpoints are enabled.

For example, to enable breakpoints 0, 4, and 8, type

```
be 0 4 8
```

To enable all breakpoints, type

```
be *
```

3.6.11 bl – List Breakpoints

The **bl** command lists current information about all breakpoints created by the **bp** (**set breakpoints**) command. The syntax for **bl** is

bl

If no breakpoints are currently defined, the DOS-resident debugger displays nothing. Otherwise, the breakpoint number, enabled status, breakpoint address, number of passes remaining, initial number of passes (in parentheses), and any optional debugger commands to be executed when the breakpoint is reached are displayed on the screen, as in the following example:

```
0 e 04BA:0100
4 d 04BA:0503 4 (10)
8 e 0D2D:0001 3 (3) "R;DB DS:SI"
```

In this example, breakpoints 0 and 8 are enabled (e), while 4 is disabled (d). Breakpoint 4 had an initial pass count of 10 and has 4 remaining passes to be taken before the breakpoint. Breakpoint 8 had an initial pass count of 3 and must make all 3 passes before it halts execution and forces the debugger to execute the optional debugger commands enclosed in quotation marks. Breakpoint 0 shows no initial pass count, which means it was set to 1.

3.6.12 bp – Set Breakpoints

The **bp** command creates a software breakpoint at an address. The syntax for **bp** is

bp[*n*] *addr* [*passcnt*] ["*cmds*"]

where *n* is a number specifying which breakpoint is being created, *addr* is any valid instruction address (the first byte of an instruction opcode), *passcnt* specifies the number of times the breakpoint is to be ignored before being executed, and "*cmds*" is an optional list of debugger commands to be executed in place of the default command when the breakpoint is reached.

During program execution, software breakpoints stop program execution and force the DOS-resident debugger to execute the default or optional command string. Unlike breakpoints created by the **g** (**go**) command, software breakpoints remain in memory until you remove them with the **bc** (**clear breakpoints**) command or temporarily disable them with the **bd** (**disable breakpoints**) command.

The DOS-resident debugger allows up to ten software breakpoints (0–9). (Note that no space is allowed between the **bp** and *n*.) If you specify more than ten breakpoints, the DOS-resident debugger returns a "Too Many Breakpoints" message.

If *n* is omitted, the first available breakpoint number is used. *Passcnt* can be any 16-bit value. You must enclose optional command(s) in quotation marks, and separate optional commands with semicolons (;).

The following are examples of the **bp** command:

```
bp 123

bp8 400:23 "db DS:SI"
```

```
bp 100 10 "r;c100 L 100 300"
```

The first example creates a breakpoint at address CS:123; the second creates breakpoint 8 at address 400:23 and executes a **db** (**display bytes**) command. The third example creates a breakpoint at address 100 in the current CS selector and displays the registers before comparing a block of memory. The breakpoint is ignored 16 (10H) times before being executed.

3.6.13 c – Compare Memory

The **c** command compares one memory location against another memory location. The syntax for **c** is

c *range addr*

where *range* specifies the block of memory that is to be compared with a block of memory starting at *addr*.

If the two memory areas are identical, the DOS-resident debugger displays nothing and returns the debugger prompt. Differences, when they exist, are displayed as follows:

addr1 byte1 byte2 addr2

For example, the following two commands have the same effect—each compares the block of memory from 100H to 1FFH with the block of memory from 300H to 3FFH:

```
c100 1FF 300

c100 L 100 300
```

The first example specifies a *range* with a starting address of 100H and an ending address of 1FFH. This block of memory is compared with a block of memory of the same size starting at 300H. The second example compares the same block of memory, but specifies the *range* by using the **L** (**length**) option.

3.6.14 d – Display Memory

The **d** command displays the contents of memory at a given address or in a range of addresses. The syntax for **d** is

d [*addr* ¦ *range*]

Note that the **d** command must be separated by at least one space from any *addr* or *range* value.

The **d** command displays one or more lines, depending on the *addr* or *range* given. Each line displays the address of the first item displayed. The command always displays at least one value. If you specify *range*, the command displays all values in the range. If you omit *addr* and *range*, the **d** command displays the next byte of memory after the last one displayed. The memory display is in the format defined by a previously executed **db**, **dd**, or **dw** command.

Each subsequent **d** (typed without parameters) displays the bytes immediately following those last displayed.

For example, if you type the following command, the display is formatted as described above, but 20H bytes are displayed instead:

```
d CS:100 L 20
```

Suppose you then type

```
d CS:100 115
```

As a result, the display is formatted as previously described, but all the bytes in the range 100H to 115H in the CS selector are displayed.

3.6.15 db – Display Bytes

The **db** command displays the values of the bytes at a given address or in a given range. The syntax for **db** is

db [*addr* ¦ *range*]

If you specify a *range* of addresses, the contents of the *range* are displayed. If you omit *addr* and *range*, 128 bytes are displayed beginning at the first address after the address displayed by the previous **db** command.

The display is in two portions: a hexadecimal display (each byte is shown in hexadecimal value) and an ASCII display (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary.

For example, suppose you type the following:

```
db CS:100 0A
```

In response, the DOS-resident debugger displays the lines in the following format:

```
04BA:0100 54 4F 4D 20 53 ... 45 52 TOM SAWYER
```

Each line of the display begins with an address, incremented by 10H from the address on the previous line.

3.6.16 dd – Display Doublewords

The **dd** command displays the hexadecimal values of the doublewords at the address specified or in the specified range of addresses. The syntax for **dd** is

dd [*addr* ¦ *range*]

The **dd** command displays one or more lines, depending on the *range* given. Each line displays the address of the first doubleword in the line, followed by up to four hexadecimal doubleword values. The hexadecimal values are separated by spaces. The **dd** command displays values until the end of the *range* or until the first 32 doublewords have been displayed.

For example, to display the doubleword values from CS:100 to CS:110, type

```
dd CS:100 110
```

The resulting display is similar to the following:

```
04BA:0100 7473:2041 676E:6972 5405:0104 0A0D:7865
04BA:0110 0000:002E
```

No more than four values per line are displayed.

Typing **dd** displays 32 doublewords at the current dump address. For example, if the last byte in the previous **dd** command was 04BA:0110, the display starts at 04BA:0111.

3.6.17 dg – Display GDT

The **dg** command displays the specified range of entries in the GDT (*Global Descriptor Table*). For an explanation of the GDT, see Appendix B, "References." The syntax for **dg** is

dg[**a**] [*range*]

If you omit *range*, the DOS-resident debugger displays the entire contents of the GDT. The **a** option causes all entries in the table to be displayed, not just the valid entries. The default is to display just the valid GDT entries. If the command is passed an LDT selector, it displays *LDT* and the appropriate LDT entry.

For example, to display only the valid entries from 0H to 40H in the GDT, type

```
dg 0 40
```

The resulting display is similar to the following:

```
0008  Data Seg  Base=01D700 Limit=3677 DPL=0 Present ReadWrite Accessed
0010  TSS Desc  Base=007688 Limit=002B DPL=0 Present Busy
0018  Data Seg  Base=020D7A Limit=03FF DPL=0 Present ReadWrite
0020  Data Seg  Base=000000 Limit=03FF DPL=0 Present ReadWrite
0028  LDT Desc  Base=000000 Limit=0000 DPL=0 Present
0030  Data Seg  Base=000000 Limit=0000 DPL=0 Present ReadWrite
0040  Data Seg  Base=000400 Limit=03BF DPL=3 Present ReadWrite
```

3.6.18 di – Display IDT

The **di** command displays the specified range of entries in the IDT (*Interrupt Descriptor Table*). For an explantion of the IDT, see Appendix B, "References." The syntax for **di** is

di[**a**] [*range*]

If you omit *range*, the DOS-resident debugger displays all IDT entries. The **a** option causes all entries in the table to be displayed, not just the valid ones. The default is to display just the valid IDT entries.

For example, suppose you type the following:

```
di 0 10
```

This command produces a display of valid entries similar to the following:

```
0000  Int Gate  Sel=1418    Offst=03D8 DPL=3 Present
0001  Int Gate  Sel=2D38    Offst=0049 DPL=3 Present
0002  Int Gate  Sel=1418    Offst=03E4 DPL=3 Present
0003  Int Gate  Sel=2D38    Offst=006F DPL=3 Present
0004  Int Gate  Sel=1418    Offst=0417 DPL=3 Present
0005  Int Gate  Sel=1418    Offst=041D DPL=3 Present
0006  Int Gate  Sel=1418    Offst=0423 DPL=3 Present
0007  Int Gate  Sel=2D38    Offst=00A3 DPL=3 Present
0008  Int Gate  Sel=1418    Offst=042F DPL=3 Present
0009  Int Gate  Sel=2D38    Offst=00CA DPL=3 Present
000A  Int Gate  Sel=2D38    Offst=00D3 DPL=3 Present
000B  Int Gate  Sel=2D38    Offst=0156 DPL=3 Present
000C  Int Gate  Sel=2D38    Offst=01A4 DPL=3 Present
000D  Int Gate  Sel=2D38    Offst=01C6 DPL=3 Present
```

3.6.19 dl – Display LDT

The **dl** command displays the specified range of entries in the LDT (*Local Descriptor Table*). For an explanation of the LDT, see Appendix B, "References." The syntax for **dl** is

dl[a] [*range*]

If you omit *range*, the entire table is displayed. The **a** option causes all entries in the table to be displayed, not just the valid ones. The default is to display just the valid LDT entries.

For example, to display all of the LDT entries, type

```
dla 4 57
```

This command produces a display similar to the following:

```
0014  Call Gate Sel=1418    Offst=0417 DPL=0 NotPres WordCount=1D
001C  Code Seg  Base=051418 Limit=0423 DPL=0 NotPres ExecOnly
0027  Reserved  Base=87F000 Limit=FEA5 DPL=3 Present
0034  Code Seg  Base=05F000 Limit=1805 DPL=0 NotPres ExecOnly
003C  Code Seg  Base=05F000 Limit=EF57 DPL=0 NotPres ExecOnly
0047  Code Seg  Base=4DC000 Limit=0050 DPL=3 Present ExecOnly
004D  Reserved  Base=71F000 Limit=F841 DPL=1 NotPres
0057  Code Seg  Base=59F000 Limit=E739 DPL=3 Present ExecOnly
```

3.6.20 dt – Display TSS

The **dt** command displays the current TSS (*Task State Segment*) or the selected TSS if you specify the optional address. For an explanation of the TSS, see Appendix B, "References." The syntax for **dt** is

dt [*addr*]

For example, to display the current TSS, type

```
dt
```

The resulting display is similar to the following:

```
AX=0000   BX=0000   CX=0000   DX=0000   SP=0000   BP=0000   SI=0000   DI=0000
IP=0000   CS=0000   DS=0000   ES=0000   SS=0000   NV UP DI PL NZ NA PO NC
SS0=0038  SP0=08DE  SS1=0000  SP1=0000  SS2=0000  SP2=0000
IOPL=0    LDTR=0028 LINK=0000
```

3.6.21 dw – Display Words

The **dw** command displays the hexadecimal values of the words at a given address or in a given range of addresses. The syntax for **dw** is

dw [*addr* | *range*]

The command displays one or more lines, depending on the *range* given. Each line displays the address of the first word in the line, followed by up to eight hexadecimal word values. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed.

For example, suppose you type

```
dw CS:100 110
```

This command displays the word values from CS:100 to CS:110, resulting in a display similar to the following:

```
04BA:0100 2041 7473 6972 676E 0104 5404 7865 0A0D
04BA:0110 002E
```

No more than eight values per line are displayed.

Typing **dw** displays 64 words at the current dump address. If the last word in the previous **dw** command was displayed at address 04BA:0110, the next display will start at 04BA:0112.

3.6.22 e – Enter Byte

The **e** command enters byte values into memory at a specified address. The syntax for **e** is

e *addr* [*list*]

If you specify an optional *list* of values, the byte values are replaced automatically with the values in *list*. (If an error occurs when you are using the list form of the command, no byte values are changed.)

If you omit *list*, the DOS-resident debugger displays the address and its contents and then waits for you to perform one of the following actions:

- Replace a byte value with a value you type. Type the value after the current value. If the byte you type is an illegal hexadecimal value or contains more than two digits, the illegal or extra character is not echoed.

- Press SPACEBAR to advance to the next byte. To change the value, type the new value after the current value. If, when you press SPACEBAR, you move beyond an 8-byte boundary, the DOS-resident debugger starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with its address and byte value displayed.
- Press RETURN to terminate the **e** command. You can press RETURN at any byte position.

For example, suppose you type

```
eCS:100
```

Suppose the DOS-resident debugger then displays the following:

```
04BA:0100 EB._
```

To change this value to, say, 41, type the number 41 at the cursor, as shown:

```
04BA:0100  EB.41_
```

To step through the subsequent bytes, press SPACEBAR. For example, pressing SPACEBAR three times might result in the following display:

```
04BA:0100 EB.41  10.  00.  BC._
```

Now, to change BC to the number 42, type *42* at the cursor, as follows:

```
04BA:0100  EB.41    10.    00.    BC.42_
```

Suppose you want to change the value 10 to 6F. Type the hyphen as many times as needed to return to byte 0101 (value 10) and then replace 10 with 6F:

```
04BA:0100  EB.41    10.    00.    BC.42-
04BA:0102  00.-_
04BA:0101  10.6F_
```

Press RETURN to terminate the **e** command.

3.6.23 f – FillMemory

The **f** command fills the addresses in a specified range with the values in the specified list. The syntax for **f** is

f *range list*

If *range* contains more bytes than the number of values in *list*, the DOS-resident debugger uses *list* repeatedly until all bytes in *range* are filled.

If *list* contains more values than the number of bytes in *range*, the DOS-resident debugger ignores the extra values in *list*.

If any of the memory in *range* is not valid (bad or nonexistent), an error will occur in all succeeding locations.

For example, suppose you type the following:

```
f04BA:100 L 100 42 45 52 54 41
```

In response, the DOS-resident debugger fills memory locations 04BA:100 through 04BA:1FF with the bytes specified, repeating the five values until it has filled all 100H bytes.

3.6.24 g – Go

The **g** command executes the program currently in memory. The syntax for **g** is

g [=*start-addr* [*break-addr*...]]

where *start-addr* specifies the address where execution is to begin and *break-addr* specifies one or more breakpoint addresses where execution is to halt.

If you type the **g** command by itself, the current program executes as if it had run outside the DOS-resident debugger. If you specify =*start-addr*, execution begins at the specified address. The equal sign (=) is needed to distinguish the starting address from the breakpoint address.

Specifying an optional breakpoint address causes execution to halt at the first address encountered, regardless of the position of the address in the list of addresses that halts execution or program branching. When program execution reaches a breakpoint, the default command string is executed.

As noted in Section 3.6.12, "bp – Set Breakpoints," you can specify up to 10 breakpoints, but only at addresses containing the first byte of an opcode.

The stack (SS:SP) must be valid and have six bytes available for this command. The **g** command uses an **IRET** instruction to cause a jump to the program under test. The stack is set, and the user flags, CS register, and IP are pushed on the user stack. (If the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es).

When the DOS-resident debugger encounters an instruction with the breakpoint code, it restores all breakpoint addresses listed with the **g** command to their original instructions. If you do not halt execution at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

For example, suppose you type the following:

```
gCS:7550
```

In response, the program currently in memory executes until address 7550 in the CS selector is executed. The DOS-resident debugger then executes the default command string, removes the INT 3 trap from this address, and restores the original instruction. When you resume execution, the original instruction will be executed.

3.6.25 h – Hexadecimal Arithmetic

The **h** command performs hexadecimal arithmetic on the two specified parameters. The syntax for **h** is

h *word word*

The DOS-resident debugger adds, subtracts, multiplies, and divides the second parameter and the first parameter, then displays the results on one line. The debugger does 32-bit multiplication and displays the result as doublewords. The debugger displays the result of division as a 16-bit quotient and a 16-bit remainder.

For example, suppose you type

```
h 100 100
```

In response, the DOS-resident debugger performs the calculations and displays the following:

```
+0200  -0000 *0000 0001 /0001 0000
```

3.6.26 i – Input Byte

The **i** command inputs and displays one byte from a specified port. The syntax for **i** is

i *word*

where *word* specifies the 16-bit port address.

For example, to display the byte at port address 2F8H, type

```
i2F8
```

3.6.27 j – Conditional Execute

The **j** command executes the specified command(s) when the specified expression is TRUE. The syntax for **j** is

j *expression* ["*cmds*"]

where *expression* must evaluate to a Boolean TRUE or FALSE, and "*cmds*" is a list of DOS-resident debugger commands to be executed when *expression* is TRUE.

If *expression* is FALSE, the DOS-resident debugger continues to the next command line (excluding the commands in *cmds*).

Cmds must be enclosed in single or double quotation marks. You must separate optional commands with semicolons (;). You can use a single or NULL command without quotation marks.

The **j** command is useful in breakpoint commands to conditionally break execution when an expression becomes true.

For example, suppose you type

```
bp 167:1454 "J AX == 0;G"
```

This command causes execution to break if AX does not equal zero when the breakpoint is reached.

The following command displays the registers and continues execution when the byte pointed to by *DS:SI +3* is equal to 40H; otherwise, it displays the descriptor table:

```
bp 167:1462 "J BY (DS:SI+3) == 40 'R;G';DG DS"
```

The following command breaks execution when the breakpoint is reached in real mode:

```
bp 156:1455 "J (MSW AND 1) == 1 'G'"
```

The following command is a shortcut that produces the same results as the preceding command:

```
bp 156:1455 "J (MSW AND 1) 'G'"
```

3.6.28 lg – List Groups

The **lg** command lists the selector (or segment) and the name of each group in the active map. The syntax for **lg** is

lg

For example, typing *lg* produces a display similar to the following:

```
#0090:0000 DOSCODE
#0828:0000 DOSGROUP
#1290:0000 DBGCODE
#16C0:0000 DBGDATA
#1A38:0000 TASKCODE
#1AD8:0000 DOSRING3CODE
#1AE0:0000 DOSINITCODE
#2018:0000 DOSINITRMCODE
#20A8:0000 DOSINITDATA
#23F8:0000 DOSMTE
#2420:0000 DOSHIGHDATA
#28D0:0000 DOSHIGHCODE
#3628:0000 DOSHIGH2CODE
#0090:0000 DOSCODE
```

3.6.29 lm – List Map

The **lm** command lists the symbol files currently loaded and which one is active. The syntax for **lm** is

lm

For example, if you type *lm*, the debugger returns a display similar to the following:

```
COMSAM2D is active.
DISK01D.
```

The last symbol file loaded is made active by default. Use the **w** (**change**

map file) command to change the active file.

3.6.30 ln – List Near

The **ln** command lists the symbol nearest to the specified address. The syntax for **ln** is

ln [*addr*]

where *addr* is any valid instruction address. The default is the current disassembly address.

ln lists the nearest symbol before and after the specified *addr*. For example:

```
6787 VerifyRamSemAddr + 10 ¦ 67AA PutRamSemID - 13
```

3.6.31 ls – List Symbols

The **ls** command lists the symbols in the specified group. The syntax for **ls** is

ls *group-name*

where *group-name* is the name of the group that contains the symbols you want to list. For example, to display all of the symbols in the DOSRING3CODE group, type

```
ls DOSRING3CODE
```

In response, the debugger displays the symbols in a format similar to the following:

```
0000 Sig_dispatch
001A Lib_Init_Disp
```

3.6.32 m – Move Memory

The **m** command moves a block of memory from one memory location to another. The syntax for **m** is

m *range addr*

where *range* specifies the block of memory to be moved, and *addr* specifies the starting address where the memory is to be relocated.

Overlapping moves—those where part of the block overlaps some of the current addresses—are always performed without loss of data. Addresses that could be overwritten are moved first. For moves from higher to lower addresses, the sequence of events is to first move the data at the block's lowest address and then work toward the highest. For moves from lower to higher addresses, the sequence is to first move the data at the block's highest address and then work toward the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data in the block *before the move* will remain. The **m** command copies the data from one area into another, in the sequence described, and writes over the new addresses—hence, the importance of the moving sequence.

For example, suppose you type the following:

```
mCS:100 110 CS:500
```

In response, the DOS-resident debugger first moves address CS:110 to CS:510 and then moves CS:10F to CS:50F, and so on, until CS:100 is moved to CS:500.

To review the results of a memory move, use the **d** (**display memory**) command, specifying the same address you used with the **m** command.

3.6.33 o - Output to Port

The **o** command writes a byte to a 16-bit port address. The syntax for **o** is

o *word byte*

where *word* specifies the 16-bit port address you are writing to. For example, if you want the DOS-resident debugger to output the byte value 4F to output port 2F8, type

```
o 2F8 4F
```

3.6.34 p – Program Trace

The **p** command executes the instruction at a specified address and then executes the default command string. The syntax for **p** is

p [=*start-addr*][*count*]

where *start-addr* specifies the starting address to begin execution, and *count* specifies the number of instructions to execute before halting and executing the default command string.

If you omit the optional *start-addr*, execution begins at the instruction pointed to by the CS and IP registers. Use the equal sign (=) only if you specify a *start-addr*.

If you specify *count*, the command continues to execute *count* instructions before stopping. The command executes the default command string for each instruction before executing the next.

The **p** command is identical to the **t** (**trace instructions**) command, except that it automatically executes and returns from any calls or software interrupts it encounters. The **t** command always stops after executing into the call or interrupt, leaving execution control inside the called routine.

For example, typing *p* causes the DOS-resident debugger to execute the instruction pointed to by the current CS and IP register values before it executes the default command string. Typing *p=120* causes the DOS-resident debugger to execute the instruction at address CS:120 before it executes the default command string.

3.6.35 r – Display Registers

The **r** command displays the contents of one or more CPU registers. The syntax for **r** is

r [*register-name*] [=*word*]

If you omit *register-name*, the DOS-resident debugger displays the contents of all registers and flags along with the next executable instruction.

For example, typing *r* produces the following display:

```
AX=0698  BX=2008  CX=2C18  DX=18AB  SP=1B7A  BP=00FF  SI=0020  DI=10CD
IP=0450  CS=18B0  DS=1BE8  ES=0DA8  SS=0048  NV UP DI PL NZ NA PO NC
GDTR=01BE80 3687  IDTR=01F508 03FF  TR=0010  LDTR=0028 IOPL=3 MSW=PM
18B0:0450 C3            RET
```

If you specify a *register-name* with the **r** command, the 16-bit value of that register is displayed in hexadecimal followed by a colon (:) prompt on the next line. You can then enter a new *word* value for the specified register or press RETURN if you do not want to change the register value.

F is a special name for the Flags register. If you type *f* as the *register-name*, the DOS-resident debugger displays each flag with a two-character alphabetic code. To change any flag, type the opposite two-letter code. The flags are either set or cleared.

Table 3.4 lists the 80286 flags and the codes for setting or clearing them.

Table 3.4
80286 Flags

Flag Name	Set	Clear
Overflow	OV	NV
Direction	DN (decrement)	UP (increment)
Interrupt	EI (enabled)	DI (disabled)
Sign	NG (negative)	PL (plus)
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE (even)	PO (odd)
Carry	CY	NC
Nested Task	NT	NT (toggles on/off)

When you use the **rf** command, the DOS-resident debugger displays the flags in a row at the beginning of a new line and displays a hyphen (-) after the last flag.

You can type new flag values in any order as alphabetic pairs. You do not have to leave spaces between these values. To exit the **r** command, press RETURN. Any flags for which you did not specify new values remain unchanged.

If you type more than one value for a flag or if you enter a flag code other than one of those shown in the Table 3.4, the DOS-resident debugger returns a "Bad Flag" error message. In both cases, the flags up to the error in the list are changed; those flags at and after the error are not changed.

For example, if you type *rf*, the DOS-resident debugger produces a display similar to the following:

```
NV UP DI NG NZ AC PE NC - _
```

Now, to change the value of a flag's setting, enter an opposite setting for the flag you wish to set. For example, typing the following command and pressing RETURN returns the DOS-resident debugger's command prompt (# or >), changes the sign flag to positive, enables interrupts, and sets the carry flag:

```
NV UP DI NG NZ AC PE NC - PLEICY
```

To see the changes made, use either the **r** or the **rf** command.

Type *rmsw* to modify the MSW (*Machine Status Word*) bits. The debugger displays the status of the MSW register and prints a colon on the next line. You can then enter one or more of the following mnemonics to change the MSW register bits:

Mnemonic	Meaning
TS	Sets the task switch bit
EM	Sets the emulation processor extension bit
MP	Sets the monitor processor extension bit
PM	Sets the protected mode bit

Note

Setting the protected-mode bit from within the debugger does *not* set the target system to run in protected mode. The debugger simulates the setting. To configure the target system to run in protected mode, you would have to set the PM bit in the MSW register and reset the target system to boot up in protected mode.

3.6.36 s – Search Bytes

The **s** command searches an address range for a specified list of bytes or an ASCII character string. The syntax for **s** is

s *range list*|"*string*"

You can include one or more bytes in *list*, but multiple bytes must be separated by a space or comma. When you search for more than one byte, the command returns only the first address of the byte string. When your *list* contains only one byte, the DOS-resident debugger displays all addresses of the byte in the *range*. If you specify a *string*, it must be enclosed in quotation marks.

For example, suppose you type the following:

```
sCS:100 110 41
```

This command produces a display similar to the following:

```
04BA:0104
04BA:010D
```

3.6.37 t – Trace Instructions

The **t** command executes one or more instructions along with the default command string, then displays of the decoded instruction. The syntax for **t** is

t [=*addr*] [*word*]

If you include the =*addr* option, tracing occurs at the specified address. The *word* option causes the DOS-resident debugger to execute and trace the number of instructions specified by *word.* When specifying an address, use the equal sign so that the debugger interprets it as an address rather than a count variable.

The **t** command uses the hardware trace mode of the Intel microprocessor. Consequently, you can also trace instructions stored in ROM (*Read Only Memory*).

If you type *t*, the DOS-resident debugger steps through the next machine instruction and then executes the default command string.

Assuming, for this example, that the current position is 04BA:011A and that the default command string is the **r** command, the DOS-resident debugger display is similar to the following:

```
AX=0E00  BX=00FF  CX=0007  DX=01FF  SP=039D  BP=0000  SI=005C  DI=0000
IP=011A  CS=04BA  DS=04BA  ES=04BA  SS=04BA  NV UP DI NG NZ AC PE NC
GDTR=01D700 3677  IDTR=020D7A 03FF  TR=0010  LDTR=0028 IOPL=3 MSW=PM
04BA:011A  CD21           PUSH   21
```

Now, suppose you type

```
t=011A 10
```

This command causes the DOS-resident debugger to execute 16 (10H) instructions beginning at 011A in the current selector. The debugger executes and displays the results of the default command string for each instruction. The display scrolls until the last instruction is executed. Use CONTROL-S to stop the display from scrolling, CONTROL-Q to resume.

3.6.38 u – Unassemble Bytes

The **u** command disassembles bytes and displays the source statements, with addresses and byte values, that correspond to them. The syntax for **u** is

u [*range*]

The display of disassembled code looks similar to a listing for an assembled file. If you type the **u** command by itself, 20H bytes are disassembled at the first address after the one displayed by the previous **u** command. If you include the *range* parameter, the DOS-resident debugger disassembles all bytes in *range*—unless you specify *range* only as an address, in which case 20H bytes are disassembled.

For example, suppose you type

```
uCS:046C
```

In response, the DOS-resident debugger returns a display similar to the following:

```
1A60:046C C3              RET
1A60:046D 9A6B3E100D      CALL    0D10:3E6B
1A60:0472 33C0            XOR     AX,AX
1A60:0474 50              PUSH    AX
1A60:0475 9D              POPF
1A60:0476 9C              PUSHF
1A60:0477 58              POP     AX
1A60:0478 2500F0          AND     AX,F000
1A60:047B 3D00F0          CMP     AX,F000
1A60:047E 7508            JNZ     0488
1A60:0480 689C26          PUSH    269C
1A60:0483 9AF105100D      CALL    0D10:05F1
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. You can also use the **u** command for the changed locations, for the new instructions viewed, and for the disassembled code used to edit the source file.

3.6.39 vc – Clear Interrupt Vector

The **vc** command clears the trapping of specified internal interrupt and exception vectors. The syntax for **vc** is

vc [**r**¦**p**] *byte,...*

where **r** and **p** specify the mode (real or protected), and *byte* specifies the interrupt vector to be cleared.

Typing the **vc** command without specifying a mode clears the specified interrupt vector in both real and protected modes. This allows the program under test to field the interrupt without debugger intervention.

For example, suppose you typed the **vl** (**list interrupt vector**) command and the DOS-resident debugger displayed the following:

```
R 0 1 2 3 4 5 6 7
P 0 2 4 5 6 7 8
```

To clear vector 8 in protected mode, type

```
vcp 8
```

To clear vector 7 in both real and protected modes, type

```
vc7
```

A subsequent **vl** command displays

```
R 0 1 2 3 4 5 6
P 0 2 4 5 6
```

3.6.40 vl – List Interrupt Vector

The **vl** command displays internal hardware interrupt and exception vectors that are in use. The sytax for **vl** is

vl[**r**¦**p**]

If you include either the **r** or the **p** option, the DOS-resident debugger displays the interrupt vectors in use in the selected mode. If you omit the **r** and **p** options, the DOS-resident debugger displays the interrupt vector for both real and protected modes.

For example, typing *rl* produces a display similar to the following:

```
R 0 1 2 3 4 5 6
P 0 2 4 5 6 8
```

Typing *vlr* displays real-mode interrupt vectors:

```
R 0 1 2 3 4 5 6
```

Typing *vlp* displays protected-mode interrupt vectors:

```
P 0 2 4 5 6 8
```

3.6.41 vs – Set Interrupt Vector

The **vs** command enables the DOS-resident debugger to regain control when the specified internal interrupt or exception occurs. The syntax for **vs** is

vs [**r**¦**p**] *byte,...*

where the **r** and **p** options limit changes to vectors in real mode and protected mode, respectively; *byte* specifies the interrupt vector you want to set (0 – 6).

For example, suppose you used the **vl** (**list interrupt vector**) command to display the following:

```
R 0 1 2 3 4 5 6
P 0 2 4 5 6 8
```

Now, if you want the debugger to gain control when vector 7 is invoked in real mode, type

```
vsr7
```

A subsequent **vl** command produces the following display:

```
R 0 1 2 3 4 5 6 7
P 0 2 4 5 6 8
```

3.6.42 w – Change Map

The **w** command changes the active map file. The syntax for **w** is

w *map-name*

where *map-name* is the name of the map file you want to make active. (Use the **lm** (**list map**) command to display a list of available map files.)

For example, suppose you used the **lm** command to display the loaded map files, and the debugger displayed the following:

```
COMSAM2D is active.
DISK01D.
```

To make *disk01d* the active symbol file, type

```
w DISK01D
```

3.6.43 x – Reboot Target System

The **x** command reboots the target system. The syntax for **x** is

x

3.6.44 z – Execute Default Command String

The **z** command executes the default command string. The syntax for **z** is

z

The default command string is initially set to the **r** (**display registers**) command by the DOS-resident debugger. The default command string is executed every time a breakpoint is encountered during program execution or whenever a **p** (**program trace**) or **t** (**trace instructions**) command is executed.

Use the **zl** command to display the default command string and the **zs** command to change the default command string.

3.6.45 zl – Display Default Command String

The **zl** command displays the default command string. The syntax for **zl** is

zl

For example, typing *zl* produces a display similar to the following:

```
"R"
```

3.6.46 zs – Change Default Command String

The **zs** command lets you change the default command string. The syntax for **zs** is

zs "*string*"

where *string* specifies the new default command string. *String* must be enclosed in single or double quotation marks. You must separate the DOS-resident debugger commands within the string with semicolons (;).

For example, to change the current default command string to a register (**r**) command followed by a comparison of memory blocks (**c**), type

```
zs "r;c100 L 100 300"
```

If you want to begin execution whenever an INT 3 is executed in your test program, type

```
zs "j (by cs:ip) == cc 'g'"
```

This will execute a **g** (**go**) command every time an INT 3 is executed.

You can use **zs** to set up a "watchpoint" as follows:

```
zs "j (wo 40:1234) == 0eeed;t"
```

This command traces until the word at 40:1234 is *not* equal to 0EEED.

3.7 Debugging with DBPRT Statements

Warning

The code that supports the DBPRT (*Debug Print*) routines is in the *main.asm* file in the sample implementation. The DBPRT routines are not a feature of the DOS-resident debugger. If you want to use the DBPRT routines in your BIOS, you must provide the code to support it.

The DBPRT routines enable you to use formatted print statements to help debug your code. DBPRT statements can be placed inside any assembly-language file linked into *os2bio.com.* They are useful for displaying execution points, register values, and so on.

DBPRT statements are conditionally assembled only in the debug version of the BIOS, *os2biod.com.*

The syntax for a DBPRT statement is

dbprt *n, m,* **<***string***>, <***a1,...an***>**

where *n, m* are unused bytes that can be any value, *string* is an ASCII format string enclosed in greater-than, less-than brackets (< >), and *a1,...an* is an optional list of arguments. Multiple arguments must be separated by a comma.

The *string* can contain two types of format specifications: literal characters and data-format specifications.

A literal character can be any character that is not part of the format specification. Special non-printing characters are as follows:

Literal	Displayed Result
\n	carriage return/line feed
\t	tab
\b	bell
\\	\
\$	$

Data-format specifications begin with the $ character; all other characters are treated as part of the literal string. A data-format specification takes the form

$ [@][*char*]

where @ displays a memory location pointed to by a register's contents. It takes a double-word pointer like ES:BX.

char is a 16-bit word. Legal values for *char* are as follows:

Value	Result
x	Causes the argument to be displayed as a hexadecimal word.
d	Causes the argument to be displayed as a decimal word.
c	Causes the argument to be displayed as an ASCII character.
b	Causes the argument to be displayed as a hexadecimal byte.
s[*nn*]	Causes the argument to be displayed as an ASCII string. *nn* is a decimal number that specifies the string length. Non-printing characters are printed in the form \nnn, where *nnn* is the octal byte value. Note that **s** cannot be followed by a digit other than the one that specifies the length of a string to be displayed. You cannot use the @ symbol when using **s**.
B*nn*	Causes the argument to be displayed as hexadecimal bytes. *nn* is a decimal number that specifies the number of bytes to print. You cannot use the @ symbol with **B**.

3.7.1 Notes on Using DBPRT Statements

Keep the following points in mind when you use DBPRT statements:

- DBPRT statements cannot be used in **DosHlp** functions.
- A string or byte length of 0 is taken as 65,536.
- Entering an illegal format specification causes an error. The following message appears:

  ```
  %%BAD FMT%%
  ```
- Using the **B** format with a length of 0 or omitting the length causes an error. The following message appears:

  ```
  %%BAD FMT %%
  ```

3.7.2 Examples of Using DBPRT Statements

The following DBPRT statement uses formatted character specifications to print various register values when the *bootdd_init* module is initialized:

```
dbprt 1,1, <BootDD_Init: BX=$x CX=$x DX=$x \n>, <bx,cx,dx>
```

When executed, the DBPRT statement displays a formatted string similar to the following:

```
BootDD_Init: BX=29B0 CX=3668 DX=1D0B
```

The next example displays the value of the DS:SI register and 19 bytes of memory pointed to by DS:SI:

```
dbprt 1,1,<BootDD_Init: Copy BPB $x:$x ($B19) \n>, <ds,si, ds,si>
```

The following example shows the use of the **s** and @ options:

```
teststring   db   'This is a test string',0

push    si
mov     si, offset dgroup:teststring
dbprt   1,1,<The test string is: $s21 \n>, <ds,si>
dbprt   1,1,<The first word is: $@x 0, <ds,si>
pop     si
```

In response, the following strings are displayed when these statements are executed:

```
The test string is: This is a test string
The first word is 6854
```

3.8 Debugging with Map Files

This section describes how you can use the information found in map files with INT 3 instructions to debug code in which symbolic support is not available. The map file is generated at link time and contains a listing of all public subroutines and labels in your code. It also lists the offset of these routines and labels.

Note

Symbolic support is provided for device drivers. The serial device driver discussed in this section is used for illustrative purposes only since you could symbolically debug this driver. The examples discussed can be applied to code in which symbolic debug support is not available.

For many of the DOS-resident debugger commands, you need a fully-qualified address (selector:offset). The offset can be obtained from the map file. However, the selector is not determined until run time.

One of the ways to obtain the selector is to place INT 3 instructions inside the code you are debugging. INT 3 instructions halt execution and cause the DOS-resident debugger to display the register values. The selector for the DATA segment can be found in the DS register and the selector for the CODE segment can be found in the CS register. The selector values can then be used with the offset found in the map file with any debugging commands that require fully-qualified addresses.

3.8.1 Reading a Map File

The following is part of the map file for the serial device driver discussed in Chapter 5, "Sample Device Drivers:"

```
Start        Length     Name          Class
0001:0000    004FEH     MAIN_DATA     DATA
0002:0000    00447H     MAIN_CODE     CODE

Address             Publics by Name

0001:0054           COM1
0001:0000           COM1DEV
0002:0020           COMEXIT0
0002:0025           COMEXIT1
0002:0025           COMEXIT2
.
.
.
0001:004C           DEVHLP
0001:001A           DISPTAB
0002:0108           GET_VIRT_ADDR

Address             Publics by Value

0001:0000           COM1DEV
0001:001A           DISPTAB
0001:004C           DEVHLP
.
.
.
0002:03CA           SAVE_PID
0002:0411           INIT_PARMS
```

The Start column lists the segment index associated with a CODE or DATA segment. A device driver's DATA segment is always the first segment, so these have the number 0001. CODE segments follow and have the number 0002. The address column lists the segment index and offset of the associated public symbol.

Looking at the map file, the public COM1 is in the DATA segment and has an offset of 0054H. To obtain the run-time selector address for this module, you could place an INT 3 instruction inside the routine. This would halt execution and display the register values.

3.8.2 Example Debug Session

Suppose you are executing the sample serial device driver described in Chapter 5, "Sample Device Drivers," and that you are having problems with the driver and want to use the DOS-resident debugger to debug it.

To obtain the run-time selector of the DATA and CODE segments, place an INT 3 instruction as the first instruction in the CODE segment. Begin execution by typing **g** (**go**) from within the DOS-resident debugger. The first instruction the device driver executes is INT 3. This halts execution and passes control to the DOS-resident debugger, which displays the register values (CODE in CS, DATA in DS).

Assume you get a CODE selector of 2568 and a DATA selector of 1D50 from the display.

Now, suppose you want to set a breakpoint that halts execution at the offset of the GET_VIRT_ADDR label. Define the breakpoint so that the debugger does a conditional execute (**j**) based on the contents of memory at the offset of the DISPTAB label in the DATA segment.

To create the breakpoint, type

```
bp 2568:0108 "J by(1D50:001A) ==2E 'T=0108 10'"
```

In the example, the debugger will trace the next 16H instructions if the memory location is equal to 2E. Otherwise, it halts execution.

Chapter 4
Automated Tests

4.1 Introduction 65
4.2 Setting Up the Tests 65
4.2.1 Running the ADD Tests 66
4.2.2 Changing the Flag Byte 67
4.2.3 Notes on ADD Testing 67
4.2.4 Overview of the ADD Tests 68
4.2.5 Summary of ADD Tests 70
4.2.5.1 Test Device Driver Headers 70
4.2.5.2 Test Device Driver Functions 70
4.2.5.3 Test IRQInfoTable 72
4.2.5.4 Test IDT 75

4.1 Introduction

This chapter describes how to debug and test the MS OS/2 device drivers. An overview of the testing strategy and a discussion of each test is provided.

A special version of the DOS-resident debugger contains the Automated Device Driver (ADD) tests. The ADD tests can be used to test your device drivers.

To debug and test your target system, you will use the following software tools:

- The DOS-resident debugger, provided as part of the MS OS/2 operating system in the Device Driver Kit
- The Automated Device Driver (ADD) tests, provided to automate some of the device driver testing

The following sections describe these software tools.

4.2 Setting Up the Tests

To set up the tests, perform the following steps:

1. Delete the *os2dos.com* file from your MS OS/2 bootable disk (that is, a copy of the boot disk provided with the MS OS/2 SDK).
2. Open and close the floppy disk drive door to ensure that DOS reuses the same space on the floppy disk previously occupied by *os2dos.com.*
3. Copy the *os2dost.com* file from your distribution disk to the boot disk as *os2dos.com.*

   The tests are now installed on your boot disk along with the DOS-resident debugger.
4. Make sure that your target system is connected to a terminal and that the terminal is configured to run as described in Chapter 3, "Using the DOS-resident Debugger."

4.2.1 Running the ADD Tests

The following message appears when the ADD tests are started:

```
Do you want to switch test output to the target system
for the automated device driver tests (y/n)?
```

Type *Y* to send all output from the tests to the target system's screen. Type *N* to send the output to the debug terminal.

Note

All input for the ADD tests comes from the debug terminal regardless of where the output is redirected.

The output from the tests is displayed faster if you send it to the debug terminal. However, if you plan on doing extensive debugging of your device drivers, it is recommended that you switch the screen output from the debug terminal to the target system's screen. This enables you to clearly separate the test's output from any debugging information that may be displayed.

After the tests are invoked, the ADD displays information similar to the following on your debugging terminal:

```
The user flags can be set in any combination using the
debugger.  How you set these flags determines which automated
device driver tests are run.
   01H = Do not test device driver headers
   02H = Do not test valid device driver commands
   04H = Do not test character device driver input commands
   08H = Do not test character device driver output commands
   10H = Do not test block device driver commands
   20H = Do not test invalid device driver commands
   40H = Do not test base device drivers
   80H = Do not display changes to the IDT or DOS IRQ table

To skip all the tests and boot the system, set the flags bytes
to 0FFH.  The user configuration flags byte is at xxxx:xxxx
```

By mixing the bit values listed on the screen, you can control which tests, if any, are to be executed.

4.2.2 Changing the Flag Byte

Use the **e (enter byte)** command to change the value of the byte located at address *xxxx:xxxx*. (See Chapter 3, "Using the DOS-resident Debugger," for information on the **e** command).

Suppose the flag byte is located at address 0960:3f44H and you do not want to test the device driver headers. To set the flag to skip this part of the tests, type

```
e 0960:3f44 1
```

Suppose you have already run the base device drivers and the device driver header tests for your supplemental device drivers. To skip these tests, type

```
e 0960:3f44 41
```

Now, when you run the tests, the base device drivers will not be tested, nor will the headers of your supplemental drivers.

To skip all the tests and boot the system, type

```
e 0960:3f44 ff
```

4.2.3 Notes on ADD Testing

Keep the following in mind when you run the ADD tests:

- The tests display one screen of information at a time. Press any key to continue with the tests.
- To break out of the tests, press CONTROL-C twice.
- Certain tests require user input. You are prompted to perform any needed tasks.
- Set the flag byte for no character output if your target system has a parallel printer port, but no printer attached to it.
- Skipping the character input and character output commands causes the debugger to skip the testing of the invalid character device driver commands.
- Skipping the block device driver commands causes the debugger to skip the testing of the invalid block device driver commands.
- Skipping the valid device driver commands does *not* cause the debugger to skip the testing of invalid device driver commands.
- Base device driver functionality testing cannot be done at base device driver initialization time. All base device drivers are tested at once.

- Block device driver output commands are not tested.

4.2.4 Overview of the ADD Tests

When you run the ADD tests, the base device drivers are tested first. After all the base drivers have been tested, any supplemental device drivers you have specified in the boot disk's *config.sys* file are tested.

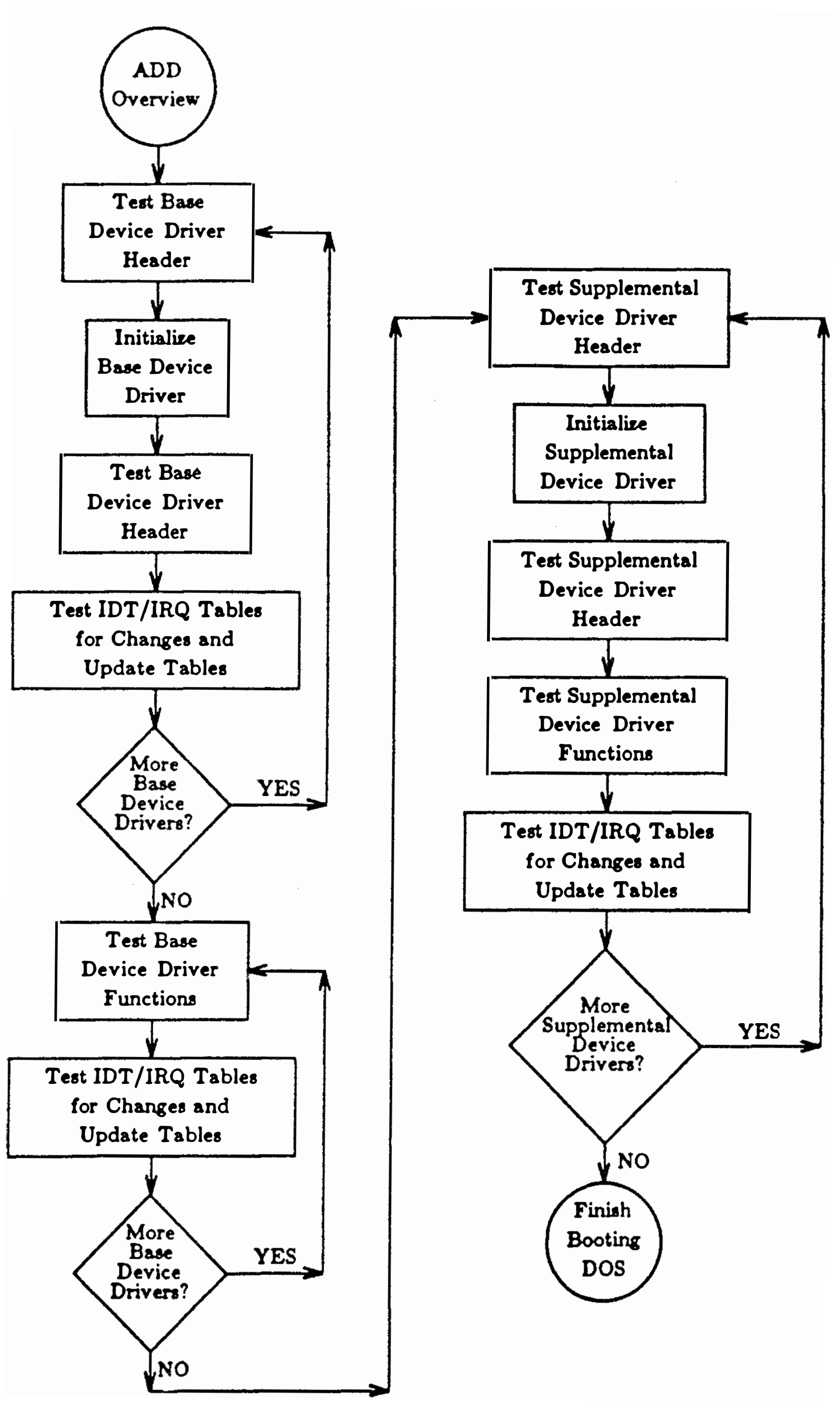
Note

Before you can test any of your supplemental device drivers, you must have a **device** =entry in the *config.sys* file on your boot disk for each of the supplemental device drivers you want to test.

If you do not want to test a supplemental device driver, remove its entry from the *config.sys* file.

For an overview of the ADD tests refer to Figure 4.1. For flowcharts illustrating each of the tests, refer to Appendix C, "Overview of Automated Tests."

Figure 4.1 Overview of Automated Device Driver Tests


ADD Overview
Test Base Device Driver Header
Initialize Base Device Driver
Test Base Device Driver Header
Test IDT/IRQ Tables for Changes and Update Tables
More Base Device Drivers?
YES
NO
Test Base Device Driver Functions
Test IDT/IRQ Tables for Changes and Update Tables
More Base Device Drivers?
YES
NO
Test Supplemental Device Driver Header
Initialize Supplemental Device Driver
Test Supplemental Device Driver Header
Test Supplemental Device Driver Functions
Test IDT/IRQ Tables for Changes and Update Tables
More Supplemental Device Drivers?
YES
NO
Finish Booting DOS

4.2.5 Summary of ADD Tests

This section briefly describes the ADD tests outlined in Figure 4.1. Note that sections of the tests can be skipped by setting the appropriate bits in the flag byte (see Section 4.4.6, "Changing the Flag Byte.")

4.2.5.1 Test Device Driver Headers

The base device driver headers are tested before and after the device driver is initialized. The following describes what part of the device driver header is tested:

Character Device Drivers

1. Test that there are no control characters in the device name.
2. Test that there is only one device attribute bit set in the flag word.
3. Test that there is only one STDIN, STDOUT, and CLOCK device.

Block Device Drivers

1. Test that there are no more than 26 block devices.
2. Test that there are no character-only bits set in the attribute word.

4.2.5.2 Test Device Driver Functions

The following tests the functionality of the device driver:

Character Device Drivers:

1. Test whether the open command agrees with the open bit in the header.
2. Test the character device input commands:
   a. Input command
   b. Nondestructive read command
   c. Input status command
   d. Input flush command

3. Test the character device driver output commands:
   a. Output command
   b. Output command with a verify command
   c. Output flush command
   d. Output status command
4. Test whether the close command agrees with the open bit in the header.
5. Test for invalid character device driver calls:
   a. Media check
   b. Build BPB
   c. Removable media check
   d. Reset media
   e. Get logical map
   f. Set logical map

Block Device Drivers

1. Test calls supported by block device drivers:
   a. Open command
   b. Block device's ability to get and set logical map
   c. Removable media command agrees with removable-media-supported bit
   d. Block device's ability to read sectors 0 and 1
   e. Build BPB command
   f. Media check command
   g. Multiple concurrent input commands
   h. Get and set logical maps for systems with one floppy drive
   i. Close commands

2. Test invalid block device driver calls:
    a. Nondestructive read
    b. Input status
    c. Input flush
    d. Output status
    e. Output flush

4.2.5.3 Test IRQInfoTable

The **IRQInfoTable** (*Interrupt Information Table*) is an internal table that MS OS/2 uses to route hardware interrupts.

When testing a device driver, the **IRQInfoTable** entries are checked for changes. Table entries change when a device driver calls the **DevHlpSetIRQ** function. If there are changes in the **IRQInfoTable** entries, the name of the device driver that caused the changes is displayed along with the old and new table entries. There are 16 entries in the **IRQInfoTable** that correspond to 16 IRQ's (0–15).

Note

Changes made to the **IRQInfoTable** do not indicate problems with the device driver. However, there are problems if the device driver calls the **DevHlpSetIRQ** function and there is no corresponding change in the **IRQInfoTable**.

An **IRQInfoTable** entry contains the following values:

IRQ Entry	(Type) - Description
IRQL_Flags	(word) - This is a flags word for the IRQ. The flags word for the IRQ is initially set up to be the value returned by the **DosHlpInitInterrupts** function. Flag values are as follows:

Flag	Value
0001H	If set, the protect-mode interrupt is switched to real mode if the 3.*x* box is the foreground screen group.
0002H	If set, the protect-mode interrupt is switched to real mode only if the 3.*x* box is the current process.
0004H	If set, the IRQ can be shared by one or more MS OS/2 device drivers. This bit reflects the shared parameter of the first **DevHlpSetIRQ** issued.
0008H	If set, the IRQ is owned by the system and the handler cannot be changed or removed.
0010H	If set, the IRQ is owned by one or more MS OS/2 device drivers (set by MS OS/2).
0020H	If set, the IRQ is owned by the 3.*x* box.
0040H	If set, the IRQ can *only* be owned by either an MS OS/2 device driver or the 3.*x* box. If clear, the IRQ can be owned by *both* an MS OS/2 device driver and the 3.*x* box.
0080–8000H	These flags bits are reserved.

IRQL_IRQNumber	(byte) - This is the number of the IRQ. For example, on the IBM PC AT, the keyboard uses IRQ 1, and the fixed-disk controller uses IRQ 14.

IRQL_SharedCount	(byte) - This is the number of interrupt handlers that share the IRQ. The number is controlled by the values in the **IRQL_Flags**.
IRQL_RealIDTOffset	(word) - This word contains the offset in real-mode IDT of the interrupt.
IRQL_3xBoxVector	(dword) - This contains the 3.*x* box interrupt handler address. It is used only if the 3.*x* box owns the IRQ. It routes interrupts in protected mode if the IRQ is owned by the 3.*x* box and the 3.*x* box is the foreground screen group.
IRQL_RealRouter	(word) - This is the real-mode router address. It is the offset to the entry point of an interrupt not owned by the 3.*x* box while in real mode. The interrupt vector table is intialized with the value of the **IRQL_RealRouter** if the IRQ is not owned by the 3.*x* box. This is where the interrupt manager intercepts hardware interrupts occurring in real mode.
IRQL_NewDD	(structure) - This entry contains up to four structures for new device driver interrupt handlers. These values can be used to find the address of your interrupt handling routine. Each structure contains a table of four entries, defined as follows:

Structure	Value
IRQL_RealVector	(dword) - Real-mode handler address.
IRQL_RealDS	(word) - Real-mode device driver data segment.
IRQL_ProtVector	(dword) - Protected-mode handler address.
IRQL_ProtDS	(word) - Protected-mode device driver data segment.

The number of **IRQL_NewDD** entries displayed depends on the value of **IRQL_SharedCount**. For example, if

IRQL_SharedCount = 1, then only one **IRQL_NewDD** entry is displayed.

4.2.5.4 Test IDT

The IDT (*Interrupt Descriptor Table*) is used by the 80286 and 80386 microprocessors to route interrupts in protected mode. The table entries must point to the MS OS/2 kernel. Device drivers are not allowed to modify the IDT. Any change to the table indicates a bug in the device driver.

Note

A change in the IDT may occur if you use the debug commands **vs** (**vector set**) or **vc** (**vector clear**) while the ADD tests are running. If you get a message indicating a change in the IDT, rerun the tests to make sure that the change was not caused by the use of the **vc** or **vs** commands.

If changes are discovered in the IDT, a message appears listing the device driver that caused the change, which IRQ entry was changed, and the old and new IDT entries.

For a description of the IDT and the flag values, refer to Appendix B, "References."

Chapter 5
Sample Device Drivers

5.1 Introduction 79
5.2 General Development Approach 79
5.3 Sample Serial Device Driver 82
5.3.1 Configuration 83
5.3.2 File Organization 84
5.3.3 Sample Code Operation 85
5.3.3.1 The Device Driver Header 85
5.3.3.2 Request Packets 86
5.3.3.3 DevHlp Routines 86
5.3.3.4 The Strategy Routine 88
5.3.3.5 The Interrupt Routine 95
5.3.4 Sample Serial Device Driver Development Approach 98
5.3.4.1 The Write Request Code 99
5.3.5 Debugging the Sample Serial Device Driver 100
5.3.5.1 Examining the Device Driver Code 101
5.4 Sample Disk Device Driver 109
5.4.1 Disk Driver Organization 109
5.4.2 File Organization 110
5.4.3 Request Packets 110
5.4.4 The Strategy Routine 111
5.4.4.1 Request Packets Processed by the Strategy Routine 114
5.4.4.2 DevHlp Routines Used by the Strategy Routine 114
5.4.5 The Interrupt Routine 116
5.4.5.1 Request Packets Processed by the

Interrupt Routine 116
5.4.5.2 DevHlp Routines Used by the Interrupt Routine 116
5.4.5.3 How the Interrupt Routine is Entered 117
5.4.5.4 The State Machine 124
5.4.6 The ROM BIOS Interlock 132
5.4.7 Generic IOCtl Functions 137
5.4.8 Development Approach 137
5.4.9 Debugging the Sample Disk Device Driver 138
5.4.9.1 A Read Request 138
5.4.9.2 Hands-on Debugging Session 140

5.1 Introduction

This chapter describes many aspects of MS OS/2 device drivers, including a general approach to their development and testing. The Device Driver Kit includes two sample device drivers that demonstrate the operational details of MS OS/2 device drivers. The device drivers provided are functional; that is, you can observe them in operation on an IBM PC AT or compatible computer using the DOS-resident debugger.

The first device driver described is a serial (character) device driver. This driver is described in two stages of completion and should provide a basic example of device driver structure and request packet handling, and some suggestions on practical ways to approach the development of a device driver. Although the sample serial device driver provides limited functionality and does not support the I/O control features recommended for serial drivers, it does provide valuable knowledge for further device driver development.

The second sample device driver described is a fully functional disk (block) device driver. This sample driver demonstrates many of the advanced features of MS OS/2 device drivers.

After reading this chapter and studying the two sample device drivers, you should be able to perform driver modifications or develop new device drivers required for your adaptation.

5.2 General Development Approach

The development of an MS OS/2 device driver can be a large, complicated task. You may want to modify the existing device driver code rather than write your own. In most cases, you can use the structure and organization of the existing device drivers. In some cases, however, you may have to develop a device driver from scratch. The remainder of this section describes a step-by-step approach for this task. You can simplify this task by dividing the work into smaller, more manageable portions.

This section presents a general approach you can use to develop and test the device driver code in small modules. You test each module individually, then combine all modules to form a functional device driver. This approach to development simplifies the development procedure and allows you to thoroughly test all parts of the device driver.

Note

The approach presented in this section is intended as a general guideline for developing MS OS/2 device drivers. Devices and their operational details will vary, so you may need to alter the procedure slightly to provide the best approach for your particular device driver. For further guidance on adapting this approach for your device, study the specific development procedures described for the sample device drivers in Section 5.3, "Sample Serial Device Driver," and Section 5.4, "Sample Disk Device Driver."

The major stages of the development process are as follows:

1. Identify the tasks that the finished device driver should perform.

   List the device functions you want the completed device driver to perform and the sections of code necessary for these tasks. This list will serve as an outline for the development of the device driver code.

2. Using the list you made in the step one, identify the minimal set of functions needed for a noninterrupting device driver.

   You will develop a noninterrupting version of the device driver first. This will allow you to postpone interrupt-driven coding difficulties, such as interrupt vector initialization, interrupt controller manipulations, and the use of semaphores, which can cause *reentrant code* problems and *race* conditions.

3. Write and test small portions of the code.

   After determining which sections of the driver are required for noninterrupting operation of the device driver, develop and test the low-level routines required for these sections. You can test many of the routines written for a device driver independently using MS-DOS version 3.*x*. Although it is not always possible, you should debug as many of the low-level routines as you can using this method.

   For specific examples of testing and debugging device driver code, see the descriptions of the sample device drivers presented in Section 5.3, "Sample Serial Device Driver," and Section 5.4, "Sample Disk Device Driver."

4. Develop a noninterrupting driver.

   Integrate the various modules developed in the previous stage and organize the logical structure of the device driver (that is, the strategy routine, interrupt routine, dispatching of interrupts, and so on). Use MS OS/2 and the DOS-resident debugger to test your noninterrupting code.

   Note

   When first using MS OS/2 to debug your device driver, you should set the **Protectonly** entry in the *config.sys* file to YES. By running in protected mode only, you can temporarily eliminate any dual mode errors, invalid memory access, and so on, that might otherwise occur.

   When organizing the code structure of the noninterrupting device driver, you may want to use the strategy routine to simulate the conditions of an interrupt-driven device driver. To do this, call the interrupt routine to dispatch and handle a simulated interrupt. This technique allows you to organize the code structure for interrupt-driven operation. The description of the sample serial device driver presented in this chapter provides an example of this technique.

5. Enable interrupt-driven operation.

   Integrate the interrupt-driven code. You must implement and test the code dealing with interrupt vectors, interrupt controllers, and semaphores in this stage of development.

   Some devices can generate interrupts from many sources. You should develop and debug the code using an interrupt from only one new source at a time, if your device permits you to do so. After one source of interrupts is working, implement and test another source of interrupts. Repeat this procedure until the device driver is fully interrupting.

   The disk device, for example, generates interrupts at the completion of **read**, **write**, and **seek** requests. To develop and test the code for a **read complete** first, you must also debug interrupts from the **seek complete**. To simplify the procedure, develop and test the **seek complete** first, then implement the **read complete** code. After this code is tested, you can then implement and test the **write complete** code.

6. Add other device driver functions.

   The last stage in the development process involves the addition of any advanced features, such as IOCtl functions, monitor support functions, and MS-DOS 3.*x* compatibility features.

This type of development approach separates the tasks involved and allows you to verify your work before moving to the next task. You should perform tests in each of these stages so that you can isolate any problems as they are introduced.

The following sections describe the sample serial and disk device drivers and the development stages for each driver.

5.3 Sample Serial Device Driver

The sample serial device driver provided in the MS OS/2 Device Driver Kit acts as an interface between MS OS/2 and the COM1 serial I/O device in your system to provide interrupt-driven serial communications.

The MS OS/2 Device Driver Kit includes two versions of a sample serial device driver with limited functionality. These two versions represent the device driver in two stages of development. The following paragraphs describe both versions of the sample serial device driver:

- Sample serial device driver—noninterrupting version

  The first version of the sample serial device driver illustrates the first stage of development. The device driver can perform only noninterrupting output, no matter how the overall form and structure of the interrupting device driver is organized. Since this version of the driver provides minimum functionality, it is for demonstration purposes only. You may want to use it as a reference when studying the second version of the sample device driver or when creating your own device drivers.

- Sample serial device driver—interrupting version

  The second version of the driver illustrates a more advanced stage of development. In this stage, all interrupts are enabled, so the device driver can perform interrupt-driven output and input.

For more information on the MS OS/2 serial device driver, see the *Microsoft Operating System/2 Device Drivers Guide.*

After reading this section and studying both versions of the sample driver provided, you should understand the basics of MS OS/2 device driver structure and some approaches to developing your own MS OS/2 device drivers. In addition, this section should familiarize you with the following concepts:

- Device driver headers
- Device driver structure
- **DevHlp** service routines
- Request packet handling

5.3.1 Configuration

The MS OS/2 sample device driver supports output to the COM1 serial device, using the 8250 asynchronous communications chip. (For more information on the 8250, refer to the *IBM PC XT Technical Reference.*) The MS OS/2 sample serial device driver supports the following communication parameters:

Parameter	Value
Baud rate	9600
Data bits	8
Stop bits	1
Parity	none
Handshaking	none

To configure the baud rate, parity, and number of data bits and stop bits, modify the following variables:

Variable	File
BAUD_DFLT	*comdefs.inc* (modifies baud rate)
LC_DFLT	*com.inc* (modifies data bits, stop bits, and parity)

You cannot modify the handshaking parameter in the sample serial device driver.

5.3.2 File Organization

The serial device driver code is located in the following files:

File	Description
comsam1.asm	Contains the noninterrupting version of the sample serial device driver
comsam2.asm	Contains the final, interrupt-driven version of the sample serial device driver

The DOS **make** program updates and links device driver modules. When you run **make**, the following files are created:

File	Description
comsam1.sys	Contains the working version of the noninterrupting sample serial device driver
comsam1d.sys	Contains the debug version of the noninterrupting sample serial device driver
comsam2.sys	Contains the working version of the interrupt-driven sample serial device driver
comsam2d.sys	Contains the debug version of the interrupt-driven sample serial device driver

You will use the debug versions of the serial device driver later in this section for a sample debugging session. In this debugging session, you can view and modify the device driver code.

5.3.3 Sample Code Operation

This section provides an overview of the general operation of the sample serial device driver. The following components of the serial device driver are described:

- Device driver headers
- Request packets
- **DevHlp** routines
- Strategy routine
- Interrupt routine

5.3.3.1 The Device Driver Header

The data segment in the sample serial device driver program contains a device header that describes the COM1 device. The *device attribute* field in this header describes specific characteristics of the device driver. Certain bits in the device attribute field, when turned on (set to 1), represent these characteristics. The following is an illustration of the device attribute field for the sample serial device driver:

```
 15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| C |///| I |///| O |///|           | G |///|///| C | N | S | K |
| H |///| B |///| P |///| L E V E L | I |///|///| L | U | C | B |
| R |///| M |///| N |///|           | O |///|///| K | L | R | D |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |       |       |           |       |           |   |   |   |
  1       0       1          001      0           0   0   0   0
```

The bits that are turned on (set to 1) specify the following:

Bit(s)	Meaning
15	This device driver controls a character device.
11	The device driver supports open/close calls.
9,8,7	The device driver is supported at the MS OS/2 level.

In the device attribute field for the fully functional MS OS/2 serial device driver, bit 6 is also turned on. This bit represents the presence of Generic IOCtl functions, which the sample serial device driver does not use.

5.3.3.2 Request Packets

Following are the device driver commands found in the request packets that the sample serial device driver processes:

Code	Function
0	Init
4	Read
5	Nondestructive read, no wait
6	Input status
7	Input flush
8	Write
9	Write with verify
10	Output status
11	Output flush
13	Device open
14	Device close
20	Deinstall

If the sample serial device driver receives a request packet for any of the other device driver commands listed in the *Microsoft Operating System/2 Device Drivers Guide* (the commands that are not functional in the sample serial device driver), it sets the done and error bits and returns.

For individual descriptions of the device driver commands, see the *Microsoft Operating System/2 Device Drivers Guide.*

5.3.3.3 DevHlp Routines

The **DevHlp** (device helper) routines provide an interface to operating system services. (For a description of the **DevHlp** routines, see the *Microsoft Operating System/2 Device Drivers Guide.*) In the sample serial device driver, the strategy and interrupt routines use **DevHlp** routines for the following functions:

- Manipulating character queues
- Converting memory addresses
- Accessing and requesting interrupt services

Manipulating Character Queues

The sample serial device drivers use character queues to control the input and output of data to and from the serial port. In this driver, the queues are accessed by the strategy routine on a read request, or by the interrupt routine on a write request.

MS OS/2 provides **DevHlp** routines to manipulate the character queues. Device drivers are not required to use these queue manipulation routines; however, using these routines can simplify the development of a character device driver.

The sample serial device driver uses the following **DevHlp** routines to manipulate the character queues:

Function	Description
QueueInit	Initializes the character queue structure so the serial device driver can read and write characters to and from the serial port.
QueueFlush	Clears the input and output queues.
QueueWrite	Inserts a character into the output queue on a read request when the strategy routine is running, inserts a character from the serial port into the input queue at interrupt time, or sets an indicator if the queue is full.
QueueRead	Removes a character from the output queue for output to the serial port on a write command, removes a character from the input queue on a read command, or sets an indicator if the queue is empty.

Converting Memory Addresses

MS OS/2 passes addresses to the driver in physical form. These physical addresses are not directly usable by an MS OS/2 device driver. The sample serial device driver uses the **DevHlp** memory address conversion routine to change a physical address to an address that is meaningful to the driver.

The sample serial device driver uses the following memory conversion routines:

Routine	Description

PhysToVirt	Converts the 32-bit address (source or destination of the character to be sent or received) to a segment:offset address if the serial device driver is running in real mode, or a selector:offset address if the serial device driver is running in protected mode.
UnPhysToVirt	Converts a segment:offset address (if running in real mode) or a selector:offset address (if running in protected mode) to a 32-bit address.

Accessing and Requesting Interrupts

MS OS/2 provides interrupt services for device drivers. Since direct manipulation of interrupt vectors by the driver is not allowed, MS OS/2 provides **DevHlp** routines to allow the device drivers to acccess interrupt services.

The sample serial device driver uses the following interrupt access/request routines to inform MS OS/2 of an interrupt request:

Routine	Description
SetIRQ	Sets up the interrupt handlers for the serial device driver.
UnsetIRQ	Instructs MS OS/2 to remove the current interrupt handlers reserved for the serial device driver.

5.3.3.4 The Strategy Routine

When MS OS/2 issues an I/O request for the serial device driver, the strategy routine is called with a request packet for the requested function. This occurs during *task-time* execution.

In general, a process that performs serial communications will perform the following operations:

- Open the device (exclusive access)
- Perform read/write to the device
- Close the device (release it to other processes)

Figure 5.1 illustrates the flow of operation of the strategy routine in the sample serial device driver. The open, read, write, and close operations are illustrated in figures 5.2–5.5.

Open Request

When the strategy routine calls an open request, it performs the following operations, as illustrated in Figure 5.2:

1. It checks to see if any other processes have access to the serial device.
2. It sets up the queue structure and hardware if the device is unowned; or fails, marks the failure, and instructs the strategy routine to terminate if the device is owned.

Read Request

When the strategy routine calls the read request, it performs the following operations, as illustrated in Figure 5.3:

1. It checks to see that this was the process that opened the device.
2. It transfers the character(s) from the input queue to the destination address.
3. It blocks the requesting process, if the queue is empty, and waits for characters to become available.

Write Request

When the strategy routine calls the write request, it performs the following operations, as illustrated in Figure 5.4:

1. It checks to see that this was the process that opened the device.
2. It transfers characters from the source address to the output queue for transfer by the interrupt routine.
3. It blocks the requesting process, if the output queue is full, and waits until the request can complete.

Close Request

When the strategy routine calls the close request, it performs the following operations, as illustrated in Figure 5.5:

1. It checks to see that this was the process that opened the device.
2. It flushes the output queue.
3. Then it releases the hardware so other processes can access the device driver.

Figure 5.1 Strategy Routine Operation

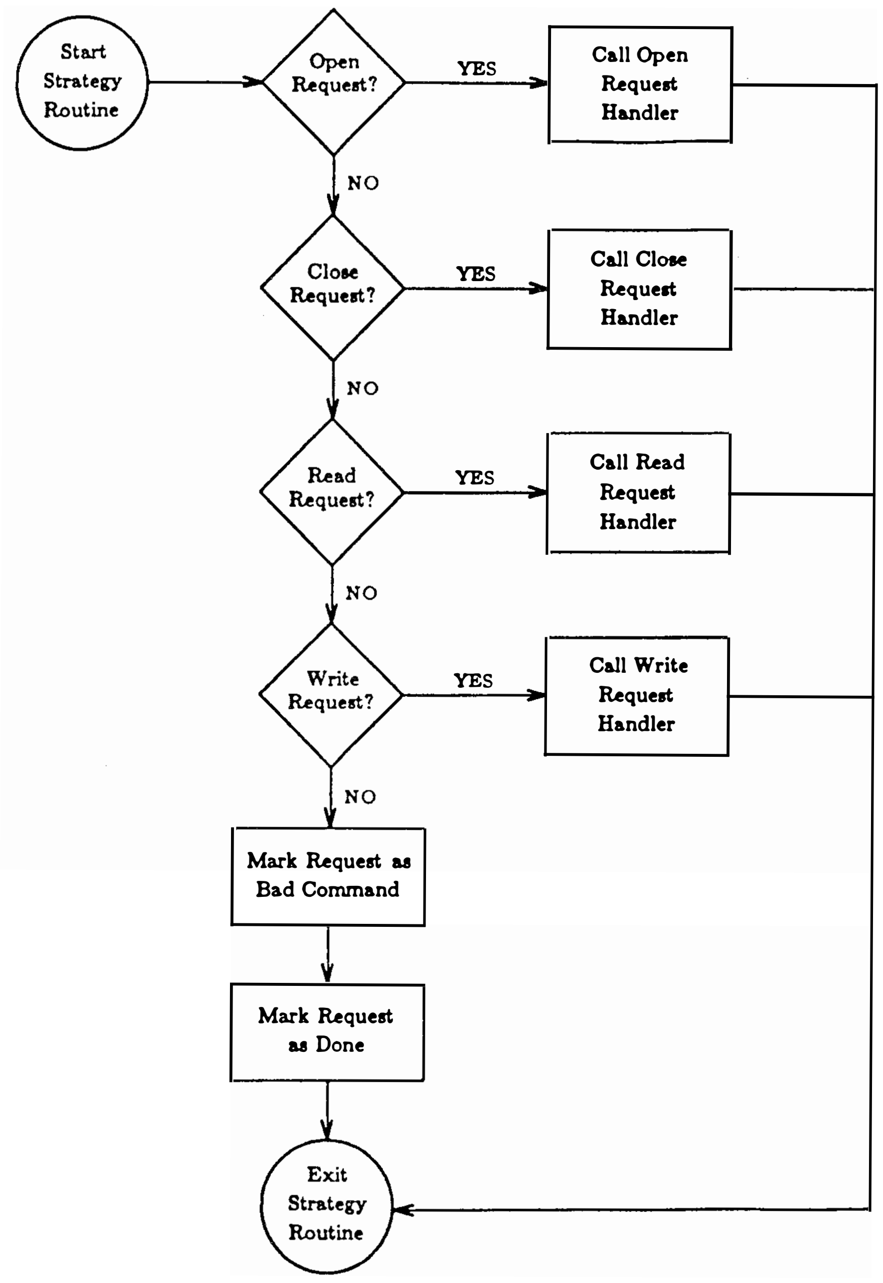


Figure 5.2 Open Request (Strategy Time)

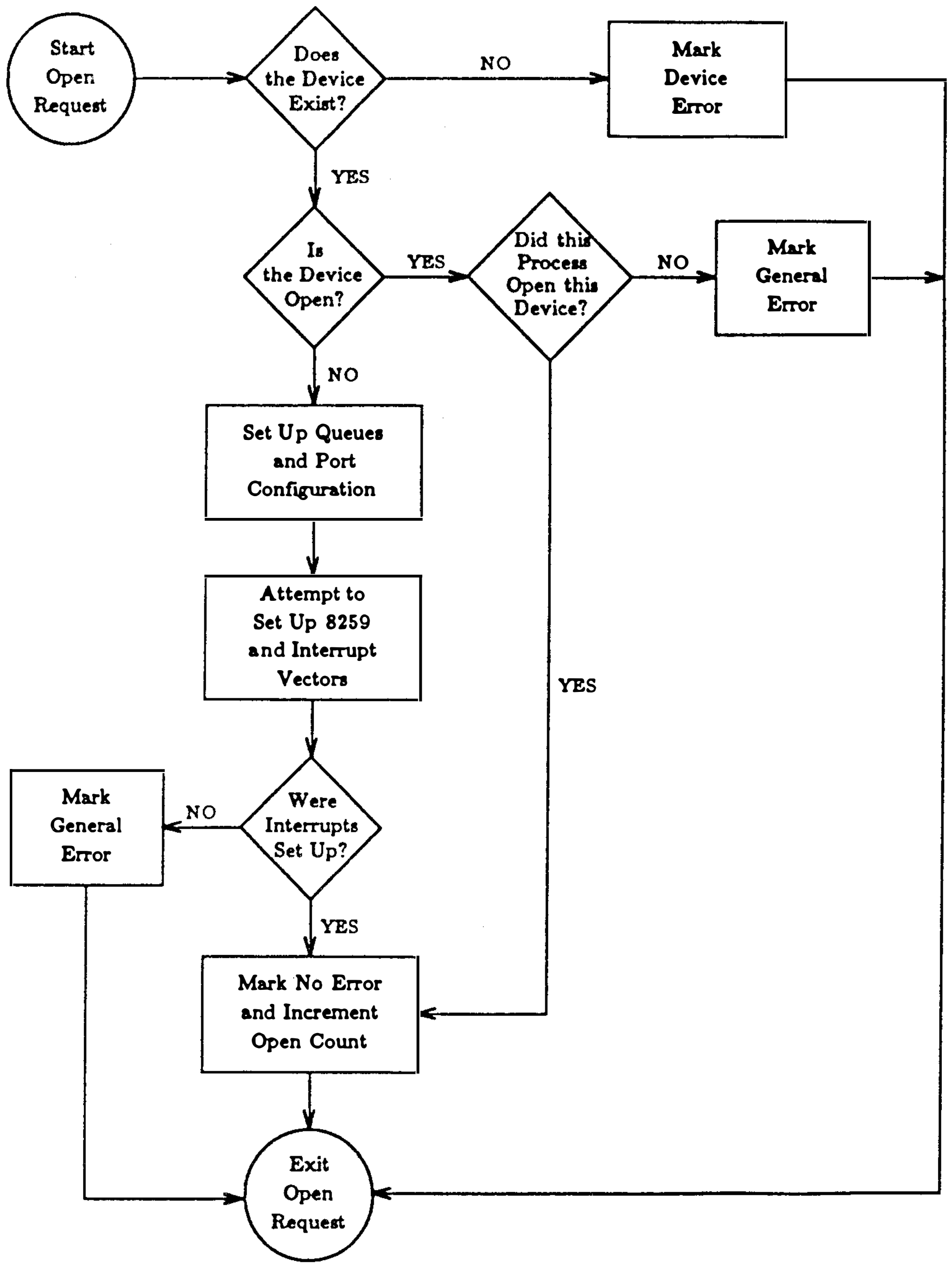


Figure 5.3 Read Request (Strategy Time)

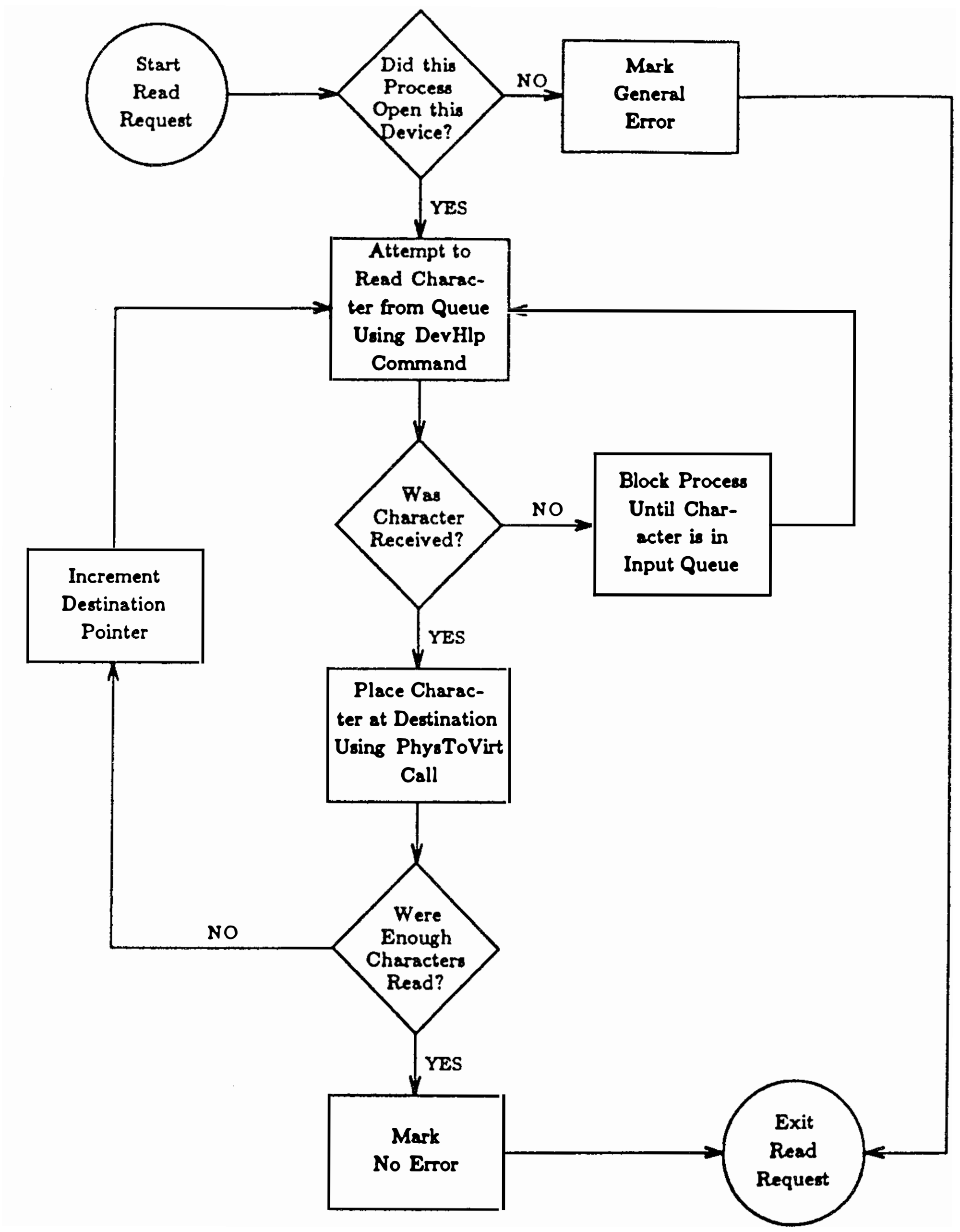

Start Read Request
Did this Process Open this Device?
NO
Mark General Error
YES
Attempt to Read Character from Queue Using DevHlp Command
Was Character Received?
NO
Block Process Until Character is in Input Queue
YES
Place Character at Destination Using PhysToVirt Call
Increment Destination Pointer
Were Enough Characters Read?
NO
YES
Mark No Error
Exit Read Request

Figure 5.4 Write Request (Strategy Time)

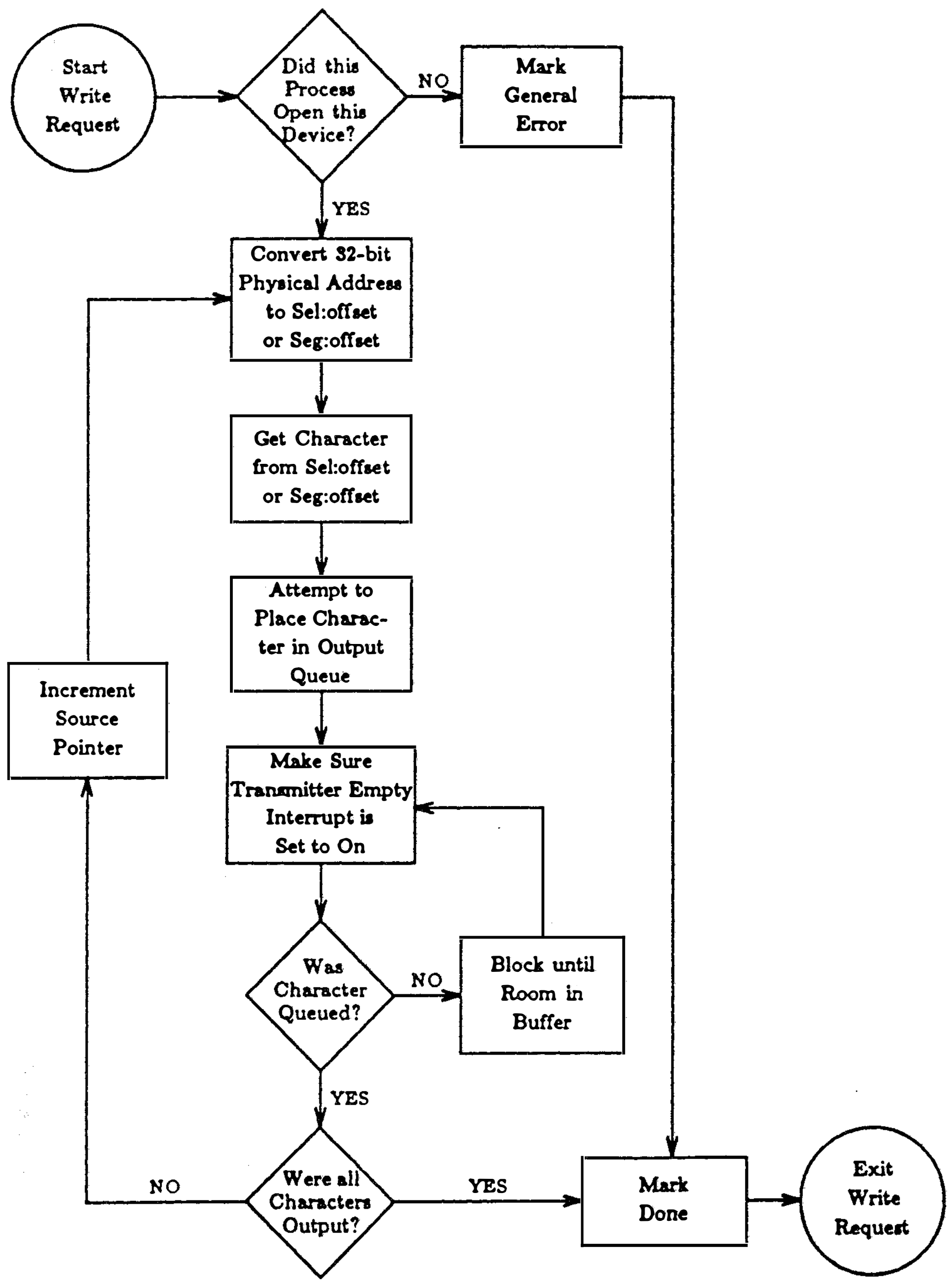
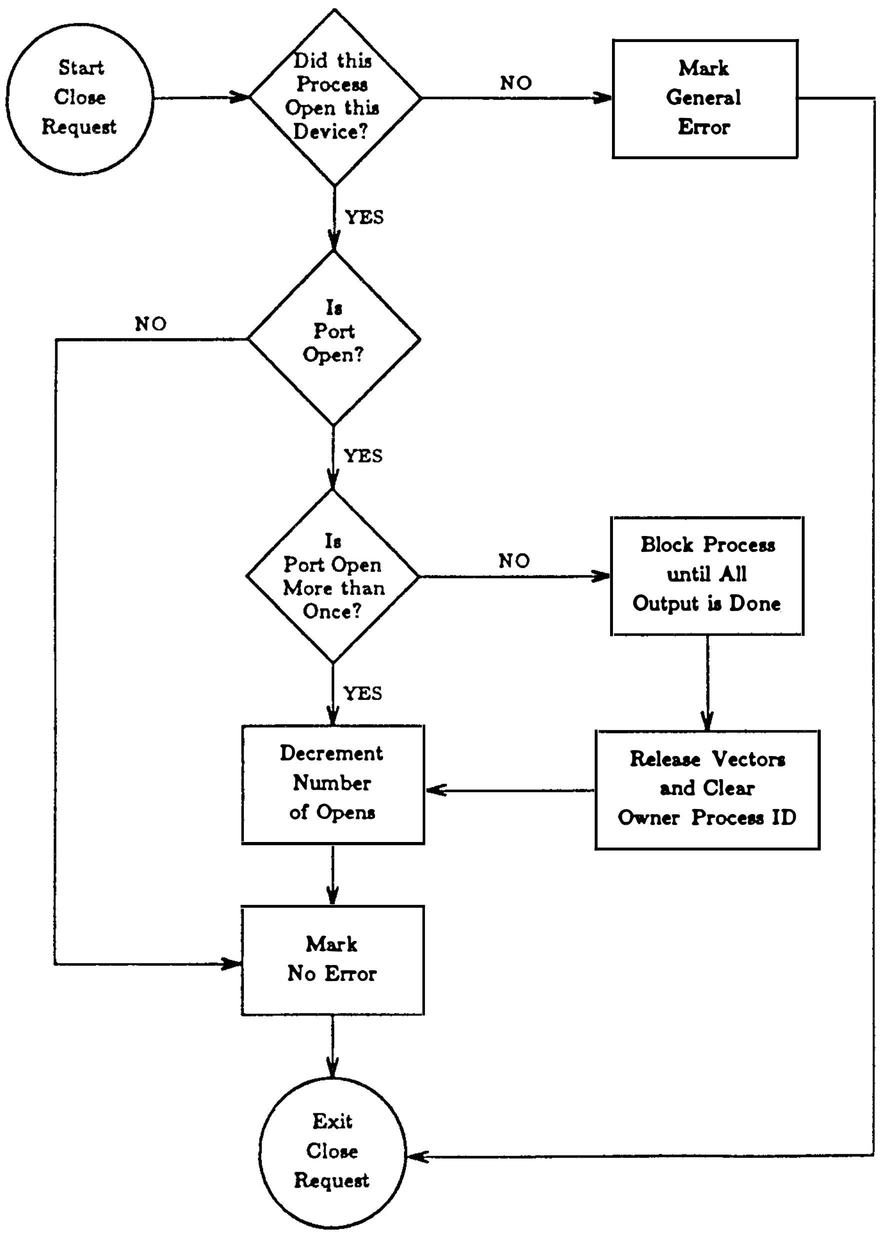

Start Write Request
Did this Process Open this Device?
NO
Mark General Error
YES
Convert 32-bit Physical Address to Sel:offset or Seg:offset
Get Character from Sel:offset or Seg:offset
Attempt to Place Character in Output Queue
Increment Source Pointer
Make Sure Transmitter Empty Interrupt is Set to On
Was Character Queued?
NO
Block until Room in Buffer
YES
NO
Were all Characters Output?
YES
Mark Done
Exit Write Request

Figure 5.5 Close Request (Strategy Time)


Start Close Request
Did this Process Open this Device?
NO
Mark General Error
YES
Is Port Open?
NO
YES
Is Port Open More than Once?
NO
Block Process until All Output is Done
YES
Decrement Number of Opens
Release Vectors and Clear Owner Process ID
Mark No Error
Exit Close Request

5.3.3.5 The Interrupt Routine

The interrupt routine in the sample serial device driver is invoked when an interrupt occurs. The types of interrupts handled by the sample driver are as follows:

- Receiver line status interrupt
- Transmitter empty interrupt
- Data available interrupt

Receiver Line Status Interrupt

The sample serial device driver generates a receiver line status interrupt when an error occurs at the serial port. The interrupt service routine (ISR) for the receiver line status interrupt sets a flag to inform the next read request of the error status.

Transmitter Empty Interrupt

The sample serial device driver generates a transmitter empty interrupt when the serial device is ready to transmit a character. The transmitter empty ISR transfers characters from the output queue (characters placed in the queue as a result of a strategy routine write request) to the serial device.

If no characters are available in the output queue, the transmitter empty interrupt is disabled and the transmitter portion of the interrupt routine must wait for the strategy routine to restart the device.

The functional details of the transmitter empty interrupt routine are shown in Figure 5.6.

Data Available Interrupt

The serial device generates a data available interrupt when a character arrives at the serial port and is waiting for the device driver to read it. The data available ISR transfers any characters at the serial port to the input queue, where they are read by a read (strategy time) request. When the serial device is open, this routine is always awaiting characters.

The functional details of the data available interrupt are shown in Figure 5.7.

Figure 5.6 Transmitter Empty Interrupt

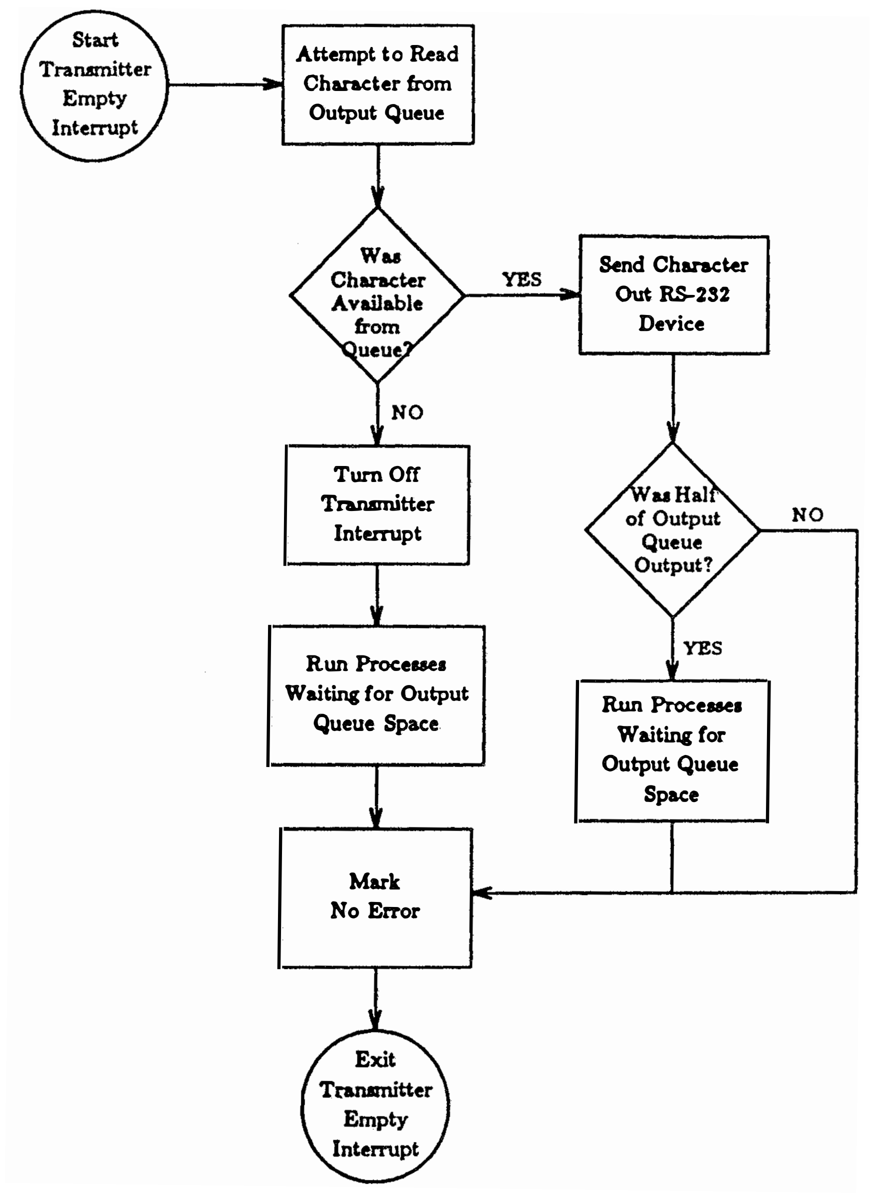
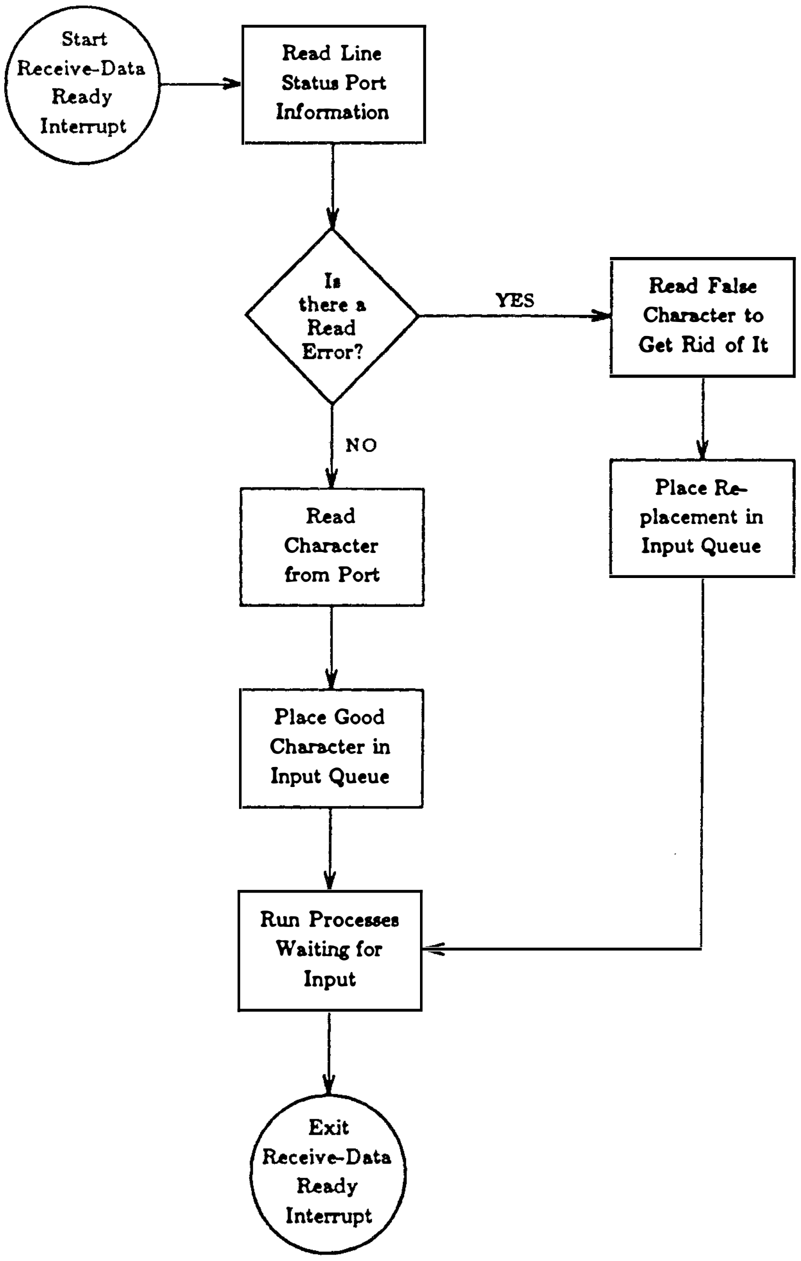

Start Transmitter Empty Interrupt
Attempt to Read Character from Output Queue
Was Character Available from Queue?
YES
Send Character Out RS-232 Device
NO
Turn Off Transmitter Interrupt
Was Half of Output Queue Output?
NO
YES
Run Processes Waiting for Output Queue Space
Run Processes Waiting for Output Queue Space
Mark No Error
Exit Transmitter Empty Interrupt

Figure 5.7 Data Available Interrupt


Start Receive-Data Ready Interrupt
Read Line Status Port Information
Is there a Read Error?
YES
NO
Read False Character to Get Rid of It
Place Re-placement in Input Queue
Read Character from Port
Place Good Character in Input Queue
Run Processes Waiting for Input
Exit Receive-Data Ready Interrupt

5.3.4 Sample Serial Device Driver Development Approach

Following is a list of the development stages for the sample serial device driver. This development procedure is based on the MS OS/2 development procedure presented in Section 5.2, "General Development Approach."

1. List the functions you want the finished serial device driver to perform. For the sample serial device driver, the list includes the following:
   - initialize the device driver
   - output characters (interrupt driven)
   - input characters (interrupt driven)
   - return device status
2. Outline the sections of code you will need to write. Since you will develop a noninterrupt-driven version of the driver first, identify the sections of code you need for a minimally functional, noninterrupting driver.

   The noninterrupting portions of code include those that
   - initialize the device driver
   - return device status
3. Develop short support routines that will help perform each of these functions. For example, to initialize the device driver, you will need routines that will perform functions such as serial port line control and serial port baud rate.

   Test your routines, if possible, using MS-DOS 3.*x*. If testing under MS-DOS 3.*x* is not possible, use the MS OS/2 DOS-resident debugger to test your routines.
4. Integrate the routines you created in stage 3 into a device driver that will perform noninterrupting output. In this stage of development, you should create the logical structure of the driver to establish communication with the COM1 device.
5. Enable interrupts in the device driver code to perform interrupt-driven output.
6. Enable interrupts in the device driver code to perform interrupt-driven input.
7. Add any advanced features to the code in this stage. For example, if the sample serial device driver were to support MS-DOS 3.*x* box features, the necessary code would be added to the device driver in this stage.

It is important to perform testing in each stage of development, so that you can isolate any problems and verify your work before adding more complex functions.

5.3.4.1 The Write Request Code

This section describes the sample serial device driver code for a write request. You should examine the following lists of steps completed by the device driver while you study the actual device driver code. This should help you become familiar with the operation of the device driver and the structure of the device driver code. You may want to print a listing of the serial device driver code so you can refer to it as you study this section. See also Figure 5.4.

When MS OS/2 makes a write request, the following events occur:

1. MS OS/2 calls the strategy routine with an open request. The strategy routine then attempts to reserve the interrupts it will need in order to complete the I/O request.
2. MS OS/2 calls the strategy routine with a write request.
3. The strategy routine transfers write request characters to an output queue for processing by the interrupt routine. If the queue is full, the process is blocked until room becomes available. This routine enables the transmitter empty interrupt.
4. MS OS/2 repeats steps 2 and 3 until all characters are placed in the output queue.
5. When the transmitter empty interrupt occurs, the interrupt routine attempts to transfer a character from the output queue to the serial device. If the output queue is empty, the transmitter interrupt is disabled and processes waiting for output queue space are allowed to execute.
6. The serial device driver repeats step five whenever the strategy routine places characters in the output queue.
7. MS OS/2 calls the strategy routine with a close request after all write requests are processed.
8. The close request terminates.

5.3.5 Debugging the Sample Serial Device Driver

This section describes and demonstrates techniques for debugging MS OS/2 device drivers. During the sample debugging session presented here, you will become familiar with basic device driver operations and debugging procedures. In addition, this section provides some suggestions that will help you debug the MS OS/2 sample serial device driver. General MS OS/2 debugging procedures for all types of device drivers are described in Chapter 3, "Using the DOS-resident Debugger."

Some of the concepts you will use in this section are

- breaking code execution with INT3H interrupts
- setting DOS-resident debugger breakpoints in the code

Breaking Execution Code with the INT 3H Interrupt

The sample serial device driver uses INT 3H interrupts to cause a break in the execution of the source code. When the device driver code is executed and an INT 3H is encountered, a break occurs and control is transferred to the DOS-resident debugger.

INT 3H is useful for finding the memory (segment/selector) location of a device driver. For example, suppose an INT 3H is placed in the code at the beginning of the strategy routine. When the INT 3H is encountered, execution will break and you can obtain the code and data selector used by the strategy routine.

Note

A *selector* indicates a segment register value if your computer is running in protected mode. A *segment* indicates the segment register value if your computer is running in real mode. In the remainder of this chapter, *selector* refers to the segment register value in both real and protected modes. Any segment/selector differences are mentioned specifically.

If an INT 3H is placed in the strategy routine of the sample serial device driver code, the code and data segment selectors may change between initialization (init) time and task time, since MS OS/2 reuses and relocates the system's selectors during the device driver initialization process.

Setting DOS-resident Debugger Breakpoints

You use the symbolic commands of the DOS-resident debugger to set breakpoints in the sample serial device driver code. These breakpoints break execution of the code and transfer control to the DOS-resident debugger.

Note that if a breakpoint is set during initialization time, the code segment selector will be incorrect when task time occurs, preventing the break in execution.

For information about setting, enabling, and disabling breakpoints, see Chapter 3, "Using the DOS-resident Debugger."

5.3.5.1 Examining the Device Driver Code

If you want to examine the device driver code or make modifications to the device driver, you will want to view the code using the DOS-resident debugger. This section provides a sample debugging session. During this session, you will view the various routines that make up the sample serial device driver and set, enable, and disable some DOS-resident debugger breakpoints. This will allow you to watch the execution of selected device driver functions.

Requirements

To complete the debugging session for the sample serial device driver, you need the following:

- a terminal installed on the COM1 serial port of your target system to interact with the sample serial device driver. This terminal must run at 9600 baud with 8 data bits, 1 stop bit, no parity, and no handshaking.
- a terminal installed on the COM2 serial port of your target system to support the DOS-resident debugger. See Chapter 3, "Using the DOS-resident Debugger," for information about configuring the COM2 terminal.
- a bootable system disk that contains the following:
  - the DOS-resident debugger
  - the **debug** version of the sample serial device driver (*comsam2d.sys*) that is created when you run the **make** program to update and link the device driver modules. (To ensure that the procedure will work as described, make sure that there are no other debug versions of device drivers on the disk.)

- the symbol file (*comsam2d.sym*) for the **debug** version of the sample serial device driver that is created when you run the **make** program
- the *config.sys* file, which must contain the entry **device=comsam2d.sys**
- the testing program to exercise the sample serial device driver (*comtest.exe*)

- an assembled source code listing you can follow during the sample debugging session

The Hands-on Debugging Session

This hands-on debugging session will guide you through the sample serial device driver code in four parts:

- initialization
- writes
- interrupts
- reads from the device driver

In each part of the session, you will see the various routines that make up the device driver. You will also use the DOS-resident debugger to set and change some breakpoints so that you can follow the execution of the routines. If you are uncertain about the syntax for a command, refer to the description of the command in Chapter 3, "Using the DOS-resident Debugger," or display the DOS-resident debugger help screen by typing the following at your debug terminal:

```
?
```

Follow these steps to view the initialization portion of the sample serial device driver code:

1. Boot the **debug** version ofM S OS/2. MS OS/2 displays the following prompt on your DOS-resident debugger terminal:

   ```
   #
   ```

2. Following the prompt on your console, type the following command and press RETURN:

   ```
   g
   ```

   This executes initialization portion of the boot process. MS OS/2 will execute until it attempts to initialize the sample serial device driver and encounters the INT3H in the strategy routine. When the INT 3H is reached, execution breaks and the # prompt reappears on your terminal.

3. Execute to the label PULL_REQUEST by typing

   g PULL_REQUEST

   When this code is executed, the strategy routine will obtain the command (an INIT command) from the MS OS/2 request packet and move a value for the function into register AL.

4. Execute to the JMP_REQUEST section of the strategy routine by typing

   g JMP_REQUEST

   JMP_REQUEST converts the command into an offset of the desired request handler routine. Since MS OS/2 is executing an INIT command, control is transferred to the COM_INIT handler to complete the request.

5. Execute to the COM_INIT routine by typing

   g COM_INIT

   COM_INIT performs the following functions:

   - saves the address of the **DevHlp** routine
   - saves the process ID (PID)
   - ensures that the proper COM port exists and is located at the proper address

6. Execute to the location where the PID address is stored by typing

   g SAVE_PID

   SAVE_PID retrieves the address of the PID storage location in MS OS/2 and saves it in its own data area.

7. Execute to the I IT_PARMS routine by typing

   g INIT_PARMS

   INIT_PARMS examines the hardware to make sure that the COM port exists and is located at the proper address.

You have now seen the initialization routines in the sample serial device driver. In the following section, you will view the task time portion of the sample serial device driver. I/O for the sample serial device driver is performed in this portion of the code.

To view the I/O operations, follow these steps:

1. Transfer control from the DOS-resident debugger to the MS OS/2 session manager by typing

   g

   The session manager displays a menu of options.

2. Select the "Start a Program" option. When executed, this routine will start up a protected-mode MS OS/2 command interpreter (*cmd.exe*). This interpreter allows you to run protected-mode applications.
3. Run the *comtest* program by typing

   comtest

   This program will perform reads and writes to exercise the sample serial device driver. Later steps will trace the input and output execution of *comtest.exe* using the DOS-resident debugger.

   When MS OS/2 issues a request to the strategy routine, control transfers from the session manager to the DOS-resident debugger.
4. When control is transferred to the DOS-resident debugger, the INT 3H breakpoint is no longer needed. Change the INT 3H at the beginning of the strategy routine to a NOP by typing the following (the DOS-resident debugger displays the information shown here in italics):

   ECS:IP
   *CC.***90**
5. Set a "sticky" breakpoint at the current CS:IP register by typing

   bp CS:IP

   This will help control the execution of the sample serial device driver by allowing the breakpoint to be disabled or enabled at any point.

   Note that the selector values (CS and DS) may be different than they were during the INIT process, since MS OS/2 reuses the selectors. These new selector values will remain in effect any time the sample serial device driver is called on to perform I/O.

   MS OS/2 first issues an open request to the sample serial device driver.
6. Execute to the COM_OPEN routine by typing

   g COM_OPEN

   COM_OPEN sets up the serial device. Access to this entry point is controlled through the PID. When COM_OPEN is called, the PID is checked to make sure that no other processes have control of the device.
7. Execute to the SET_PARMS routine by typing

   g SET_PARMS

   SET_PARMS sets up the 8250 asynchronous communications chip by calling the following routines:

Routine	Description
SET_LINEC	Sets the communication parameters, such as stop bits, parity, and data bits.
SET_BAUD	Sets the baud rate.

8. Execute to the area of code where the parameter routines are loaded by typing

   g SET_UP_DEVICE

   Note that the baud rate set in the SET_BAUD routine is contained in the register AX. If your terminal requires a different baud rate, you should modify it now.

9. Set a breakpoint in the COM_WRITE routine by typing

   bp COM_WRITE

10. Continue execution of the sample serial device driver by typing

    g

    When MS OS/2 encounters the breakpoint you set in the COM_WRITE routine, execution will break. MS OS/2 will have passed through the strategy routine and issued a write request to the sample serial device driver. The DOS-resident debugger will have control.

11. Execute to the GET_VIRT_ADDR label by typing

    g GET_VIRT_ADDR

    GET_VIRT_ADDR converts the 32-bit physical address that is passed to the sample serial device driver in the request packet into a usable segment/selector and offset.

    Note that the **DevHlp** routine **PhysToVirt** is the routine that actually converts the address to a usable value and places it in the register DS:SI. In the sample serial device driver, DS:SI is a pointer to the string to be output.

    To see the information on the new selector, type the **dl** (display LDT) command. This command displays information such as the segment type, starting 24-bit base address, and the address and size of the segment.

Note

The **dl** command always works in protected mode; however, if you are in real mode and you used **loadall** to generate the address, a selector is not entered into the LDT and the **dl** command is not referencing a valid selector.

After the value is used, the **DevHlp** routine **UnPhysToVirt** must inform MS OS/2 that the address (temporary selector) is no longer needed.

12. Execute to the SEND_CHAR routine by typing

    g SEND_C AR

    SEND_CHAR sends a character to the output queue or blocks the calling request until room becomes available in the output queue. This routine also verifies that the transmitter empty interrupt is enabled. If this routine is blocked, the interrupt routine restarts it when room becomes available in the output queue.

13. Execute to the top of the loop that receives characters from the source and places them into the output queue by typing

    g COM_WRITE_LOOP

    This loop is repeated until the output queue is empty.

    The following MS OS/2 message should appear on your COM1 terminal:

    ```
    DOS COM1 Tester
    ```

14. Disable the COM_WRITE breakpoint by listing the current breakpoints (use the **bl** command) and finding the one located at the COM_WRITE label.

You have now seen some characters output to the COM device. Next, you will see the execution of an interrupt.

Follow these steps to view the execution of an interrupt:

1. Set a breakpoint on the interrupt routine by typing

   bp COM_INT

2. Resume execution of the serial code by typing

   g

   Wait for execution to break again when an interrupt occurs. The 8250 will generate an interrupt, since the breakpoint set on the sample serial device driver interrupt service routine is enabled. This will most likely happen when the transmitter register becomes empty or when a character is waiting to be read from the 8250.

3. When execution breaks, type the following command to execute to the COM_INT_AGAIN label:

g COM_INT_AGAIN

COM_INT_AGAIN uses the interrupt ID to determine the source of the interrupt. When the source is found, execution is dispatched to the proper ISR.

4. Find the definition of the interrupt identification register in the following table and determine which interrupt occurred and is pending service.

Interrupt ID Register			**Interrupt Source**
Bit 2	Bit 1	Bit 0	
1	1	0	Line state interrupt
1	0	0	Data available interrupt
0	1	0	Transmitter empty interrupt
0	0	0	Modem status interrupt

5. After determining the source of the interrupt, go to the ISR to service the pending interrupt and perform the indicated service.

The interrupt will be one of the following:

Interrupt	Description
LX_INT	Line status interrupt. Sets a flag for the next read request to indicate that an error has occurred. Type **g** and wait for the next break to occur.
RX_INT	Data available interrupt. Informs MS OS/2 of a character waiting to be read from the 8250 communications chip (execute to the WRITE_CHAR label to see the character read and placed in the input queue by the **DevHlp** routine WRITE_CHAR). To see this operation performed, type **g INT_PUSH_CHAR_Q** The character will be read from the 8250 and placed in the input queue. If no room is available in the input queue, a buffer overflow will occur and the character will be lost.
TX_INT	Transmitter empty interrupt. Informs MS OS/2 that the 8250 is available for the next character to transmit.
MX_INT	Modem service interrupt. Serves no function in the sample serial device driver, but must be supported for the interrupt ID register.

6. To view the operation of any of these interrupts, type

   g

   Then wait for the next break in execution.

You have now viewed the operation of the interrupts and seen how they are handled by the routines. The following section allows you to follow a particular interrupt and view the functions it performs.

To trace a particular interrupt, follow these steps:

1. Disable the DOS-resident debugger breakpoint for COM_INT and set a breakpoint on the ISR you want to trace.
2. Trace through the interrupt. You will not have to determine which interrupt occurred.
3. Disable the breakpoint for any interrupts you want to view.
4. Press any key on the COM1 terminal device to clear any pending read requests.
5. Type a few characters at the COM1 terminal. These characters will be read by the RX_INT ISR and placed in the input queue to be read by the COM_READ routine during the next read request. If the input queue is full, a buffer overflow will occur and the characters will be lost.
6. Execute to the area of code where the read routine tries to remove a character from the input queue by typing

   g PULL_CHAR_Q

   If no characters are available to be read, the calling request is blocked until some become available.
7. Execute to the area of code where characters are placed at the destination address (the address found by the **DevHlp PhysToVirt** call) by typing

   g READ_PASSED

   This process continues until all needed characters are read. When the requested characters are read, the routine will mark that no outstanding reads are left and will return control to MS OS/2.

The steps and operations that have been described should help in writing MS OS/2 device drivers. Although most of these steps are related directly to the serial device driver, most of the concepts can be applied to the development of other MS OS/2 device drivers.

5.4 Sample Disk Device Driver

The MS OS/2 disk device driver provided in the Device Driver Kit acts as an interface between MS OS/2 and the disk devices in your system. This interface lets your system perform multiple queued requests in an order that attempts to minimize movement of the access mechanism across the disk. This section describes the MS OS/2 disk device driver and the advanced techniques it uses to perform efficiently in the MS OS/2 multitasking environment.

The device driver described here is the disk device driver used by MS OS/2 in the MS OS/2 SDK implementation. After reading this section and studying the device driver code, you can apply the advanced techniques and concepts presented here to write similar block device drivers for your MS OS/2 system. Section 5.4.8, "Development Approach," describes the development procedure for the MS OS/2 disk device driver.

The concepts you will learn in this section include

- Using semaphores
- Using ROM BIOS interlocking to prevent interference from interrupts issued from the 3.*x* box
- Optimizing disk access with queue sorting
- Locking a memory segment
- Mapping a physical address to a virtual address
- Using IOCtl functions to perform disk operations

After completing this section, you should be able to make any modifications necessary for the MS OS/2 disk device driver to operate with your system. For information on using the DOS-resident debugger to assist you in testing your modifications, see Chapter 3, "Using the DOS-resident Debugger."

5.4.1 Disk Driver Organization

The disk driver is composed of four major sections, as follows:

- Strategy routine
- Interrupt routine
- ROM BIOS interlock
- Generic IOCtl functions

The function of each section is described later in this section.

5.4.2 File Organization

The disk device driver code is located in the following files:

File	Contents
abinit.asm	Device driver initialization code
absubr.asm	Common subroutines for the device driver
abdsk1.asm	Strategy routine and command processors
atcache.asm	Cache support for hard disks
atdsk2.asm	Floppy disk state machine and interrupt routine, and low-level code for accessing the 765 disk controller
atdsk3.asm	Hard disk state machine and interrupt routine, and low-level disk controller code
abdsk4.asm	Timer services, including the timer interrupt service routine that is used only for turning the motor off for floppies (all other disk timer services are handled by the **DevHlp TickCount** routine)
abdsk5.asm	Partitioned disk support, including routines to set up tables for any extended (partitioned) volumes that may exist on a hard drive, and routines to handle command 22 (returns the number of partitionable fixed disks) and command 23 (returns the logical unit map)
abdsk6.asm	IOCtl support (includes physical drive IOCtl calls)
atint13.asm	Real mode INT 13H support (allows all operations to floppy disk drives and all operations to hard drives, except write, write long, format, and init drive characteristics).

5.4.3 Request Packets

Following are the device driver commands found in the request packets handled by the MS OS/2 disk device driver:

Code	Command
0	Init
1	Media check
2	Build BPB
4	Read
8	Write
9	Write with verify
15	Removable media
16	Generic IOCtl
17	Reset uncertain media
18	Get logical drive map
19	Set logical drive map
21	Get number of partitions
23	Get unit map
24	Read, no cache
25	Write, no cache
26	Write/verify, no cache

The sample disk device driver processes the following Generic IOCtl functions (command 16):

Install BDS
Get device parameters (logical)
Get device parameters (physical)
Read track (logical)
Read track (physical)
Verify track (logical)
Verify track (physical)
Set device parameters (logical)
Set device parameters (physical)
Write track (logical)
Write track (physical)
Format track (logical)
Format track (physical)

The individual device driver commands and Generic IOCtl functions listed are described in detail in the *Microsoft Operating System/2 Device Drivers Guide.* Some of the request packets just listed are processed by the strategy routine, while others must be processed by the interrupt routine.

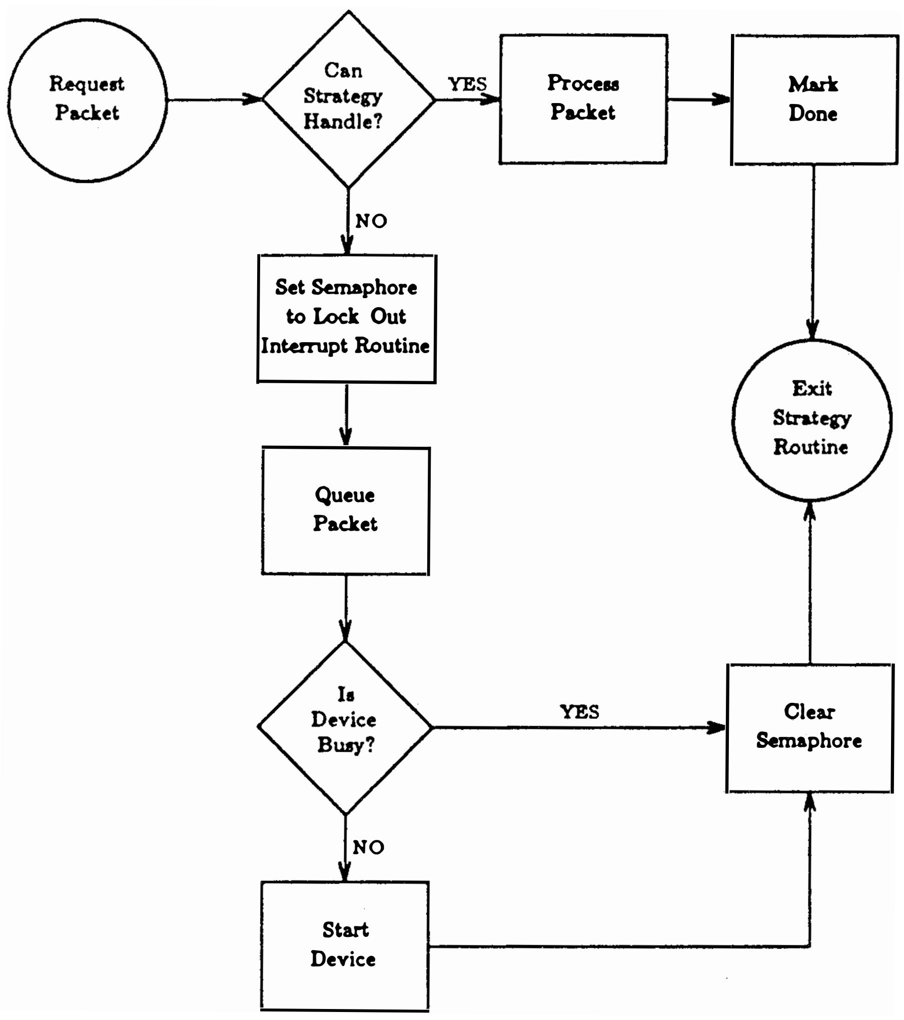
5.4.4 The Strategy Routine

When MS OS/2 issues an I/O request packet for the disk device driver, the strategy routine is called. The strategy routine services the request immediately by either processing the request itself or putting it in a queue for the interrupt routine to process.

If the strategy routine cannot process the request immediately, and if the disk device is idle, the strategy routine starts the device and then returns to the kernel. If the disk device is active, the request is left in the queue, and the strategy routine returns to the kernel. If necessary, the packet is then processed by the interrupt routine.

Figure 5.8 shows the operational flow of the strategy routine.

Figure 5.8 Strategy Routine Operation



5.4.4.1 Request Packets Processed by the Strategy Routine

All request packets are processed in some way by the strategy routine. In some cases, the strategy routine can process the packet immediately without having to place it in a queue. This is generally true for request packets with device driver commands that do not perform reads or writes to the disk. In general, the interrupt routine must process request packets that contain read or write commands.

The strategy routine can handle the following request packets immediately:

Code	Command
0	Init
1	Media check (unless changeline is supported)
2	Build BPB (unless the BPB is not in the boot sector)
6	Input status
7	Input flush
13	Open
14	Close
15	Removable media
16	Generic IOCtl (those functions not accessing the disk)
17	Reset uncertain media
18	Get logical drive map
19	Set logical drive map

The interrupt routine must also process all other request packets handled by the disk driver. The device driver commands for those packets processed by the interrupt routine are listed later in this section.

5.4.4.2 DevHlp Routines Used by the Strategy Routine

The device helper (**DevHlp**) routines, as described in the *Microsoft Operating System/2 Device Drivers Guide,* provide an interface to operating system services. The strategy and interrupt routines use **DevHlp** routines to provide services for the MS OS/2 disk device driver. The **DevHlp** routines that the strategy routine uses perform the following functions for the MS OS/2 disk device driver:

Routine	Description
SortReqPacket	Inserts new requests in the work queue according to the starting sector number on the disk
PhysToVirt	Converts a 32-bit physical memory address to a segment:offset in real mode or a selector:offset in protected mode
VirtToPhys	Converts a segment:offset or selector:offset pair to a 32-bit physical address
UnPhysToVirt	Informs MS OS/2 that the virtual address provided by **PhysToVirt** is no longer needed
AllocReqPacket	Reserves system memory for use as a request packet, then returns a pointer to the request packet
FreeReqPacket	Releases an allocated request packet
SetIRQ	Sets up handlers for the floppy interrupt and the fixed disk interrupt
TickCount	Sets up the drive timer handlers
SetROMVector	Replaces a real-mode software interrupt handler with a handler from the device driver for INT 13H interlocking
GetDOSVar	Gets the timer interval while setting up the drive timer handlers
Lock	Locks segments used for passing parameters between user programs and the IOCtl driver functions
UnLock	Unlocks the IOCtl packets after completion of the operation
ProcBlock	Blocks a process requesting a serial resource, such as the INT 13H ROM code, until the resource is available
ProcRun	Unblocks a process when the serial resource becomes available

5.4.5 The Interrupt Routine

The interrupt routine completes the processing of queued requests. When there is new work in the queue, the interrupt routine redrives the disk device. The interrupt routine then indicates that the previous operation is complete and unblocks any threads waiting for this request to complete.

5.4.5.1 Request Packets Processed by the Interrupt Routine

The following request packets are processed by the interrupt routine after the strategy routine places them in a queue:

Code	Command
1	Media check (if changeline is supported)
2	Build BPB (if the BPB is not in the boot sector)
4	Read
5	Nondestructive read
8	Write
16	Generic IOCtl (those functions that access the disk)

For a description of each of these commands, see the *Microsoft Operating System/2 Device Drivers Guide.*

5.4.5.2 DevHlp Routines Used by the Interrupt Routine

The **DevHlp** routines that the interrupt routine uses perform the following functions in the MS OS/2 disk device driver:

Routine	Description
PullRequest	Pulls the specified packet address from the disk device queue
DevDone	When called by the disk driver, signifies that a request is completed
ROMCritSection	Used for INT 13H ROM BIOS interlocking to flag a critical section in the ROM BIOS, preventing the 3.*x* box from being frozen

5.4.5.3 How the Interrupt Routine is Entered

The interrupt routine is entered through one of the following interrupts:

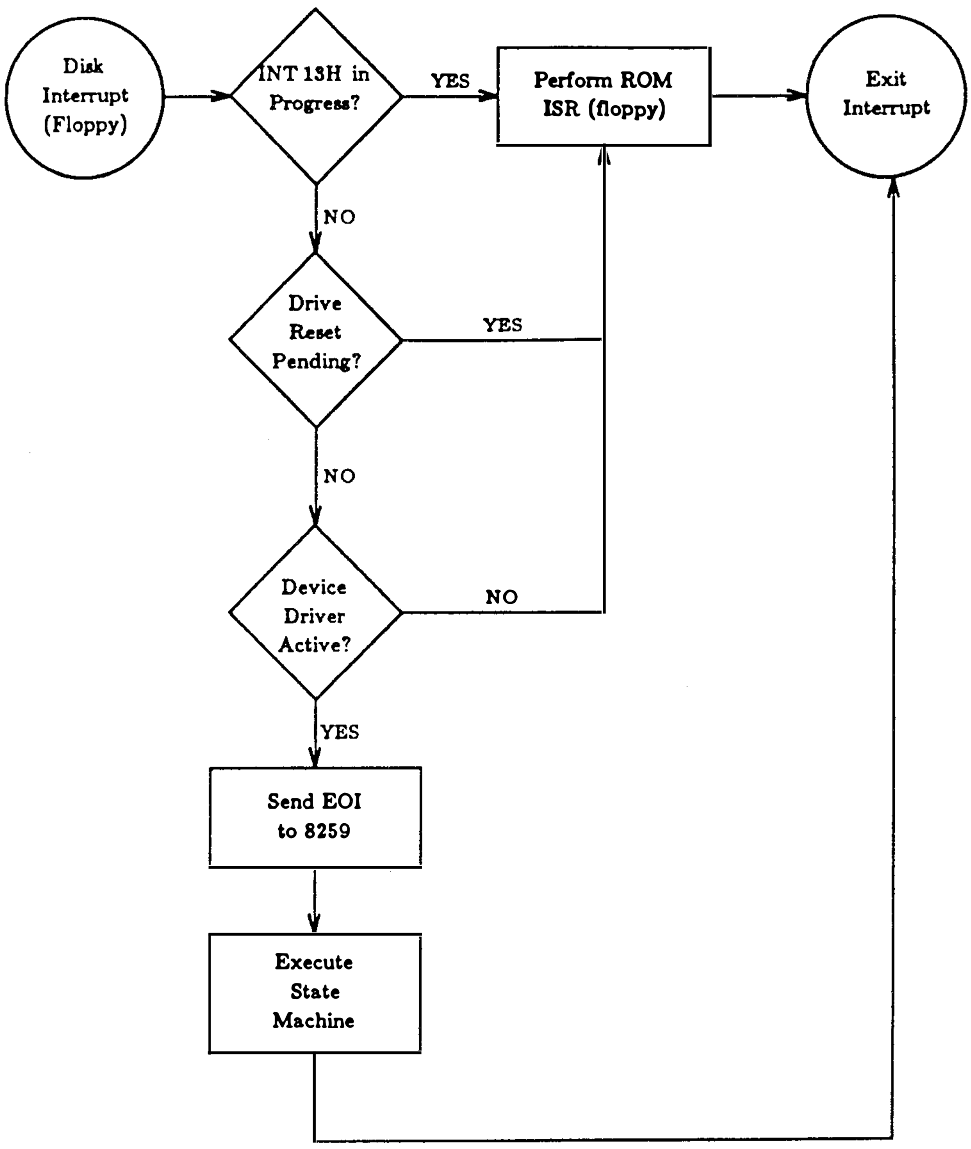
- Floppy disk interrupt
- Fixed disk interrupt
- Timer interrupt

Disk Interrupt (Floppy)

The floppy disk interrupt is issued by the floppy disk controller upon the completion of most commands. For example, when the **seek** command completes, the controller issues the disk interrupt, which is used to advance the state machine from one state to the next.

Figure 5.9 shows the operational flow of the floppy disk interrupt.

Figure 5.9 Disk Interrupt (Floppy)

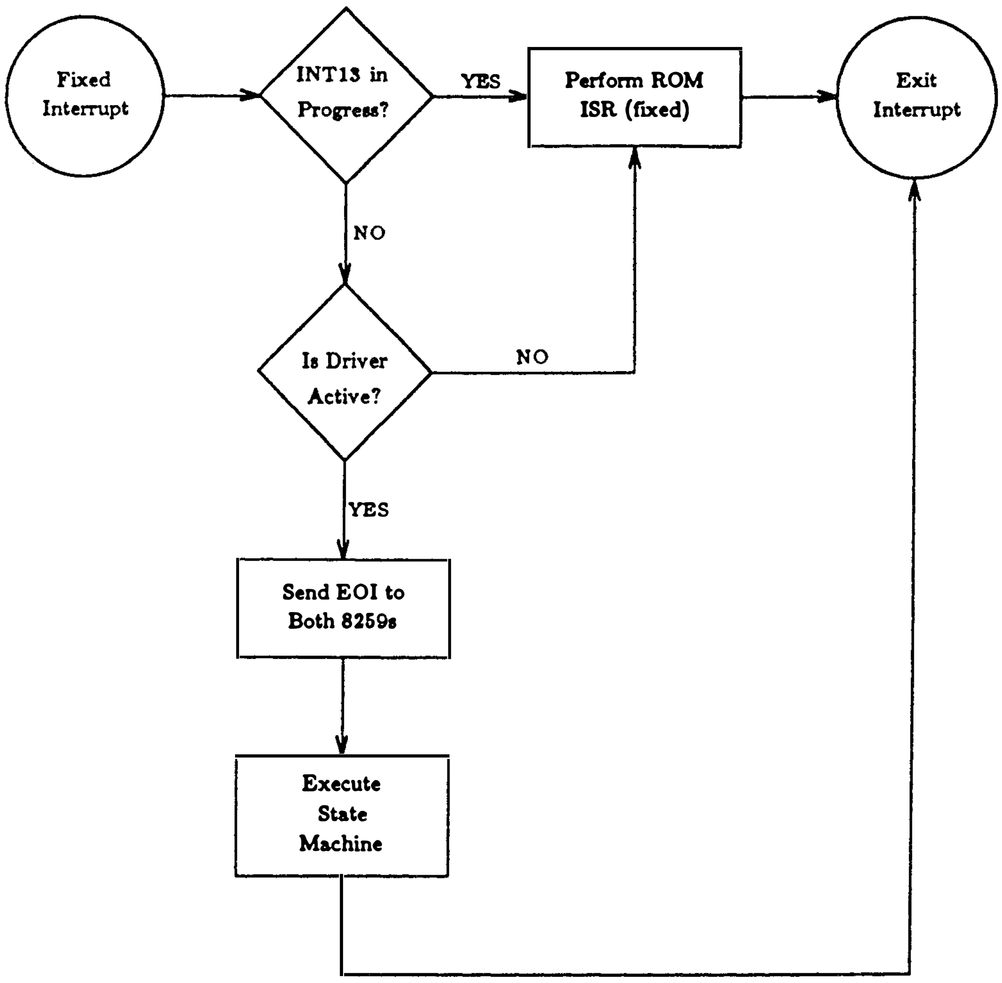


Fixed Interrupt

The fixed interrupt is issued by the fixed disk controller upon completion of most commands. For example, to read a sector, the fixed disk controller issues a fixed interrupt after the data is read from the disk. This advances the state machine to the state that will read the data into memory from the disk controller. Note that since DMA is not used to transfer the data, the device driver must move the data itself.

Figure 5.10 shows the operational flow of the fixed disk interrupt.

Figure 5.10 Fixed Disk Interrupt



Timer Interrupt

The timer interrupt is used for the following purposes:

1. To check for time-out errors during an operation

   For example, if an operation cannot complete because there is no disk in the drive, the timer interrupt will signal the time-out.

2. To time certain events

   For example, a certain amount of time is allowed for the head to settle when a disk drive is being used. The timer interrupt will interrupt when this time has passed.

This device driver uses two timer entry points. **DriveTimer** is called periodically to turn off the floppy drive motors after the last disk access. **FloppyTimer** is used to time certain events, such as head settle time, for floppy disks. Both timers use the **DevHlp TickCount** routine to provide the timer services. There is also a facility to time certain events for the hard drive, but it is not used by this driver.

Figures 5.11 and 5.12 show the operational flow of the timer interrupts.

Figure 5.11 Drive Time Interrupt

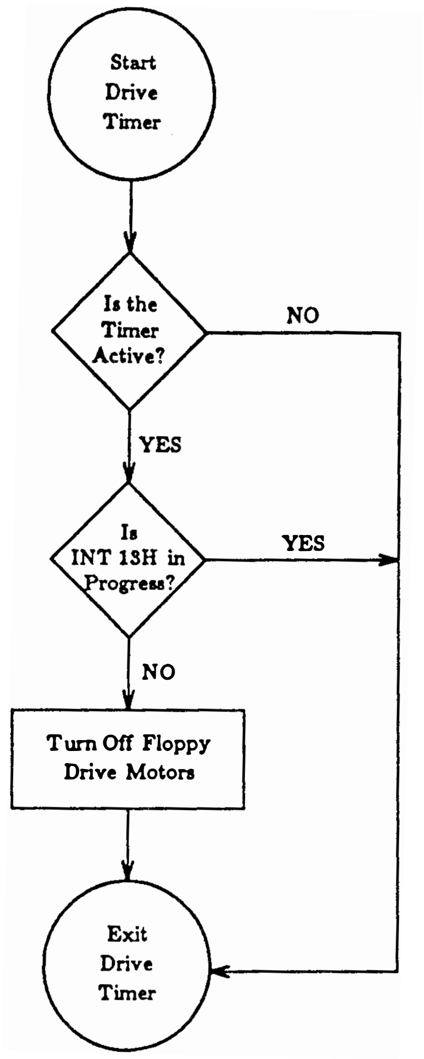
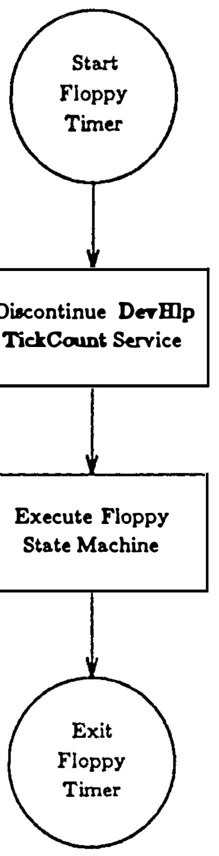


Figure 5.12 Floppy Timer Interrupt


Start Floppy Timer
Discontinue DevHlp TickCount Service
Execute Floppy State Machine
Exit Floppy Timer

5.4.5.4 The State Machine

The interrupt routine calls the MS OS/2 state machine to handle requests that the strategy routine cannot execute. The state machine is composed of several discrete states or functions. Each state is composed of certain instructions and conditions and contains a transition to the next state. A transition occurs only after the instructions for the state are completed or an error is identified.

The MS OS/2 disk device driver contains two separate state machines: a floppy disk state machine and a fixed disk state machine. These state machines are described in the two following sections.

The Floppy Disk State Machine

The floppy disk state machine used by the MS OS/2 disk device driver is composed of the following states:

0	**Start**
1	**Calc**
2	**Select**
3	**Recal**
4	**Seek**
5	**Settle**
6	**RdWri**
7	**Done**
8	**Idle**
9	**Error**
10	**Rset**
11	**TimOut**
15	**ChkChngLine**
16	**SeekReset1**
17	**SeekReset0**
18	**SenseSeek0**

Each of these states is described in detail in the following list (note that states 12–14 are used by the fixed disk state machine):

Start:

```
If this is not an error retry
    move request parameters to local variables
Next state is Calc
Go to next state
```

```
Calc:
        Calculate head, sector, cylinder, count,
            and check for DMA wrap
        If this is a check for changeline request
            next state is ChkChngLine
        else
            next state is Select
        Select the drive through the controller
        If select is already done
            go to next state
        else
            return

Select:
        Set correct transfer rate for controller
        If the drive needs a recal
            next state is Recal
            start recal operation
            set FloppyTimer for timeout
            return
        Next state is Seek
        If the drive is not on the correct cylinder
            Start seek operation
            return
        Go to next state

Recal:
        If there was an error
            next state is Error
            go to next state
        Next state is Select
        Go to next state

Seek:
        If there was an error
            next state is Error
            go to next state
        Next state is Settle
        Set timer 1 for head settle time
        Return

Settle:
        Next state is RdWri
        Set FloppyTimer for timeout
        Start the operation (read, write, etc.)
        Return

RdWri:
        If there was an error
            next state is Error
```

```
          go to next state
     Update ROM variable at segment 40H
     Next state is Done
     Go to next state

Done:
     If the transfer was through a scratch buffer
          copy it out
     If operation is write plus verify, and it just did the write
          turn off write bit in packet (leaving verify only)
          next state is Settle
          go to next state
     If operation was the verify portion of write plus verify
          turn on write bit in packet
     If more sectors left in request
          adjust buffer address
          next state is Calc
          go to next state
     Mark request done and remove it from queue
     If queue is empty
          next state is Idle
     else
          next state is Start
     Go to next state

Idle
     Set timer 2 to turn off drive motor after delay time
     return

Error:
     Force recal on drive next time through
     If valid media established or drive not ready
          next state is Rset
          reset controller
          go to next state
     Assume media not established
     If all media types tried
          next state is Rset
          go to next state
     Try next media type
     Next state is Calc
     Go to next state

Rset:
     If the retry limit has not been reached
          increment retry count
          next state is Start
          go to next state
     Fail request (flag the error)
     Next state is Done
```

```
        Go to next state

TimOut:
        This state is reached via a timer interrupt
        Next state is Rset
        Force recal on drive next time through
        Set retry count to maximum (no retries allowed)
        Go to next state

ChkChngLine:
        Read controller port to check for changeline active
        Fill in changeline state variable
        If changeline not active
            next state is Done
            go to next state
        Next state is SeekReset1
        Reset controller
        Go to next state

SeekReset1:
        Next state is SeekReset0
        Start seek operation to cyl 1
        If seek complete
            go to next state
        else
            return

SeekReset0:
        Next state is SenseSeek0
        Start seek operation to cyl 0
        If seek complete
            go to next state
        else
            return

SenseSeek0:
        Get controller status (ignore it)
        Next state is Done
        Go to next state
```

The Fixed Disk State Machine

The fixed disk state machine used by the MS OS/2 disk device driver is composed of the following states:

0	**Start**
1	**Calc**
2	**ParamSet**
3	**StartRead**
4	**StartWrite**
5	**StartVerify**
6	**Verify**
7	**Done**
8	**Idle**
9	**Error**
10	**CacheRead**
11	**CacheWrite**
12	**Read**
13	**Write**
14	**SetParam**

Each of these states is described in detail in the following list:

Start:

```
        Move request parameters to local variables
        Next state is Calc
        Go to next state
```

Calc:

```
        Calculate head, sector, cylinder, and count
        Set up data structure for cache

        If cache is not active or operation is not read or write
            next state is SetParam
            go to next state

        If this is a read operation
            next state is CacheRead

        If this is a write operation
            next state is CacheWrite
            go to next state
```

```
ParamSet:
        Get controller status
        If error
            next state is Error
        else
            if this is a read
                next state is StartRead
            else if this is a write
                next state is StartWrite
        else
            next state is StartVerify
        Set up command packet for controller
        Go to next state

StartRead
        Next state is Read
        Send command packet to controller
        If error
            go to next state
        else
            return

StartWrite
        Next state is Write
        Send command packet to controller
        If no error
            write data to controller
            return
        Next state is Error
        Go to next state

StartVerify
        Next state is Verify
        Send command packet to controller
        If error
            go to next state
        else
            return

Verify:
        Get controller status
        If no error
            next state is Done
        else
            next state is Error
        Go to next state

Done:
        If more sectors left in request
            advance sector number
        update count and address
        next state is **Calc**
        go to next state
    Mark request done and remove it from queue
    If queue is empty
        next state is **Idle**
    else
        next state is **Start**
    go to next state

Idle:
    Return

Error:
    If this is a Not Ready error or retry limit
          has been reached
             next state is Done
             get error code from controller
             map error code to DOS error code
        free any cache buffers for this request
    else
        increment retry count
        next state is Calc
        free any cache buffers for this request
    go to next state

CacheRead
    Next state is Done
    Calculate pointers to cache buffer and request buffers
    Transfer data from cache to request buffer
    Go to next state

CacheWrite
    Next state is SetParam
    Calculate pointers to cache buffer and request buffers
    Transfer data from request buffer to cache
    Go to next state
```

```
Read:
        Get controller status
        Get buffer address
        Convert to virtual address (PhysToVirt)
        Transfer data from controller
        If there was an error
            next state is Error
            go to next state
        Update buffer address
        If there are more sectors to go
            return (wait for next sector)
        If cache is active
            next state is CacheRead
        else
            next state is Done
        Go to next state

Write:
        Get controller status
        If there was an error
            next state is Error
            go to next state
        Adjust buffer address
        If there are more sectors to go
            write next block of data to controller
            return (wait for sector to be written)
        If this is not a write plus verify operation
            next state is Done
            go to next state
        Next state is StartVerify
        Set up command packet for controller
        Go to next state
            next state is Start

SetParam:
        Set up command packet for controller
        Next state is ParamSet
        Send command packet to controller
        If error
            go to next state
```

5.4.6 The ROM BIOS Interlock

Some applications require the device driver to intercept ROM BIOS software interrupts from the 3.*x* box. These ROM BIOS support routines run only in real mode. Without interlocking, the ROM BIOS routines directly access the hardware without respect to operations that may be in progress. Interference between ROM BIOS routines and currently executing MS OS/2 protected-mode operations could result in a fatal error, data destruction, or machine lock up.

Interlocking causes the ROM BIOS requests to be sent to the device driver. The device driver then allows the request to complete when the device becomes available.

The following figures describe the ROM BIOS interlocking technique used in the MS OS/2 disk device driver.

Figure 5.13 shows the separate paths of INT 13H (real mode) ROM BIOS requests (dashed line) and MS OS/2 (protected mode) disk requests (solid line), without the use of interlocking. Notice that there is no coordination between the two types of requests and the timing with which they reach the disk device.

Figure 5.14 shows the new path of INT 13H requests with the use of interlocking. Notice that the request is first sent to the MS OS/2 disk device driver, where it will wait until any currently executing MS OS/2 disk operations are complete. The request is then sent back to the ROM BIOS for processing.

Figure 5.13 Disk Requests without ROM BIOS Interlock

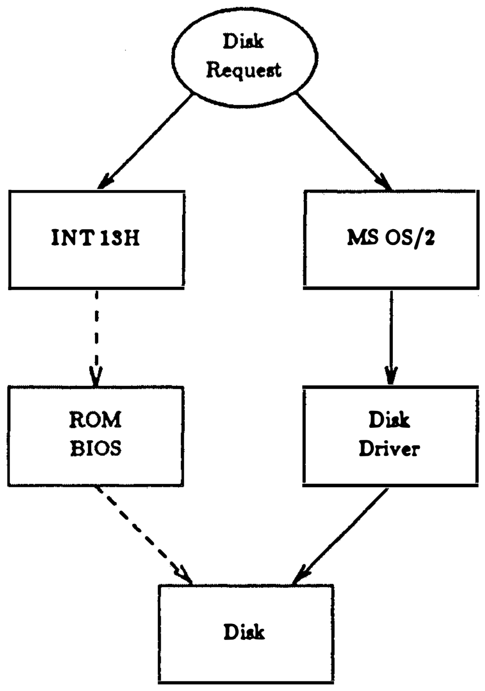
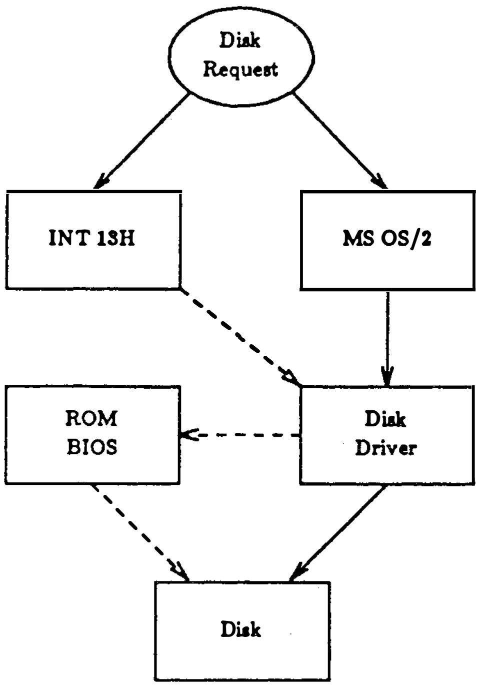


Figure 5.14 Disk Requests with ROM BIOS Interlock



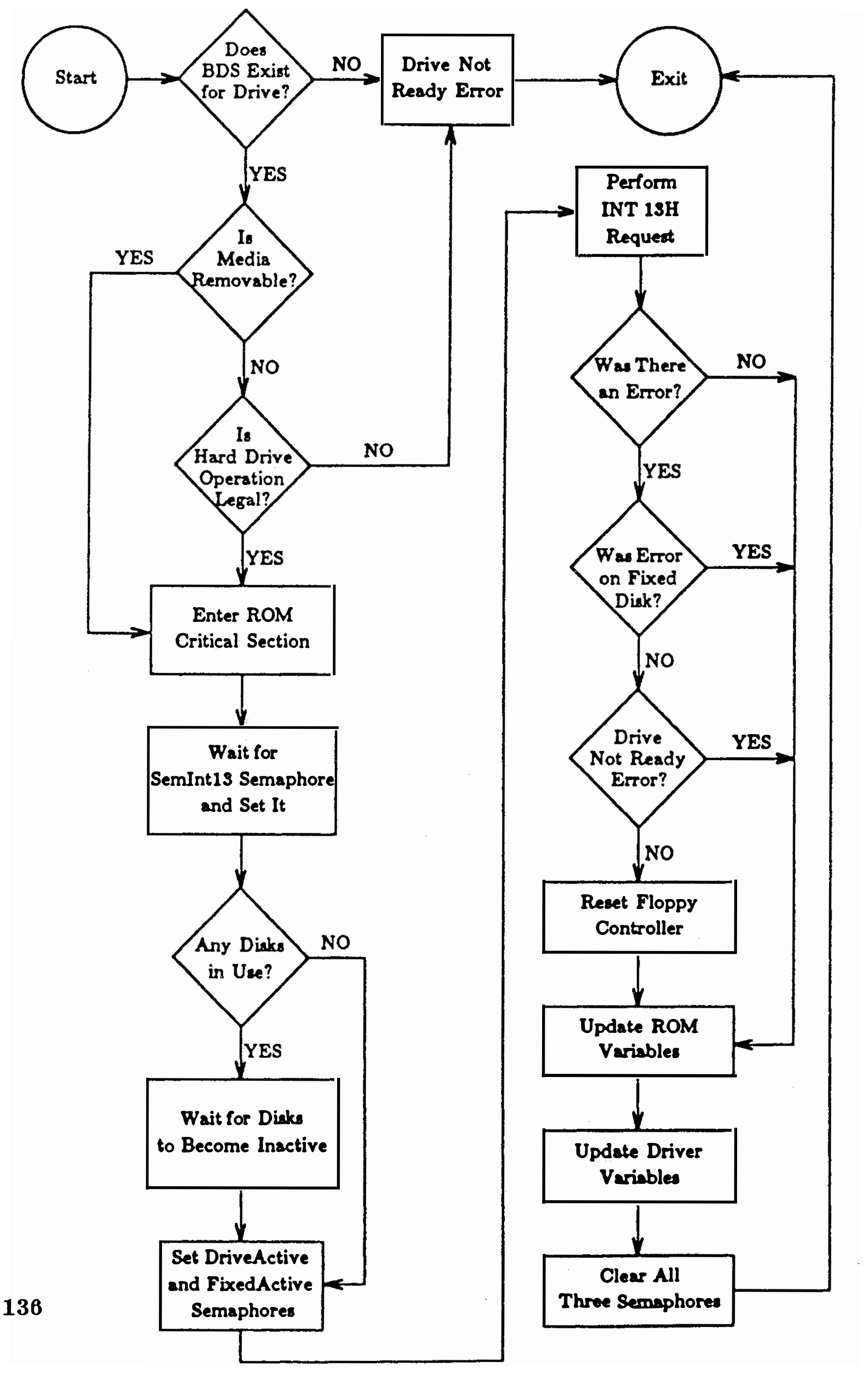
To provide the interlock mechanism, the disk device driver must intercept all INT 13H requests from real-mode applications. It can then set the semaphores used to interlock INT 13H requests with other pending requests, call the ROM or emulate the INT 13H function, then clean up and exit.

Figure 5.15 illustrates the stream of execution of the disk device driver's INT 13H intercept code. This code is executed whenever a real mode application issues an INT 13H.

The following is a list of the semaphores that the disk device driver sets:

Semaphore	Description
SemInt13	Serializes INT 13H requests by processes waiting for real mode INT 13H access so that only one INT 13H request can be processed at a time.
DriveActive	Indicates that a process is performing floppy disk I/O. If this semaphore is set in the INT 13H code, no other protected-mode process can access the floppy disk hardware until the INT 13H request is complete.
FixedActive	Indicates that a process is performing fixed disk I/O. If this semaphore is set in the INT 13H code, no other protected-mode process can access the fixed disk hardware until the INT 13H request is complete.

Figure 5.15 INT 13H Stream of Execution



5.4.7 Generic IOCtl Functions

The Generic IOCtl functions perform specialized services for disk-related operations. For more information about the Generic IOCtl functions, see the *Microsoft Operating System/2 Device Drivers Guide.*

5.4.8 Development Approach

To create a disk driver like the sample one provided, you should perform the following development tasks:

1. Outline the tasks you need to complete by listing the functions you want the finished device driver to perform. An outline for the MS OS/2 disk device driver might look like the following:

   Floppy disk I/O
   - Interrupt vector initialization
   - Disk read
   - Disk write, write/verify
   - Media check
   - Build BPB

   Fixed disk I/O
   - Interrupt vector initialization
   - Disk read
   - Disk write, write/verify
   - Perform media check
   - Build BPB

   IOCtl functions

   Additional functions
   - Open/close device
   - Check for removable media
   - Reset uncertain media

2. Separate the sections of code outlined in the first task into those that are interrupt-driven and those that can function without interrupts. You will develop a simple, noninterrupting version of the disk device driver first, then gradually add the more complex interrupting code.
3. Following your outline, get small portions of the code working for each function. As you develop the code, test each portion using small MS-DOS test routines.
4. Integrate portions of the code to create a noninterrupting disk device driver. In this stage of development, the driver should contain all portions of code that do not require interrupts. The noninterrupting driver should contain the state machine with busy states and the interrupt service routines that set flags for the floppy and hard disk interrupts.

5. Enable interrupts one at a time. The interrupts you need to enable are
   - Timer interrupt
   - Disk interrupt (floppy)
   - Disk interrupt (fixed)
6. Add ROM BIOS INT 13H interlocking and any additional device driver functionality.

5.4.9 Debugging the Sample Disk Device Driver

This section demonstrates some general techniques and suggestions for debugging an MS OS/2 block device driver. Many of the techniques presented in the debugging session for the sample serial device driver also apply to block device drivers. These techniques include placing INT 3H interrupts and breakpoints in the source code to break execution. If you are not familiar with these techniques, refer to the debugging session for the sample serial device driver in Section 5.3.4, "Sample Disk Device Driver," or see Chapter 3, "Using the DOS-resident Debugger."

The debugging session for the sample disk device driver is presented in two parts. The first part is a step-by-step walk-through of a read request to the floppy disk drive. The second part is a hands-on session with the sample disk device driver and the DOS-resident debugger. This part allows you to debug the device driver during the boot process. It also allows you to debug a real-mode INT 13H request as it is performed. This part of the debugging session demonstrates some of the steps that you will need to take to debug your own block device drivers.

5.4.9.1 A Read Request

The following is a description of a floppy disk read request. In this request, the operating system has just sent a read request packet to the strategy routine. The strategy routine first checks to see if it can handle the request itself. Since the strategy routine alone cannot handle read requests, it places the request in the queue and, assuming the device is not busy, starts the device.

When the read request is queued, the interrupt routine calls the state machine, which performs the following steps to complete the request:

1. Enters the **Start** state, which
   - moves the request parameters to local variables
   - sets the next state variable to **Calc**

- goes immediately to the next state

2. Enters the **Calc** state, which
   - calculates the data needed by the controller (head number, sector number, cylinder number)
   - sets the next state variable to **Select**
   - selects the drive (assuming that the drive has not already been selected)
   - returns from the strategy routine (all further processing of this request is handled by the interrupt routine)
   - sometime later, receives an interrupt from the floppy controller, signaling the completion of the select process
   - waits for the the drive Interrupt Service Routine (ISR) to send an end-of-interrupt (EOI) to the 8259 and jumps back to the state machine
3. Enters the **Select** state, which
   - sends the correct data transfer rate to the controller, assuming that the drive does not need to be processed in the **Recal** state
   - sets the next state variable to **Seek**, assuming that the drive is not already on the correct cylinder
   - sends the **Seek** command to the controller
   - returns from the interrupt routine
   - sometime later, receives an interrupt signaling the completion of the **Seek**
4. Enters the **Seek** state, which
   - sets the next state variable to **Settle**
   - sets timer 1 for the appropriate head settle time
   - returns from the interrupt routine
   - sometime later, receives a timer interrupt, signaling the completion of the head settle time
5. Enters the **Settle** state, which
   - sets the next state variable to **RdWri**
   - sets timer 2 in case of a timeout
   - sends the **read** command to the controller
   - returns from the interrupt routine
   - sometime later, receives a floppy interrupt, signaling the completion of the read request

6. Enters the **RdWri** state, which
   - updates the ROM variables at segment 40H
   - sets the next state variable to **Done**
   - immediately goes to the next state
7. Enters the **Done** state, which (assuming that there are no more sectors left in the request and no more requests in the queue)
   - marks the request as done by setting the **Done** bit
   - removes the request from the queue
   - sets the next state variable to **Idle**
   - goes immediately to the next state
8. Enters the **Idle** state, which
   - sets the timer to turn off the drive motor after the delay time
   - returns from the interrupt routine

Note

The data is sent by the controller by means of DMA.

5.4.9.2 Hands-on Debugging Session

The sample disk device driver is initialized and used extensively during the boot process. Therefore, it is one of the first device drivers that is needed to bring up MS OS/2. Since this driver must be functional in order to boot MS OS/2, the testing and debugging process for the driver is more complex than for other device drivers. Using the DOS-resident debugger, you can debug the disk device driver as it is called during boot. This section demonstrates some of the steps that you need to follow to bring up your disk driver for the first time.

Requirements

To complete the sample debugging session, you will need

- a bootable system disk containing
  — the debug version of MS OS/2.
  — the debug version of the sample disk device driver *disk01d.sys.* (If you copied the debug version of the serial device driver onto this disk, you will need to delete it.)

— the symbol file (*disk01d.sym*) for the debug version of the sample disk device driver.

— the testing program (*disktest.exe*) to exercise the sample disk device driver.

Note

When making the bootable diskette, *disk01d.sys* and *disk01d.sym* must be renamed to *disk01.sys* and *disk01.sym.*

- a terminal installed in your target system for use by the DOS-resident debugger
- a source listing of the disk device driver

Note

If you have installed the debug version of the sample serial device driver in the system, you will need to replace it with the standard version of the driver or remove it from the system.

If you are uncertain about the syntax of a command, see the description of that command in Chapter 3, "Using the DOS-resident Debugger," or display the DOS-resident debugger help screen by typing the following at your debug terminal:

?

Tracing the Initialization Code

To trace the initialization code, follow these steps:

1. Boot your MS OS/2 disk. Execution will break as the DOS-resident debugger is initialized. The debugger will display a register dump and the following prompt on your terminal:

   #

2. To continue the boot process, type the following command and press RETURN:

   g

When MS OS/2 encounters the INT 3H at the label **DriveInit** (*abinit.asm*), execution will break. This is the INIT entry point in the disk driver. INIT is the first disk request issued by MS OS/2 during bootup.

3. Note the value of the CS selector. The CS selector will be the code selector for all routines in the sample disk device driver, so you will need to remember this value or write it down. In the sample serial device driver, the protected-mode selectors may change after INIT time. In the sample disk device driver, however, the protected-mode selectors do not change after INIT time, because the sample it is a base device driver and is initialized earlier. The sample serial device driver is not a base device driver; it must be installed using the *config.sys* file.

   Like the **Init** code in the sample serial device driver, the sample disk device driver **Init** code saves the **DevHlp** address passed to it for future use. It also returns a pointer to a BPB array within the driver. MS OS/2 uses this array to determine the size of certain tables. For more information about BPBs, see the *Microsoft Operating System/2 Programmer's Reference.*

Continuing the Boot Sequence

You have seen the initialization code. Now, follow these steps to trace more of the boot process:

1. Set a breakpoint at the beginning of the strategy routine by typing

   bp DriveRequest

2. Continue booting by typing

   g

   When MS OS/2 makes the next call to the disk device driver, execution will break.

3. Display the contents of the request packet by typing

   d es:bx

   The third byte in the contents of the request packet is the command byte, which in this case is 11H, **Reset Media**.
   Reset Media is the first device driver call following the initialization.

4. You can now debug the **Reset Media** call. Execute to the beginning of the **DriveReset** routine by typing

   g DriveReset

5. Continue booting, rather than tracing through the **DriveReset** routine, by typing

   g

   Execution will again break at the strategy routine. The request packet now contains a **Read** function (4) to read a sector from the disk.

6. Disable the breakpoint at the **DriveRequest** routine.

7. Set a new breakpoint at **DriveWriteV** by typing

   bp DriveWriteV

   This is actually the same location in the code as **DriveRead**.

8. Execute to the breakpoint you just set by typing

   g

   This is the beginning of the strategy routine for the **Read** command.

   The routine performs some initial checks and sets up some variables. It then calls **DISK_IO** (*abdsk1.asm*) to start the actual operation.

9. Continue until **DISK_IO** is executed by typing

   g disk_io

10. Use the **p** (program trace) command to trace through **DISK_IO** until just before it calls **DevHlp** with **SortRequest**. This call to **DevHlp** will take the request packet pointed to by ES:BX and add it to the work queue for the disk device driver.

    You can return to this point at any time to check the contents of a request packet just before it is queued.

11. Execute to the beginning of the floppy drive state machine by typing

    g DriveExecute

12. Trace through this routine until you encounter a JMP instruction. Note that the current state (0) is put in register BL and is turned into an index in register BX. The JMP instruction will transfer control to the beginning of the **Start** state.

13. Continue until **FlExCalc** is executed by typing

    g FlExCalc

    This is the beginning of the **Calc** state.

14. Use the **p** command to trace through the **Calc** state until just after the call to **Sel765**. Notice that carry is not set, indicating that the drive is already selected. You do not have to wait for a completion interrupt.

15. Trace through the JNC instruction until you return to the beginning of the state machine.
16. Trace through **DriveExecute** again, noting that the current state variable is now 2 (**Select**). Continue tracing until you jump through the dispatch table to **FlExSelect**.
17. Using the source listing, disassemble the code until you find the call **Seek765** and move to the location directly following the call. Notice that the carry flag is not set, indicating that the head is already over the correct track.
18. Continue until **DriveExecute** is executed again by typing

    g DriveExecute

    The current state is now 5 (**Settle**).
19. Disassemble the code here, but do not trace it. Notice that **Settle** starts the I/O and sets the timer in case the operation hangs.
20. Continue until you reach **DriveExecute** again by typing

    g DriveExecute

    Since **Settle** just returned after starting the I/O operation, you will not encounter **DriveExecute** until the operation is complete and the interrupt occurs.
21. Trace through **DriveExecute**, noting that the current state is now 6 (**RdWri**).
22. Using the source listing, disassemble the code until you see the call to **Fini765** and continue to the location directly following that call.

    If register AX is not 0, an error has occurred and the state machine will retry the operation. If this is the case, type the following command to return to the **Settle** state:

    g FlExSettle

    Then type

    g DriveExecute

    Return to step 21.
23. At this point, the buffer specified in the request packet will contain data transferred by DMA. Continue until **FlExDone** is executed by typing

    g FlExDone

    This is the floppy **Done** state. Note in the source listing that this is the location where the request packet is dequeued and marked as done.

24. Execute until you reach **DriveExecute** again by typing

    g DriveExecute

    Note that the current state is 8 (**Idle**). This state programs the timer to turn off the drive motors after a certain period.

25. Clear all breakpoints by typing

    bc *

    The **read** request is now complete.

26. Type the following command to set another breakpoint at **DriveRequest** if you want to continue to watch each device driver command issued by MS OS/2 during the boot process:

    bp DriveRequest

27. Disable the breakpoint at **DriveRequest** when you are finished watching the device driver commands, and allow MS OS/2 to complete the boot process.

Tracing an INT 13H Request

1. After booting, you will be running the session manager.
2. Select the option to start *command.com.* Note that you must be in real mode to trace an INT 13H request.
3. Enter the time and date if prompted.
4. Press CONTROL-C on the debug terminal after you see the *A*> prompt on the main console.
5. Find the real-mode segment of the device driver by typing

   dg *selector*

   for (display GDT), where *selector* is the protected-mode code selector value for the disk driver. Divide the base value displayed by 10H to find the real-mode segment.
6. Using the real-mode segment, set a breakpoint at **Int13Handler** (*atint13.asm*).
7. Resume execution by typing

   g
8. On your main console, type

   disktest

   The **disktest** program uses INT 13H to read a sector from drive A. Execution will break at **Int13Handler**.

   The registers are set up for an INT 13H request. Note the transfer address at ES:BX.

9. Refer to the source listing and use the DOS-resident debugger **p** command to trace through a few instructions. This code obtains the real-mode data segment in register DS, obtains the BDS for the drive, and checks to see if it is removable.

   When you reach label **I13h0_1**, the code calls the **DevHlp** function **EnterROMCritSection** to prevent the 3.*x* box from being suspended during ROM I/O.

10. Continue tracing the code until you reach the call to **SWait**. This call waits on the INT 13H semaphore to ensure that only one INT 13H request is active at a time.

11. Trace through **SWait** using the **p** command. This process will set the semaphore, blocking any other processes wanting to execute an INT 13H.

    Since the ROM and device driver use very different data structures, the INT 13 handler must wait until the device driver is not using the disk system, then set a semaphore to ensure that the device driver will not attempt a disk access during an INT 13H. The two calls to **SWait** lock out further floppy and hard disk access.

12. Trace over both of the calls to **SWait**.

13. Use the **p** command to trace over the call that calls the ROM INT 13H code at the label **I13h2**.

14. Use the **D** command to dump the transfer buffer pointed to by ES:BX. You should see the boot sector from drive A.

    After several register pushes, you should get to the label **I13h4**. This code updates several ROM and BDS variables.

15. Locate the label **I13h7** by disassembling the code and comparing it with the source listing.

16. Resume execution until you reach the label **I13h7** by typing

    g *address*

    where *address* is the address of **I13h7**.

Note that **SSig** is called for each semaphore set to indicate that you are done with the disk system.

17. Use the **p** command to trace through the **SSig** calls.

The **DevHlp** function **LeaveROMCritSection** will inform MS OS/2 that you are done with the ROM I/O and the 3.*x* box can be swapped out.

The INT 13H request is now complete.

Appendix A DOS-resident Debugger Error Messages

This appendix lists the DOS-resident error messages, and explains why you might receive them.

`Bad flag error`

An invalid flag name was entered as part of a **r** (**display register**) command. For a list of valid register flags, refer to Chapter 3, "Using the DOS-resident Debugger."

`Bad handle`

An invalid handle was used in the **.h** (**display memory manager handle**) command. Use the **a** option to display all of the handles and to obtain their valid names and addresses.

`Bad register error`

An invalid register name was entered as part of the **r** (**display register**) command. For a list of valid register names, refer to Chapter 3, "Using the DOS-resident Debugger."

`Invalid selector`

An invalid selector was entered as part of a DOS-resident debugger command. Use the **dg** (**display GDT**) or the **dl** (**display LDT**) command to check the GDT or LDT for the correct descriptor value, or use a physical address to access memory.

`Invalid structure name`

An invalid structure name was used in the **.d** (**display data structures**) command. For a list of valid data structures, refer to Chapter 3, "Using the DOS-resident Debugger."

`No active maps`

The DOS-resident debugger displays this message when executing the **lm (list map)** command without *.sym* symbol files being loaded.

`No group with that name`

An invalid group was specified in the **ls** (**list symbols**) command. Use the **lg** (**list groups**) command to display a list of the groups in the active symbol file.

`No matching symbol`

An invalid symbol was specified as part of a DOS-resident debugger command. Use the **ls (list symbols)** command to display a list of valid symbols for the active symbol file.

`No symbol map with that name`

An invalid symbol file was specified with the **w (change map)** command. Use the **lm (list map)** command to display a list of loaded symbol files.

`Trap 0 - Divide Error Exception`

`Trap 1 - Unexpected Trace Interrupt`

`Trap 2 - NMI Interrupt`

`Trap 4 - INTO Detected Overflow Exception`

`Trap 5 - BOUND Range Exceeded Exception`

`Trap 6 - Invalid Opcode Exception`

`Trap 7 - Processor Extension Not Available`

`Trap 8 - Double Exception Detected`

`Trap 9 - Processor Extension Segment Overrun`

`Trap 10 (0AH) - Invalid TSS`

`Trap 11 (0BH) - Segment Not Present`

`Trap 12 (0CH) - Stack Segment Overun or Not Present`

`Trap 13 (0DH) - General Protection Fault`

These messages appear when the debugger detects one of the 80286 microprocessor traps. The traps are not a normal occurrence, and are most likely caused by a software error while running in protected mode. For an explanation of the traps, refer to Appendix B, "References."

Appendix B
References

B.1 Applicable Documents

The following Intel publications contain helpful information about the architecture of the 80286 microprocessor:

- *80286 Programmer's Reference Manual.* Order number: 210498. Intel Corporation, Santa Clara, Ca.
- *80286 Hardware Reference Manual.* Order number: 210760. Intel Corporation, Santa Clara, Ca.
- *80286 Operating Systems Writer's Guide.* Order number: 121960. Intel Corporation, Santa Clara, Ca.

The following Intel publications contain helpful information about the architecture of the 80386 microprocessor:

- *Introduction to the 80386.* Order number: 231252. Intel Corporation, Santa Clara, Ca.
- *80386 Programmer's Reference Manual.* Order number: 230985. Intel Corporation, Santa Clara, Ca.
- *80386 Hardware Reference Manual.* Order number: 221732. Intel Corporation, Santa Clara, Ca.
- *80386 Operating Systems Writer's Guide.* Order number: 231499. Intel Corporation, Santa Clara, Ca.
- *80386 High-Performance 32-bit Microprocessor with Integrated Memory Management (Data Sheet).* Order number: 231630. Intel Corporation, Santa Clara, Ca.

The following IBM publications contain helpful information about the operation and use of the IBM PC AT computer:

- *IBM PC AT Operations Guide.* International Business Machines Corporation, Boca Raton, Fl., 1985.
- *IBM PC AT Technical Reference.* International Business Machines Corporation, Boca Raton, Fl., 1985.
- *IBM PC XT Technical Reference.* International Business Machines Corporation, Boca Raton, Fl., 1985.

B.2 Microsoft Documents

The following Microsoft publications expand on the information presented in this Device Driver Guide, and will enable you to understand and complete the process of adapting MS OS/2 for your target system:

- *Microsoft Operating System/2 Setup Guide*
- *Microsoft Operating System/2 Beginning User's Guide*
- *Microsoft Operating System/2 User's Reference*
- *Microsoft Operating System/2 Programmer's Reference*
- *Microsoft Operating System/2 Programmer's Guide*
- *Microsoft Operating System/2 Device Drivers Guide*
- *Microsoft Operating System/2 LAN Manager 1.0 User's Reference*
- *Microsoft Operating System/2 LAN Manager 1.0 Administrator's Guide*

Appendix C
Overview of Automated Tests

The flowcharts on the following pages illustrate the testing strategy for the device driver tests described in Chapter 4, "Automated Tests."

Figure C.1 Automated Device Driver Header Tests

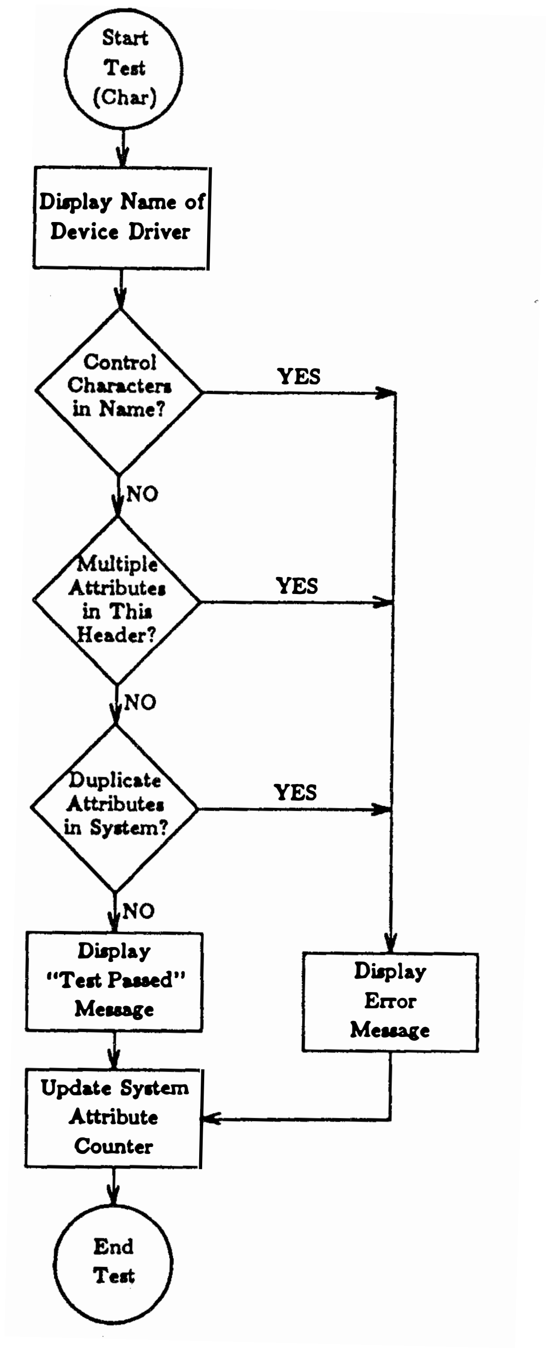

Start Test (Char)
Display Name of Device Driver
Control Characters in Name?
YES
NO
Multiple Attributes in This Header?
YES
NO
Duplicate Attributes in System?
YES
NO
Display "Test Passed" Message
Display Error Message
Update System Attribute Counter
End Test

Figure C.1 Automated Device Driver Header Tests *continued*

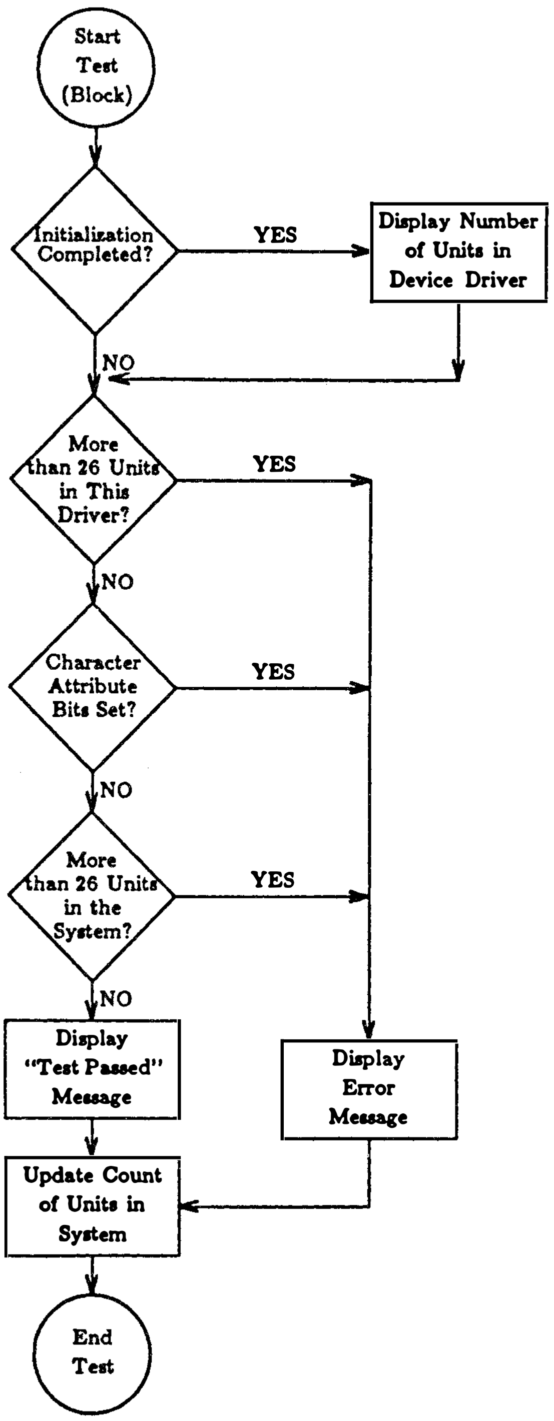

Start
Test
(Block)
Initialization Completed?
YES
Display Number of Units in Device Driver
NO
More than 26 Units in This Driver?
YES
NO
Character Attribute Bits Set?
YES
NO
More than 26 Units in the System?
YES
NO
Display "Test Passed" Message
Display Error Message
Update Count of Units in System
End
Test

Figure C.2 Character Device Driver: Overview of Tests

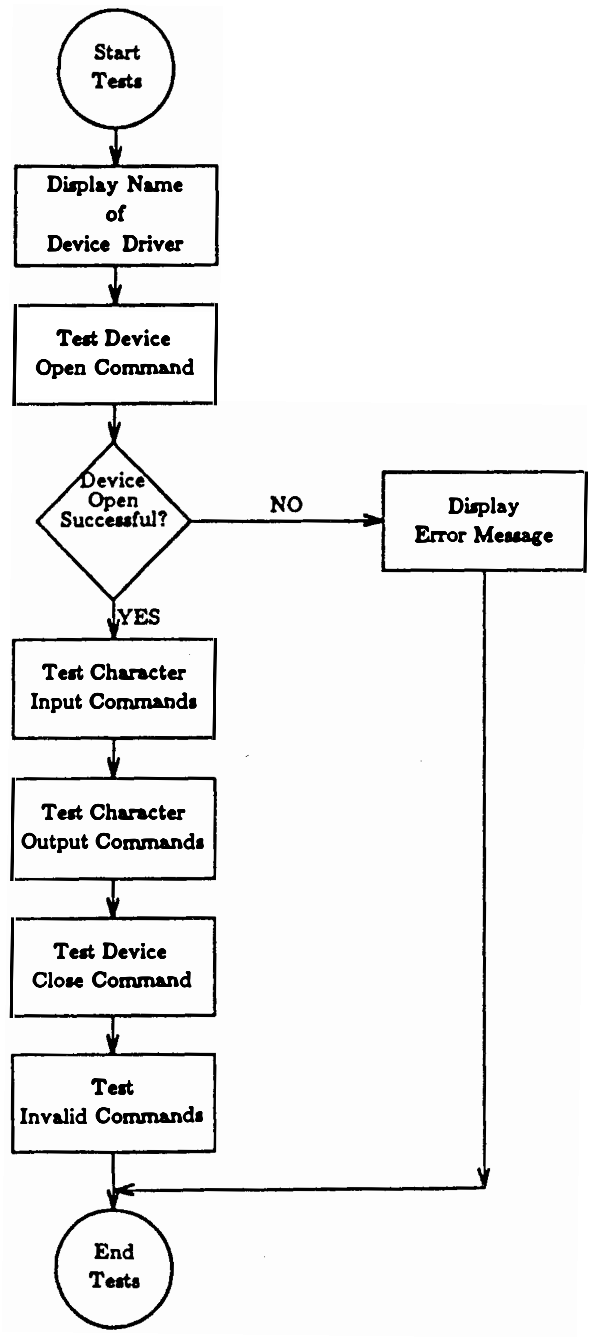


Figure C.3 Character Device Driver: Test Open and Close Commands

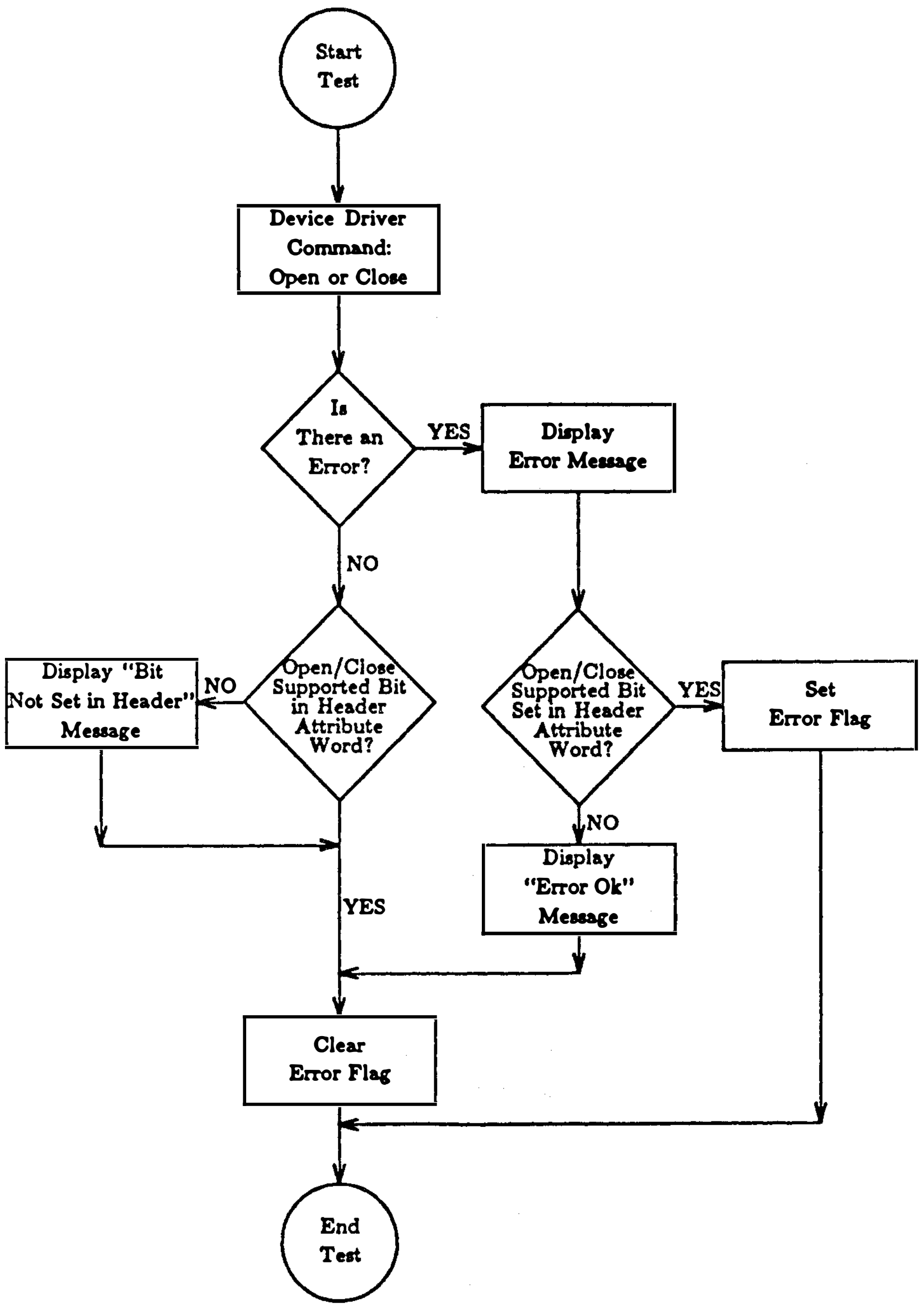

Start Test
Device Driver Command: Open or Close
Is There an Error?
YES
Display Error Message
NO
Open/Close Supported Bit in Header Attribute Word?
NO
Display "Bit Not Set in Header" Message
YES
Open/Close Supported Bit Set in Header Attribute Word?
YES
Set Error Flag
NO
Display "Error Ok" Message
Clear Error Flag
End Test

Figure C.4 Character Device Driver: Test Character Input Commands

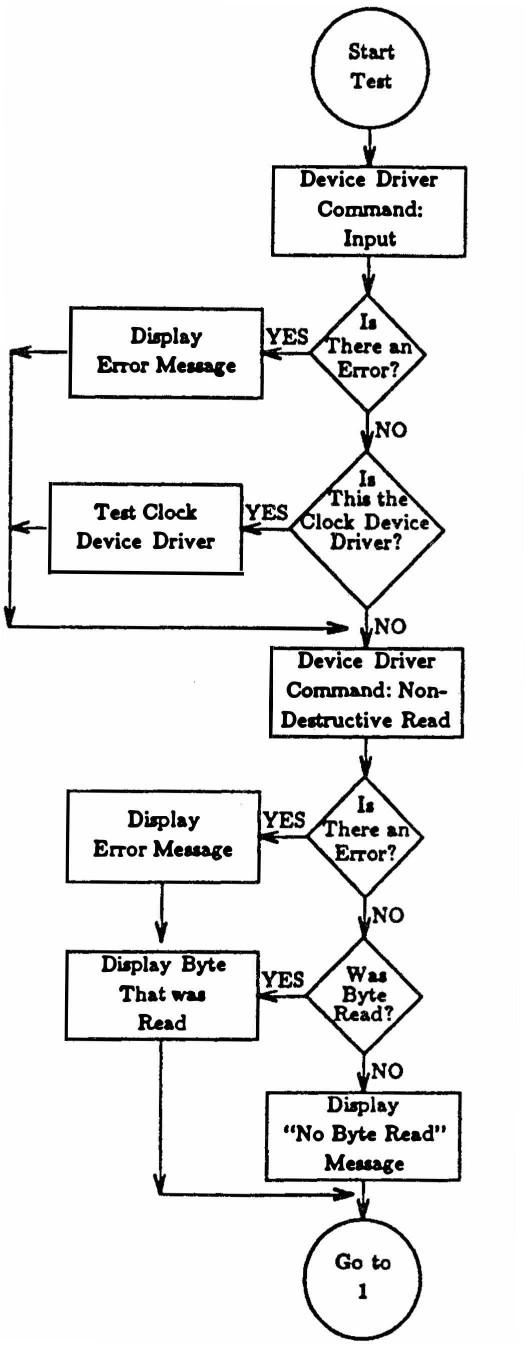

Start Test
Device Driver Command: Input
Is There an Error?
YES
Display Error Message
NO
Is This the Clock Device Driver?
YES
Test Clock Device Driver
NO
Device Driver Command: Non-Destructive Read
Is There an Error?
YES
Display Error Message
NO
Was Byte Read?
YES
Display Byte That was Read
NO
Display "No Byte Read" Message
Go to 1

Figure C.4 Character Device Driver: Test Character Input Commands *continued*

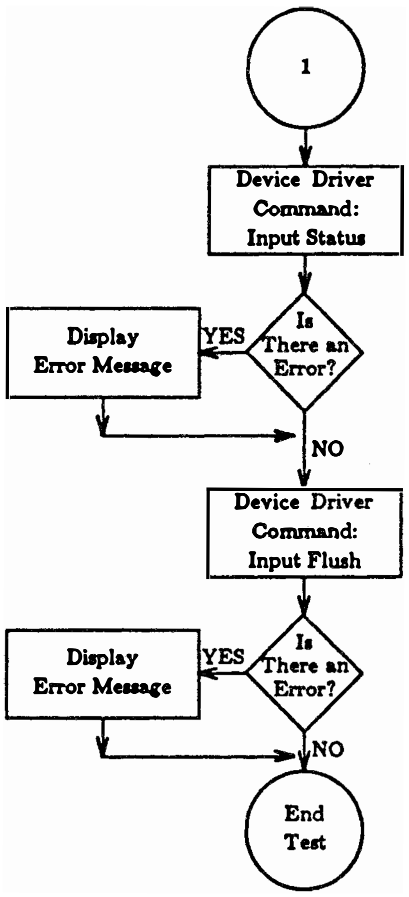


Figure C.5 Character Device Driver: Test Clock Device Driver

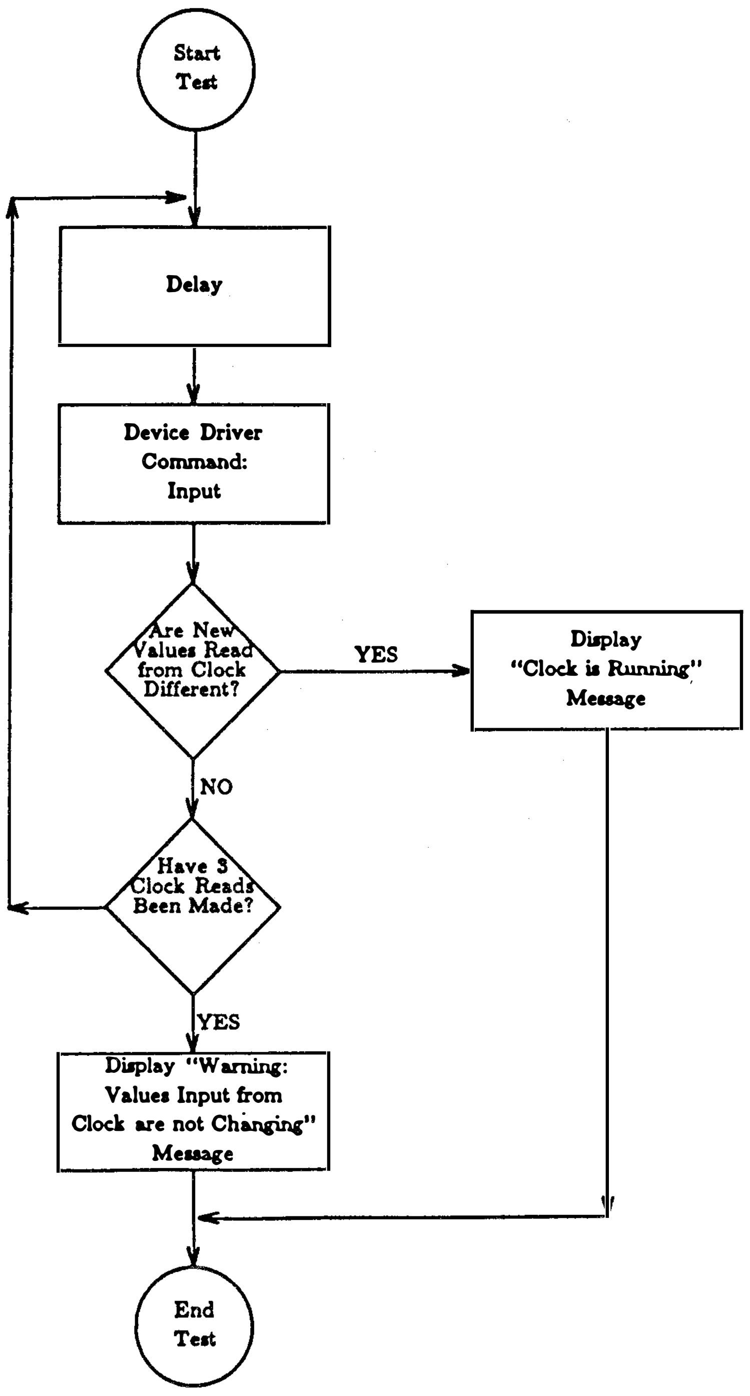

Start Test
Delay
Device Driver Command: Input
Are New Values Read from Clock Different?
YES
Display "Clock is Running" Message
NO
Have 3 Clock Reads Been Made?
YES
Display "Warning: Values Input from Clock are not Changing" Message
End Test

Figure C.6 Character Device Driver: Test Character Output Commands

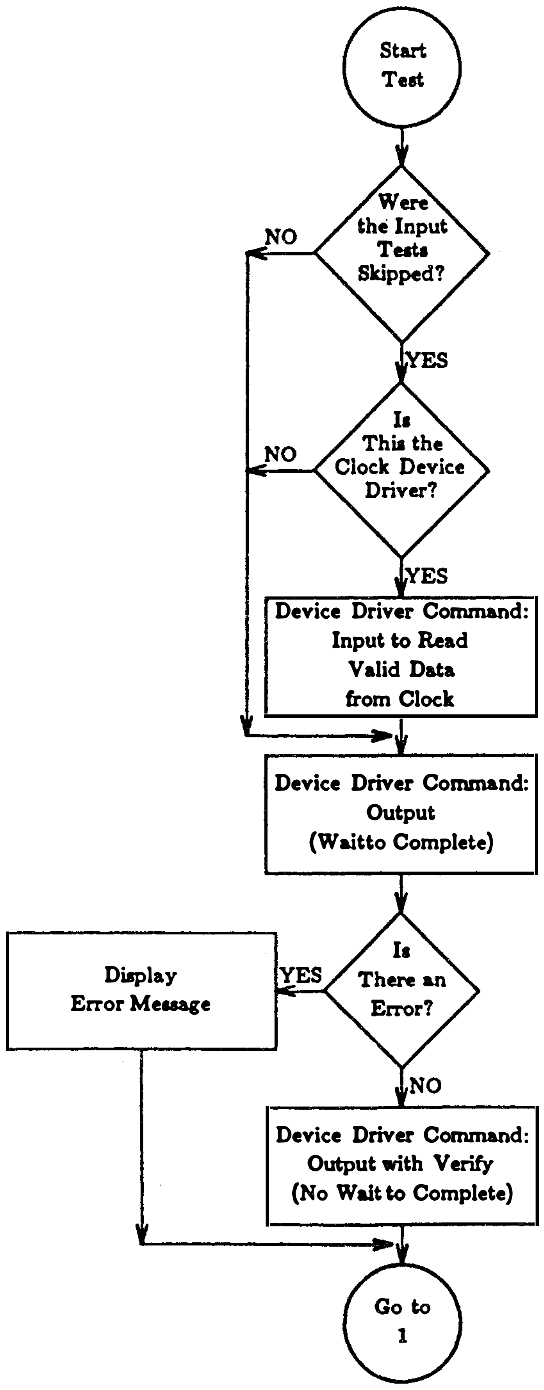

Start Test
Were the Input Tests Skipped?
NO
YES
Is This the Clock Device Driver?
NO
YES
Device Driver Command: Input to Read Valid Data from Clock
Device Driver Command: Output (Waitto Complete)
Is There an Error?
YES
Display Error Message
NO
Device Driver Command: Output with Verify (No Wait to Complete)
Go to 1

Figure C.6 Character Device Driver: Test Character Output Commands *continued*

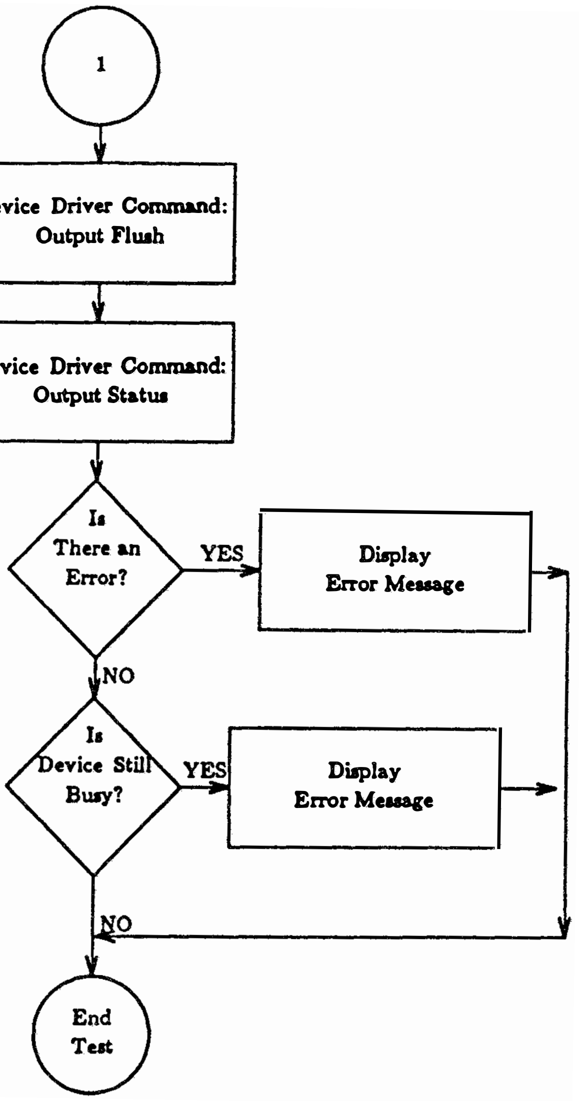


Figure C.7 Block Device Driver: Overview of Test

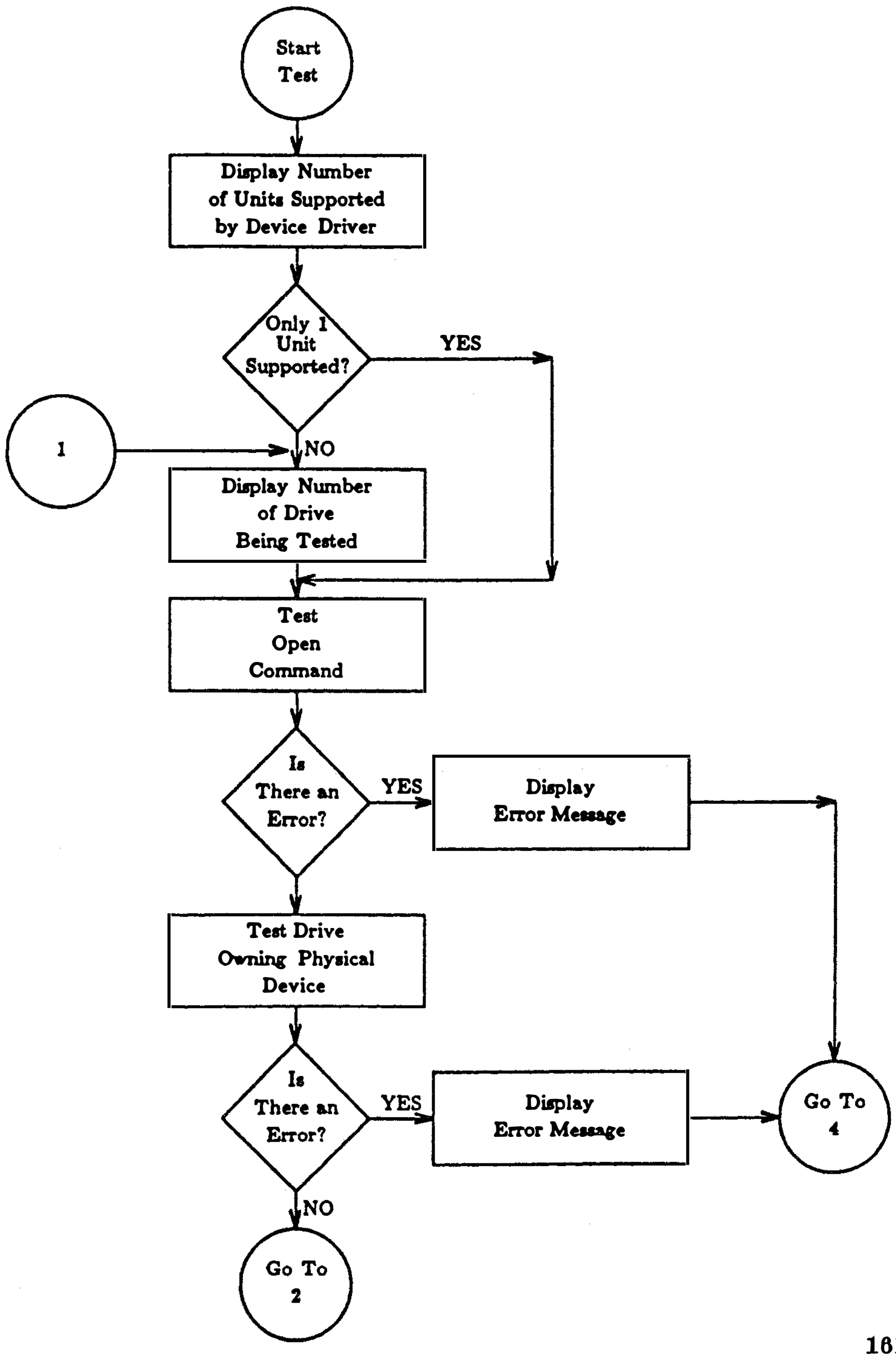


Figure C.7 Block Device Driver: Overview of Tests *continued*

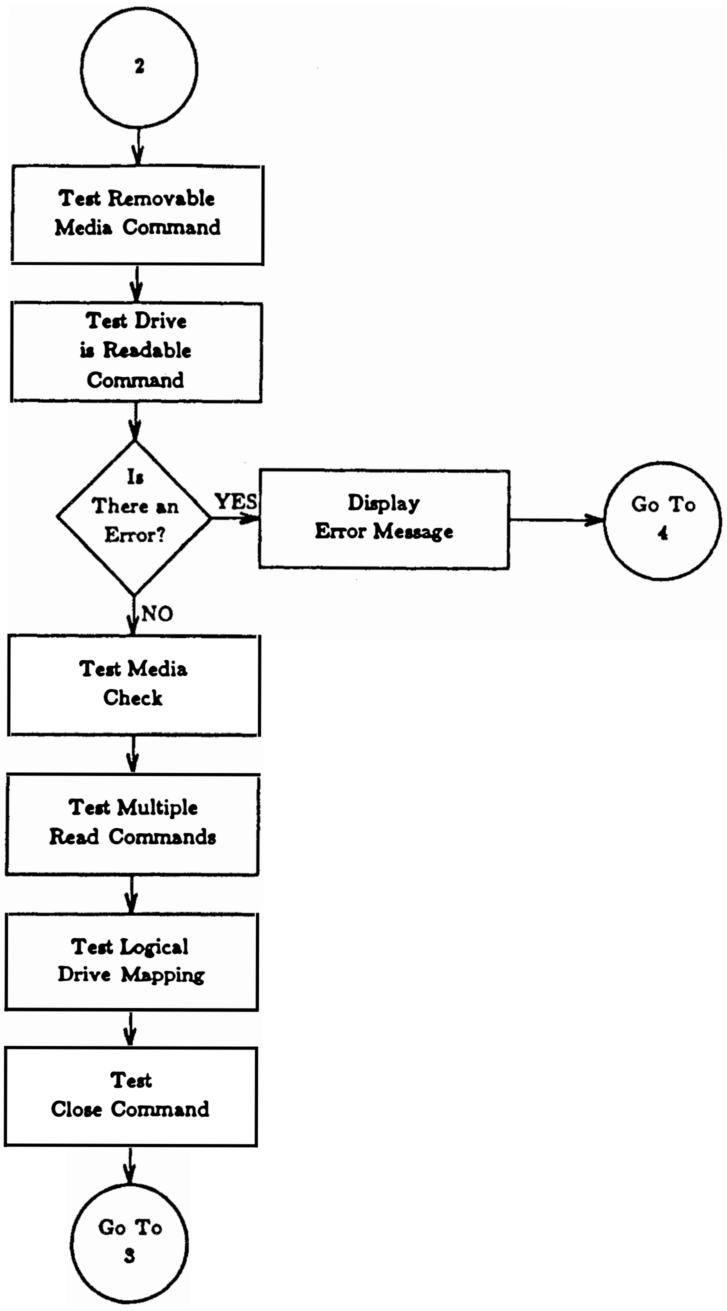

2
Test Removable Media Command
Test Drive is Readable Command
Is There an Error?
YES
Display Error Message
Go To 4
NO
Test Media Check
Test Multiple Read Commands
Test Logical Drive Mapping
Test Close Command
Go To 3

Figure C.7 Block Device Driver: Overview of Tests *continued*

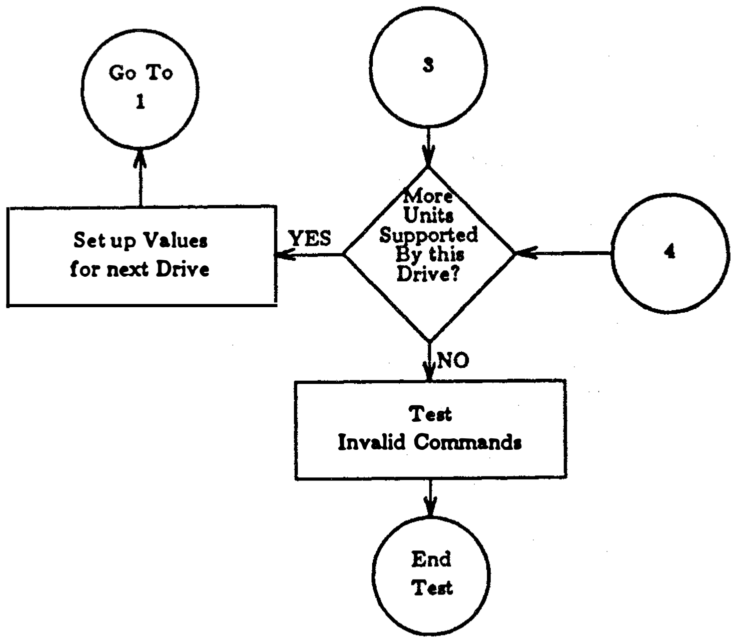

Go To 1
3
More Units Supported By this Drive?
YES
Set up Values for next Drive
4
NO
Test Invalid Commands
End Test

Figure C.8 Block Device Driver: Test Open and Close Commands

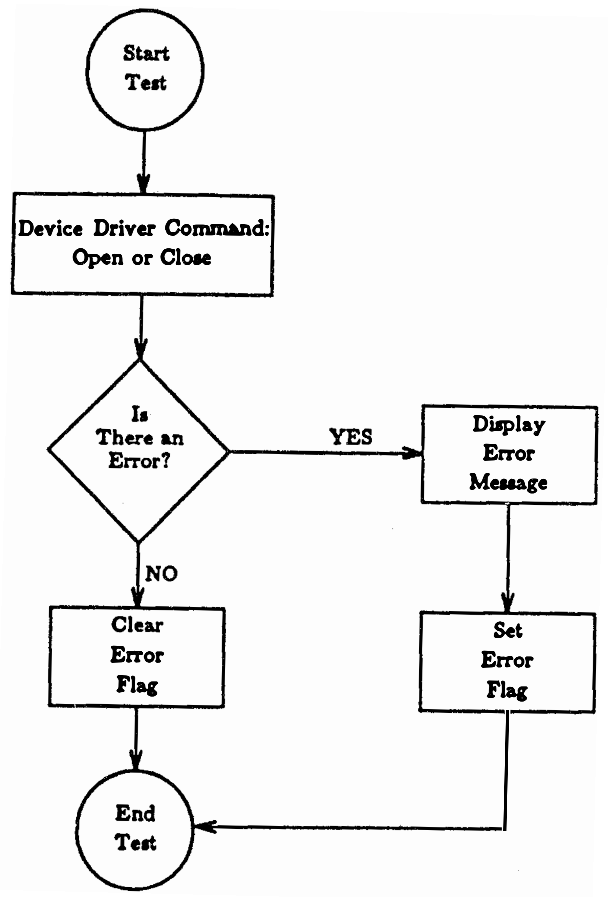


Figure C.9 Test for Drive Owning Physical Unit

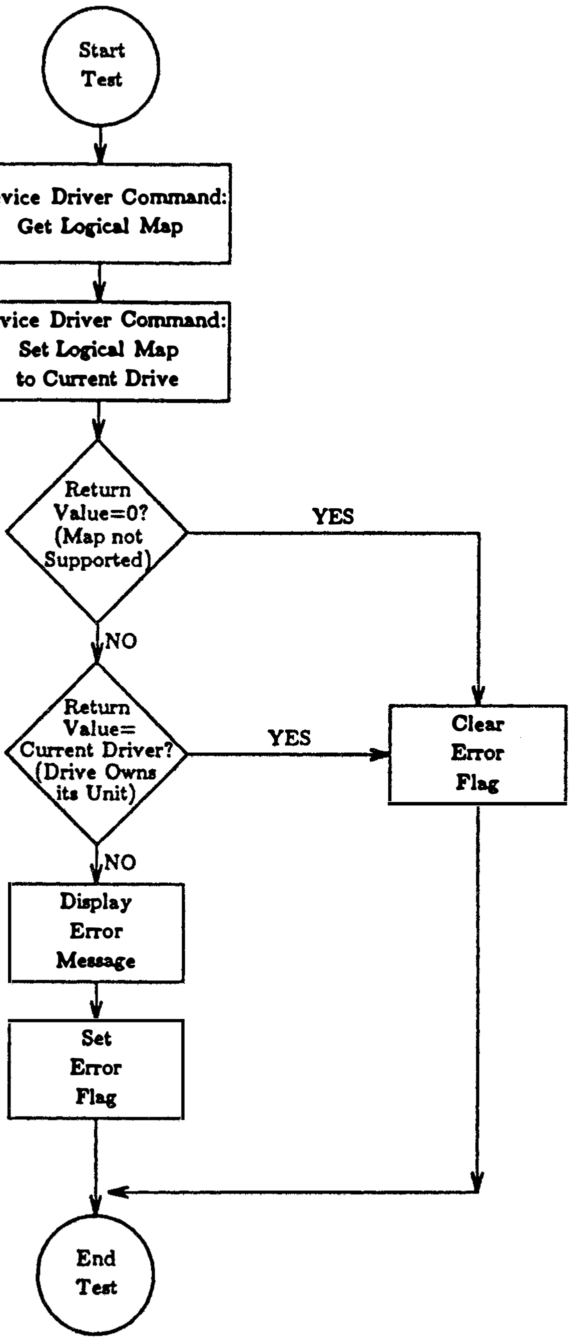

Start Test
Device Driver Command: Get Logical Map
Device Driver Command: Set Logical Map to Current Drive
Return Value=0? (Map not Supported)
YES
NO
Return Value= Current Driver? (Drive Owns its Unit)
YES
NO
Clear Error Flag
Display Error Message
Set Error Flag
End Test

Figure C.10 Test Removable Media Command

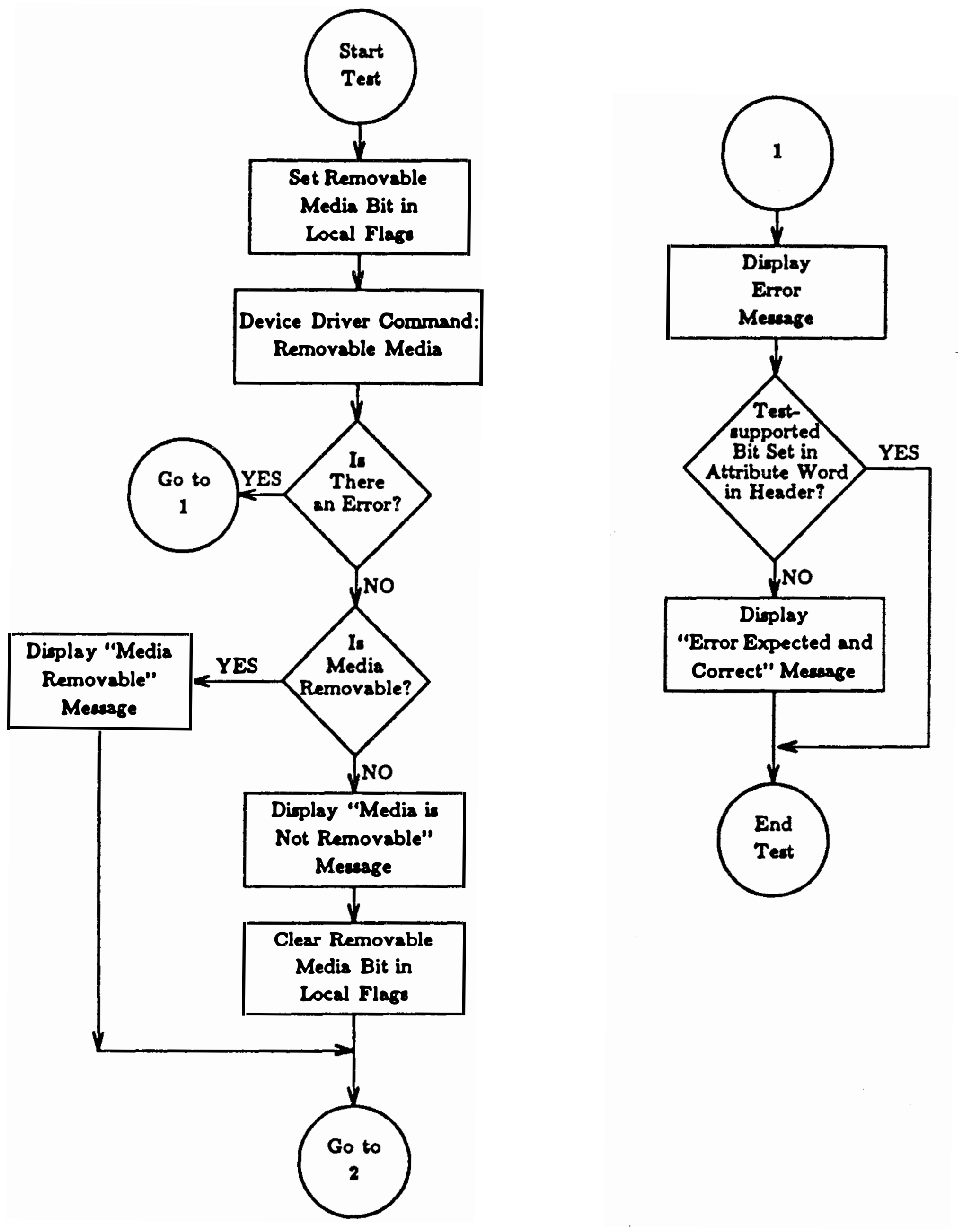

Start Test
Set Removable Media Bit in Local Flags
Device Driver Command: Removable Media
Is There an Error?
YES
Go to 1
NO
Is Media Removable?
YES
Display "Media Removable" Message
NO
Display "Media is Not Removable" Message
Clear Removable Media Bit in Local Flags
Go to 2
1
Display Error Message
Test-supported Bit Set in Attribute Word in Header?
YES
NO
Display "Error Expected and Correct" Message
End Test

Figure C.10 Test Removable Media Command *continued*

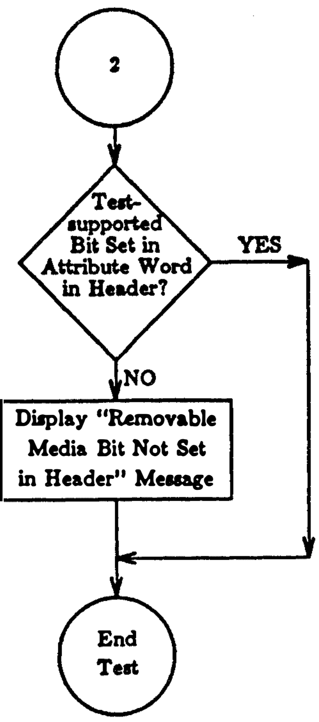

2
Test-supported Bit Set in Attribute Word in Header?
YES
NO
Display "Removable Media Bit Not Set in Header" Message
End Test

Figure C.11 Test Drive is Readable

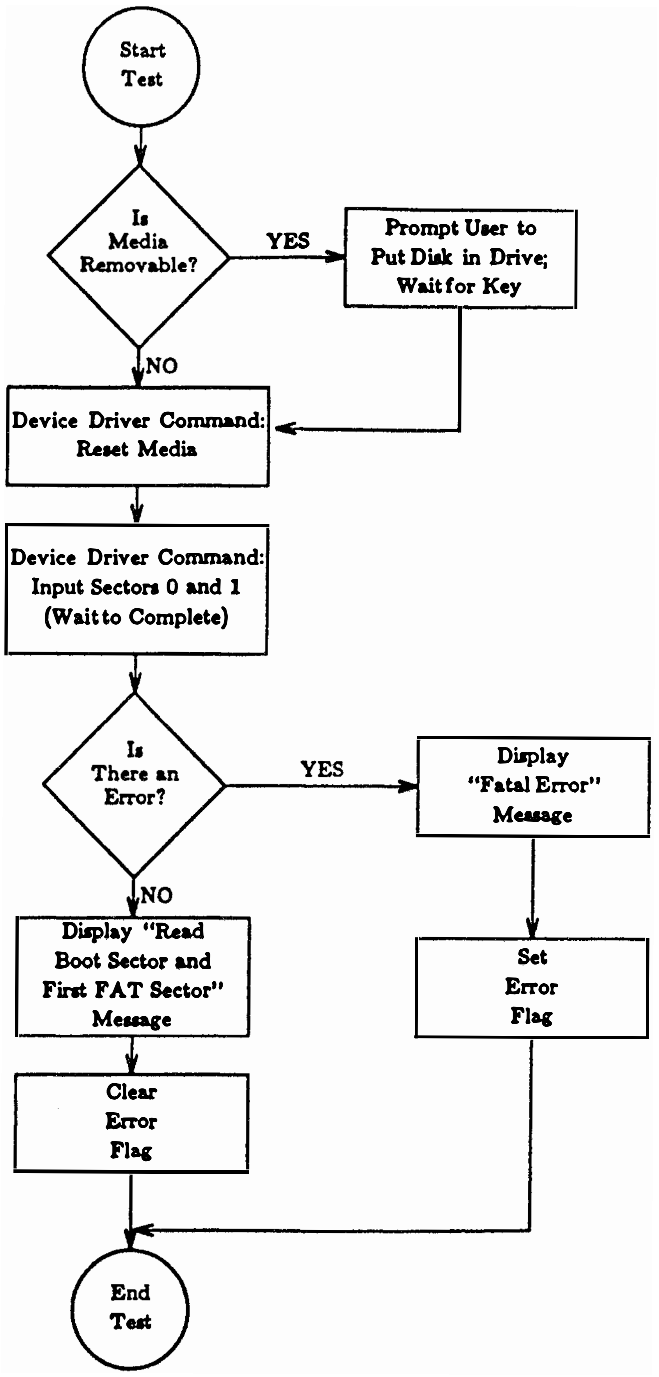

Start Test
Is Media Removable?
YES
Prompt User to Put Disk in Drive; Wait for Key
NO
Device Driver Command: Reset Media
Device Driver Command: Input Sectors 0 and 1 (Wait to Complete)
Is There an Error?
YES
Display "Fatal Error" Message
NO
Display "Read Boot Sector and First FAT Sector" Message
Set Error Flag
Clear Error Flag
End Test

Figure C.12 Block Device Driver: Test Media Check

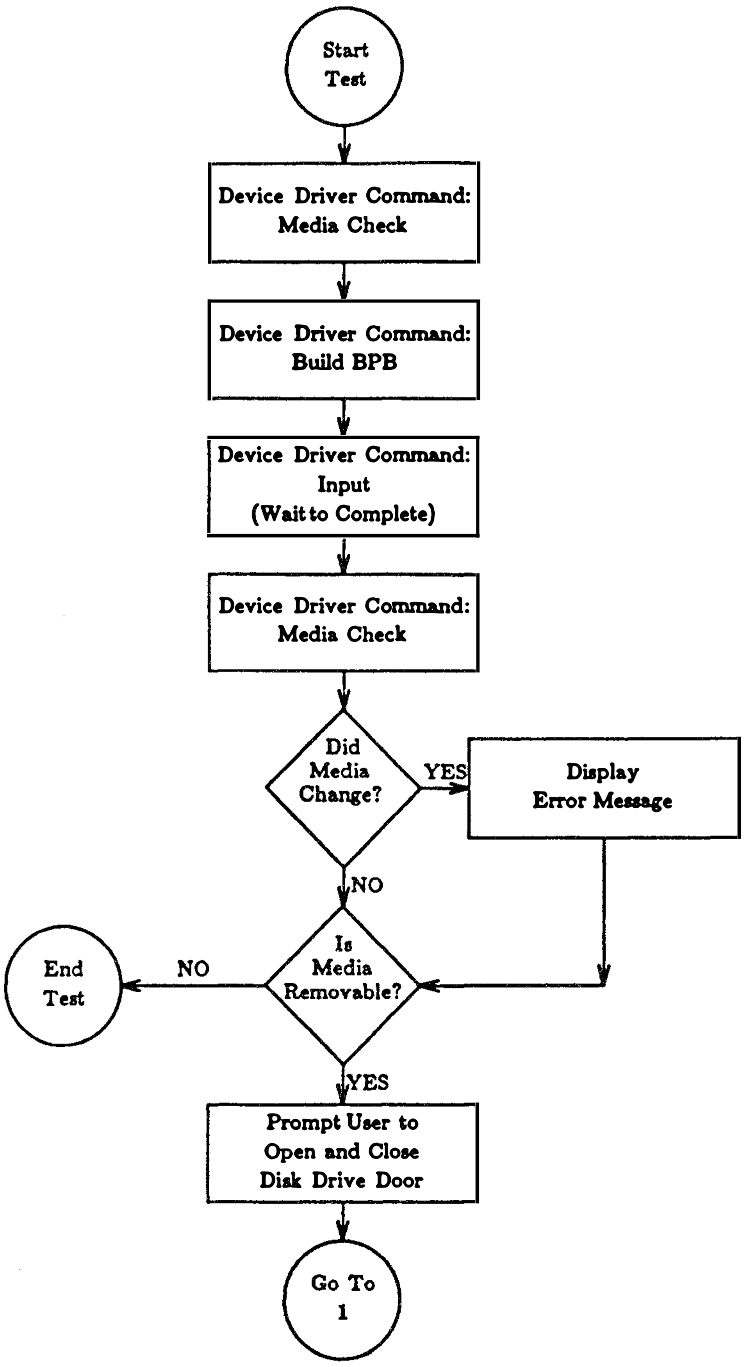

Start Test
Device Driver Command: Media Check
Device Driver Command: Build BPB
Device Driver Command: Input (Wait to Complete)
Device Driver Command: Media Check
Did Media Change?
YES
Display Error Message
NO
Is Media Removable?
NO
End Test
YES
Prompt User to Open and Close Disk Drive Door
Go To 1

Figure C.12 Block Device Driver: Test Media Check *continued*

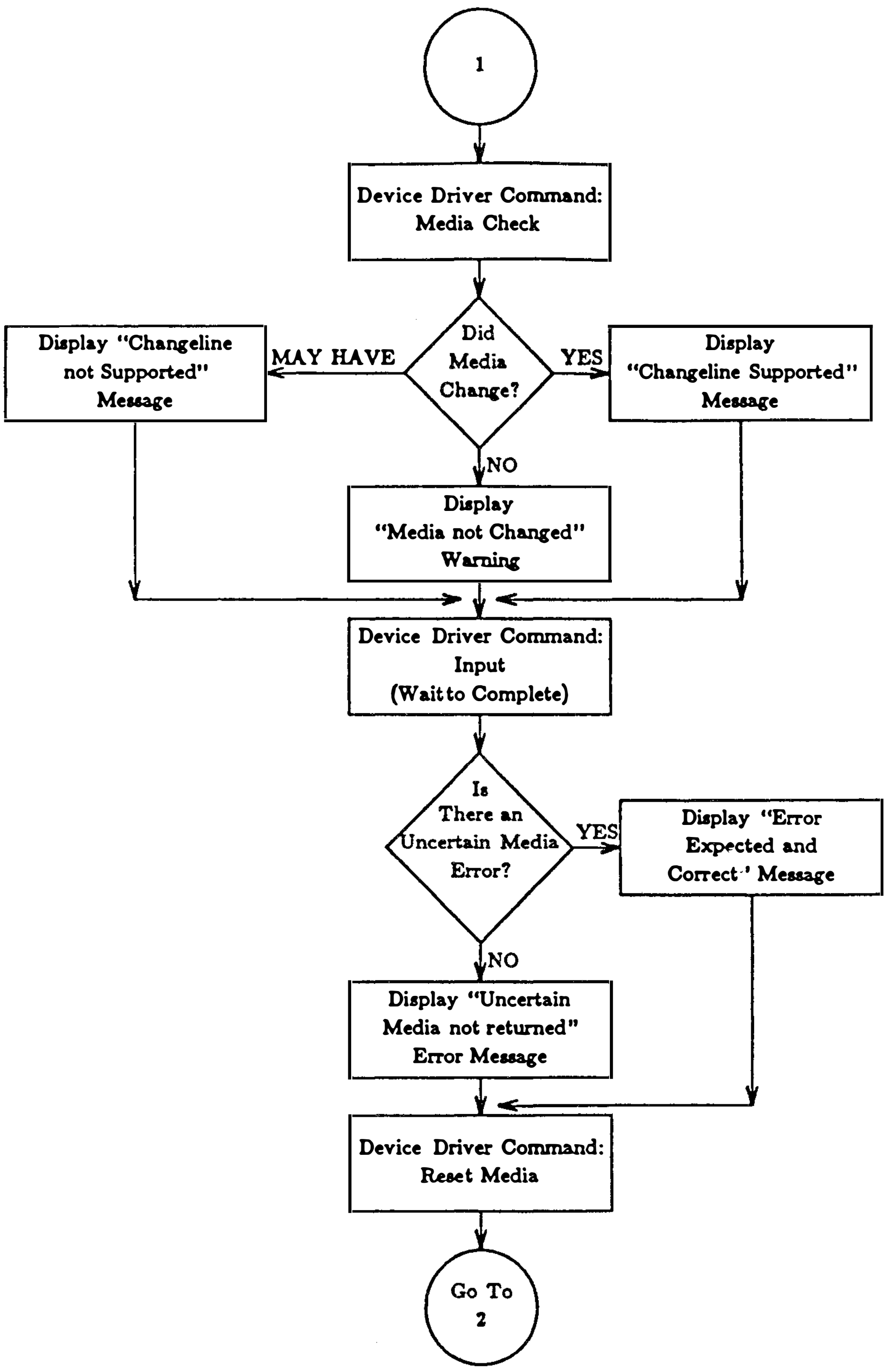

1
Device Driver Command: Media Check
Did Media Change?
MAY HAVE
Display "Changeline not Supported" Message
YES
Display "Changeline Supported" Message
NO
Display "Media not Changed" Warning
Device Driver Command: Input (Wait to Complete)
Is There an Uncertain Media Error?
YES
Display "Error Expected and Correct" Message
NO
Display "Uncertain Media not returned" Error Message
Device Driver Command: Reset Media
Go To 2

Figure C.12 Block device Driver: Test Media Check *continued*

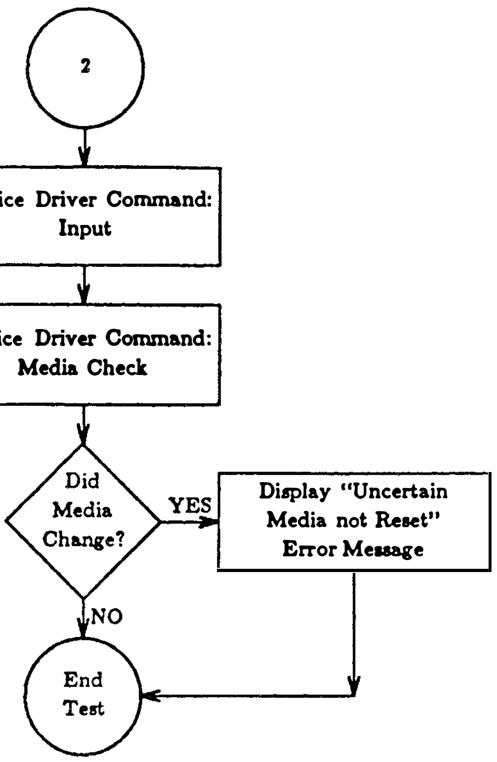

2
Device Driver Command: Input
Device Driver Command: Media Check
Did Media Change?
YES
Display "Uncertain Media not Reset" Error Message
NO
End Test

Figure C.13 Test Multiple Read Commands

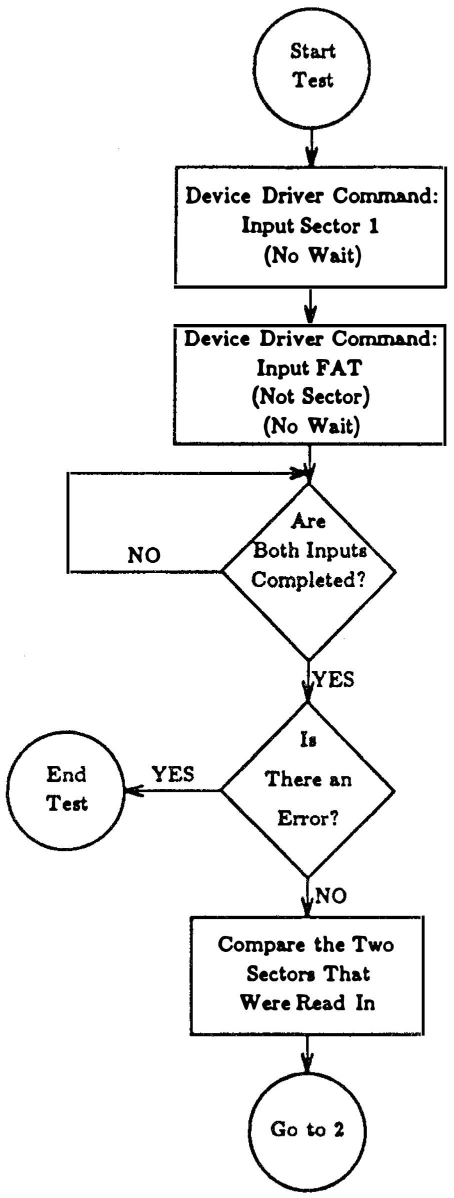

Start Test
Device Driver Command: Input Sector 1 (No Wait)
Device Driver Command: Input FAT (Not Sector) (No Wait)
Are Both Inputs Completed?
NO
YES
Is There an Error?
YES
End Test
NO
Compare the Two Sectors That Were Read In
Go to 2

Figure C.13 Test Multiple Read Commands *continued*

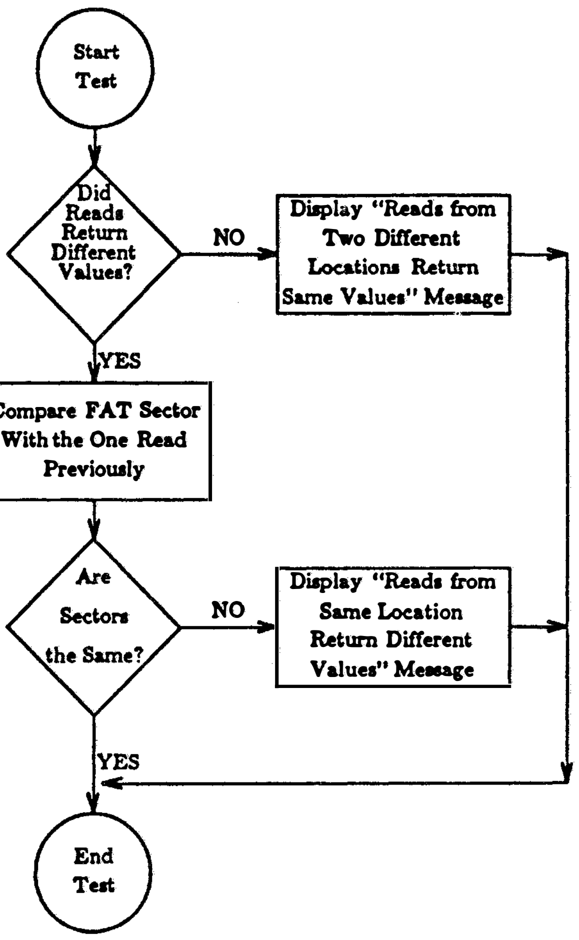

Start Test
Did Reads Return Different Values?
NO
Display "Reads from Two Different Locations Return Same Values" Message
YES
Compare FAT Sector With the One Read Previously
Are Sectors the Same?
NO
Display "Reads from Same Location Return Different Values" Message
YES
End Test

Figure C.14 Block Device Driver: Test Logical Drive Mapping

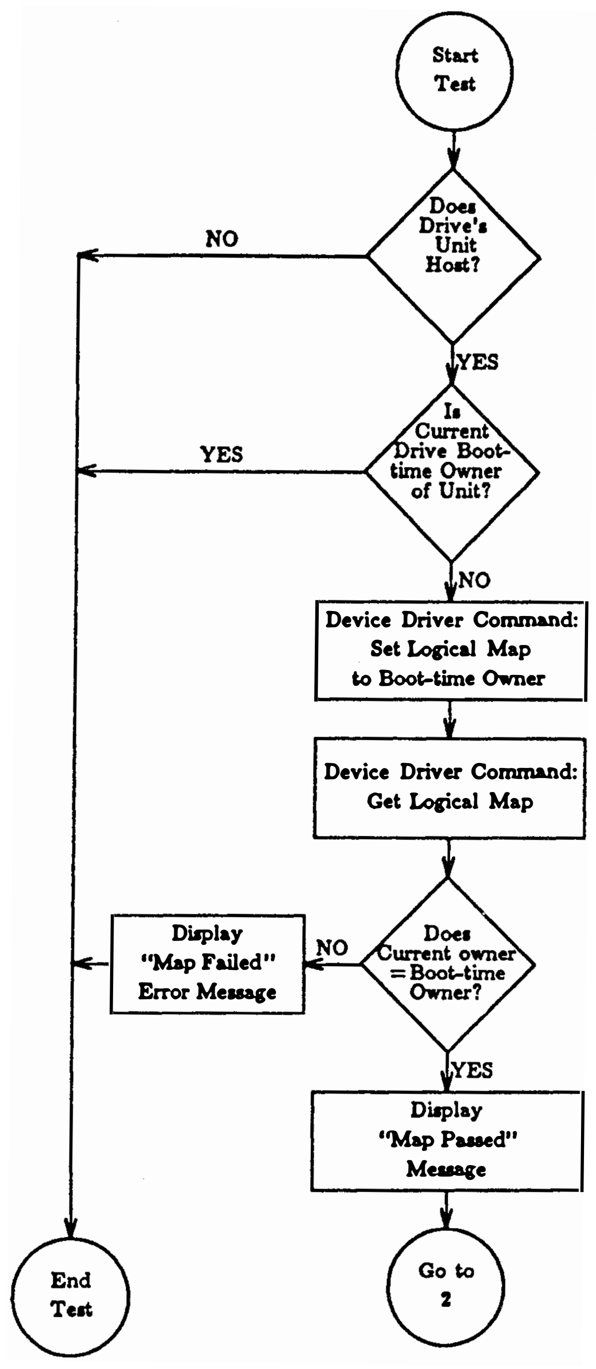
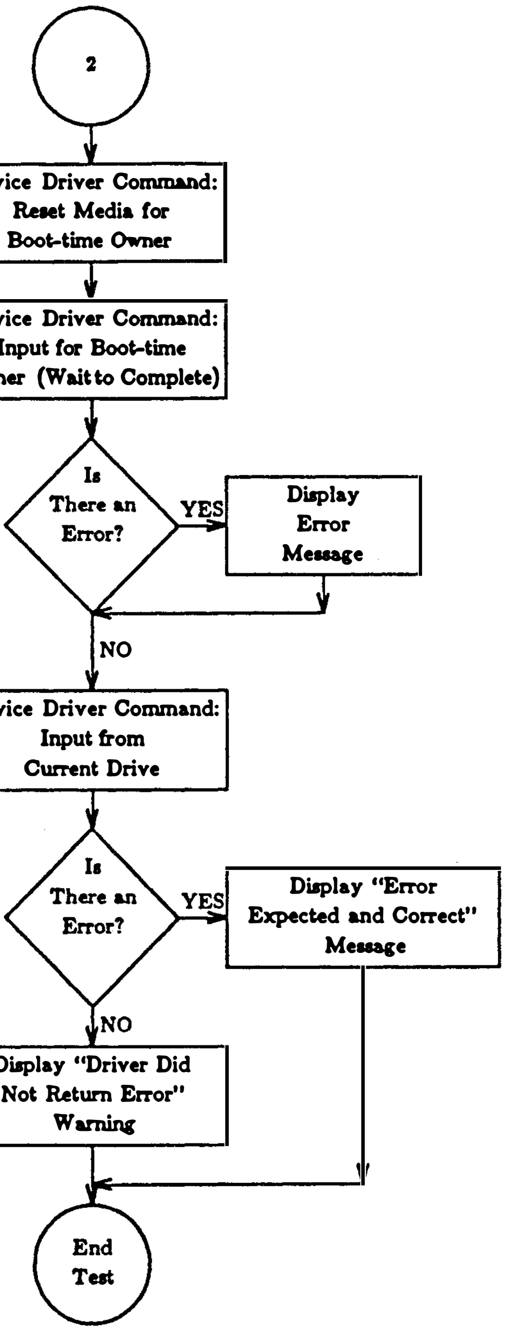

Start Test
Does Drive's Unit Host?
NO
YES
Is Current Drive Boot-time Owner of Unit?
YES
NO
Device Driver Command: Set Logical Map to Boot-time Owner
Device Driver Command: Get Logical Map
Does Current owner = Boot-time Owner?
NO
Display "Map Failed" Error Message
YES
Display "Map Passed" Message
End Test
Go to 2

Figure C.14 Block Device Driver: Test Logical Drive Mapping *continued*


2
Device Driver Command: Reset Media for Boot-time Owner
Device Driver Command: Input for Boot-time Owner (Wait to Complete)
Is There an Error?
YES
Display Error Message
NO
Device Driver Command: Input from Current Drive
Is There an Error?
YES
Display "Error Expected and Correct" Message
NO
Display "Driver Did Not Return Error" Warning
End Test

Glossary

3.*x* box

An MS OS/2 feature that allows you to run applications written for MS-DOS versions 3.2 and earlier in real mode.

base port address

The first of several consecutive port locations in the I/O space. These ports can be occupied by multiple devices.

base video address

The basic memory address of the video memory. For example, the IBM monochrome video is at B0000 and the color graphics adapter is at B8000.

bimodal

A characteristic of a device driver or routine, meaning it can be used in both real and protected modes. A routine is bimodal if it can execute in both real and protected modes. An address is bimodal if the single address can be used in both real and protected modes to refer to the same memory.

The only candidates for a bimodal segment/selector are values with the low three bits equal to zero (for example, 8H, 10H, 18H,...). Note that 0 is used as an invalid selector by the 80286 hardware. Segments/selectors of this type refer to the Global Descriptor Table (GDT). The GDT entry must have a base address of the segment/selector * 16 to ensure bimodality in real mode. *See also* dual mode.

BIOS

Basic Input/Output System.

block device

A device that performs random I/O in structured pieces called blocks (such as a disk device).

BVS

The Base Video Subsystem. This portion of the video subsystem interfaces directly with the video hardware.

character device

A device that performs serial character I/O (such as a communications port).

development system

The hardware upon which the work is performed. This guide assumes that you are using an IBM PC AT or compatible computer as your development system.

DevHlp functions

An interface provided by MS OS/2 to allow the device drivers to perform system-related services.

device driver

A software interface between the operating system and the hardware devices. When a device I/O request is made by a program, the operating system communicates with the device driver to fulfill the request.

DosHlp functions

An interface provided by the BIOS that allows MS OS/2 to perform machine-dependent services. These functions are available only for use by the operating system.

dual mode

A characteristic of a device or routine meaning it can be used in both real and protected modes. A routine is dual mode if it can execute in both real and protected modes. An address is dual mode if the single address can be used in both real and protected modes to refer to the same memory. *See also* bimodal.

dynamic-link library

A facility that improves storage usage by providing linkage to library routines at run time.

dynamic linking

Linking that occurs at program load time or during execution of a command. *See also* run-time dynamic linking and load-time dynamic linking.

Family API (Application Program Interface)

Functions supported in both the MS-DOS 3.*x* environment and the MS OS/2 environment. Programs written to the Family API interface can run unchanged under both MS-DOS 3.*x* and MS OS/2.

handle

A 16-bit value used to reference a specific device or file.

high memory

Memory above the one-megabyte boundary.

IOPL

I/O Privilege Level. An 80286 hardware protection feature that only allows programs of a certain privilege level to perform I/O. In MS OS/2, I/O privilege is granted at privilege level 2.

load time

The period of time when the MS OS/2 program loader is reading and loading an executable program into memory.

load-time dynamic linking

Dynamic linking that occurs while the MS OS/2 program loader is reading and loading an executable program into memory.

low memory

Memory below the one-megabyte boundary.

mode switching

The process of changing from real mode to protected mode or from protected mode to real mode.

OEM

Original Equipment Manufacturer.

privilege level

An 80286 hardware protection feature that allows different memory access (privilege) to different software on an 80286. In MS OS/2, the DOS and device drivers run at privilege level 0, or kernel level, which is the most privileged level. Applications run at privilege level 3, which is also called application level. Applications can also run at privilege level 2, which is called IOPL level, to allow IOPL. *See also* IOPL.

protected mode

The 80286 mode supporting the MS OS/2 multitasking environment. This mode enforces hardware-level memory protection. *See also* real mode.

race conditions

A problem that may occur when two routines or programs try to access the same piece of code or data at the same time.

real mode

The mode supporting the MS-DOS 3.*x* environment. *See also* protected mode.

reentrant code

Code that can be shared between two or more programs in a multitasking environment.

run time

The period of time when a program is first allowed to execute. This period is ended when all the program's threads terminate.

run-time dynamic linking

Dynamic linking that occurs while a program is executing.

segmented memory

Memory physically divided into segments of arbitrary size, with some segments as large as 64K bytes.

session manager

An interface used to switch between applications running under MS OS/2.

state machine

A section of code composed of a finite set of discrete states or functions. Each state is composed of certain instructions and conditions, and contains a transition to the next state.

system initialization

The process of reading the boot code, loading the system files, and initializing the hardware and operating system.

target system

The system that will run the device driver after the work is completed.

TPI

Tracks-per-inch.

VIO

Video Input/Output.

Index

@, 28
80286 microprocessor. *See* Intel 80286 microprocessor
80286
  reference material, 4
80386
  reference material, 4
ADD tests, 65
  changing the flag bytes, 66
  flag bytes, 66
  notes on running, 67
  overview of, 68
  redirecting test output, 66
  summary of, 70
  summary
    Test block device driver functions, 71
    Test block device driver header, 70
    Test character device driver functions, 70
    Test character device driver header, 70
    Test device driver functions, 70
    Test device driver headers, 70
  supplemental device drivers
    configuring, 68
  test IDT, 75
  test Interrupt Descriptor Table, 75
  test IRQInfoTable, 72
  user input, 66
ASCII characters, 36
Automated device driver tests, 65
  changing the flag bytes, 66
  flag bytes, 66
  notes on running, 67
  overview of, 68
  redirecting test output, 66
  setting up the tests, 65
  sumary of, 70
  supplemental device drivers
    configuring, 68
  user input, 66
Automated tests, 65
  description of, 151
  flowcharts, 151
  testing MS OS/2 modifications, 65

Binary and unary operators, 20
Binary distribution disks, 3
Binary operators
  list of, 21
Booting up the DOS-resident debugger, 16
Breakpoint clear command, DOS-resident debugger
  abbreviation, 32
  comments, 32
  purpose, 32
  syntax, 32
Breakpoint disable command, DOS-resident debugger
  abbreviation, 32
  comments, 32
  purpose, 32
  syntax, 32
Breakpoint enable command, DOS-resident debugger
  abbreviation, 33
  comments, 33
  purpose, 33
  syntax, 33
Breakpoint list command, DOS-resident debugger
  abbreviation, 33
  comments, 33
  purpose, 33
  syntax, 33
Breakpoint set command, DOS-resident debugger
  abbreviation, 34
  comments, 34
  purpose, 34
  syntax, 34
breakpoints
  go command, 42
Byte, 19

Change default command, DOS-resident debugger
  abbreviation, 55
  comments, 55
  purpose, 55
  syntax, 55
Change map file command, DOS-resident debugger
  abbreviation, 54
  comments, 54
  purpose, 54

Change map file command, DOS-resident debugger *(continued)*
  syntax, 54
Character queues, 87
Command parameter, DOS-resident debugger
  expression, 19
  string, 20
Command parameters, DOS-resident debugger
  addr, 19
  byte, 19
  description, 18
  dword, 19
  group-name, 19
  list, 19
  map-name, 19
  range, 20
  register-name, 20
  word, 20
Commands
  notational conventions, 6
Compare command, DOS-resident debugger
  abbreviation, 35
  comments, 35
  purpose, 35
  syntax, 35
conditional execute command, DOS-resident debugger
  abbreviation, 44
Conditional execute command, DOS-resident debugger
  comments, 44
  purpose, 44
  syntax, 44
CONTROL-C, 17
  aborting DOS-resident debugger, 17
CONTROL-Q, 18
  resume screen display, 18
CONTROL-S, 18
  stops screen display, 18
Δ 2, 17

Data types
  notational conventions, 6
Data-format specification
  B, 57
  b, 57
  c, 57
  d, 57
  s, 57
Data-format specifications
  use with DBPRT statements, 57
Data-format specification
Data-format specification *(continued)*
  x, 57
DBPRT statements
  data-format specification, 57
    $, 57
  data-format specifications, 57
    list of, 57
  examples, 58
  literal characters, 57
  literals, 57
    \$, 57
    \b, 57
    \\, 57
    \n, 57
    \t, 57
DBPRT statements,
  purpose, 56
DBPRT statements
  syntax, 56
  tips on using, 58
Debug print statements, 56
  examples, 58
  tips on using, 58
Debugger commands, 24
Debugging the system
  target system, 5
Debugging
  DBPRT statements, 56
  with INT 3 instructions, 59
  with map files, 59
Development process
  hardware requirements, 9
  knowledge required, 4
  overview, 3
  simplifying, 5
Development system
  computer, 3
  described, 9
  hardware requirements, 9
  memory requirements, 9
  optional hardware, 9
  software requirements, 10
DevHlp routines
  described, 86
DevHlp services, 83
Device, 82
Device driver headers, 85
Device Driver Kit
  contents, 3
  described, 3
  documentation, 3
Device driver
  automated tests
    overview, 6
  development approach, 5, 80
    sample serial device driver, 98

Device driver *(continued)*
  display adapter, 5
  floppy disk, 5
  hard disk, 5
  headers, 83
  IBM PC AT-compatible, 5
  keyboard, 5
  sample block device driver, 109
  sample disk device driver, 109
  sample disk driver, 79
  sample serial driver, 79
Device drivers
  developing, 79
Device driver
  serial adapter, 5
Device drivers
  interrupt-driven, 80, 81
  noninterrupting, 80, 81
Device driver
  source code, 5
Device drivers
  sample code, 5
Device driver
  structure, 83
  writing, 5
Directories
  notational conventions, 6
Display bytes command, DOS-resident debugger
  abbreviation, 36
  comments, 36
  purpose, 36
  syntax, 36
Display command, DOS-resident debugger
  abbreviation, 35
  comments, 35
  purpose, 35
  syntax, 35
Display current process command, DOS-resident debugger
  abbreviation, 32
  comments, 32
  purpose, 32
  syntax, 32
Display DOS-resident Debugger data structure command, DOS-resident debugger >>>>.xx 0 "Display doublewords command, DOS-resident debugger"
  abbreviation, 37
  comments, 37
  purpose, 37
  syntax, 37
Display expression command, DOS-resident debugger
Display expression command, DOS-resident debugger *(continued)*
  abbreviation, 27
  comments, 27
  purpose, 27
  syntax, 27
Display GDT command, DOS-resident debugger
  abbreviation, 37
  comments, 37
  purpose, 37
  syntax, 37
Display IDT command, DOS-resident debugger
  abbreviation, 38
  comments, 38
  purpose, 38
  syntax, 38
Display LDT command, DOS-resident debugger
  abbreviation, 39
  comments, 39
  purpose, 39
  syntax, 39
Display memory manager handle command, DOS-resident debugger
  abbreviation, 31
  comments, 31
  purpose, 31
  syntax, 31
Display physical memory command, DOS-resident debugger
  abbreviation, 29
  comments, 29
  purpose, 29
  syntax, 29
Display TSS command, DOS-resident debugger
  abbreviation, 39
  comments, 39
  purpose, 39
  syntax, 39
Display words command, DOS-resident debugger
  abbreviation, 40
  comments, 40
  purpose, 40
  syntax, 40
Distribution disks, 3
Documentation
  additions, 5
  changes, 5
  readme files, 5
  summary, 3
DOS-resident debugger, 15
DOS-resident debugger command

DOS-resident debugger command *(continued)*
breakpoint clear, 32
breakpoint disable, 32
breakpoint enable, 33
breakpoint list, 33
breakpoint set, 34
change default, 55
change map file 54
compare, 35
conditional execute, 44
default, 55
display, 35
display bytes, 36
display current process, 32
display DOS-resident Debugger data structure, 3
display doublewords, 37
display expression, 27
display GDT, 37
display IDT, 38
display LDT, 39
display memory manager handle, 31
display physical memory, 29
display TSS, 39
display words, 40
enter, 40
external help command 29
fill, 42
go, 42
help, 26
hex, 43
input, 44
interrupt vector clear, 52
interrupt vector list, 53
interrupt vector set, 54
list Group, 45
list map, 45
list near, 46
List Symbols, 46
move, 46
output, 47
ptrace, 47
reboot, 55
register, 48
DOS-resident debugger commands, 24
DOS-resident debugger command
search, 50
DOS-resident debugger commands
list of, 24
DOS-resident debugger command
trace, 51
unassemble, 52
Dos-resident debugger error messages, 147
description of, 147
Dos-resident debugger error messages *(continued)*
list of, 147
DOS-resident debugger
booting, 16
command delimiters, 18
command parameters
addr, 19
byte, 19
description, 18
dword, 19
expression, 19
group-name, 19
list, 19
map-name, 19
range, 20
register-name, 20
string, 20
word, 20
illegal strings, 20
initial screen display, 16
making a boot disk, 15
\$, 57
\b, 57
\\, 57
\n, 57

Enter command, DOS-resident debugger
abbreviation, 40
comments, 40
purpose, 40
syntax, 40
Environment variables
notational conventions, 6
Equipment. *See* Hardware requirements
Error messages, 147
\t, 57
Execute default command, DOS-resident debugger
abbreviation, 55
comments, 55
purpose, 55
syntax, 55
Expression, DOS-resident debugger, 19
expression
28
special, 28
External help command, DOS-resident debugger
abbreviation, 29
comments, 29
purpose, 29
syntax, 29

Filenames
  notational conventions, 6
Fill command, DOS-resident debugger
  abbreviation, 42
  comments, 42
  purpose, 42
  syntax, 42
os2dos.com, 15
os2dosd.com", 15
Flags, 49
Flowcharts
  automated tests, 151
Functions
  notational conventions, 6

Generic IOCtl functions, 109
Go command, DOS-resident debugger
  comments, 42
  purpose, 42
  syntax, 42

Hardware requirements, 9
  development system, 5, 9
  target system, 5
Help command, DOS-resident debugger
  abbreviation, 26
  comments, 26
  purpose, 26
  syntax, 26
Help screen, 26
Hex command, DOS-resident debugger
  abbreviation, 43
  comments, 43
  purpose, 43
  syntax, 43

IDT, 75
Illegal strings, DOS-resident debugger, 20
Input command, DOS-resident debugger
  abbreviation, 44
  comments, 44
  purpose, 44
  syntax, 44
INT 3 instructions, 59
  use in debugging, 59
Intel 80286 microprocessor
  memory architecture, 4
Intel 80386 microprocessor
  memory architecture, 4
Interrupt Descriptor Table, 75
Interrupt Information Table, 72
Interrupt routine, 109
  sample serial device driver, 95
Interrupt vector clear command, DOS-resident debugger
  abbreviation, 52
  comments, 52
  purpose, 52
  syntax, 52
Interrupt vector list command, DOS-resident debugger
  abbreviation, 53
  comments, 53
  purpose, 53
  syntax, 53
Interrupt vector set command, DOS-resident debugger
  abbreviation, 54
  comments, 54
  purpose, 54
  syntax, 54
Interrupts
  accessing, 88
  requesting, 88
IRQInfoTable, 72
IRQInfoTable entries, 72
  IRQI.3xBoxVector, 74
  IRQI.Flags, 73
  IRQI.IRQNumber, 73
  IRQI.NewDD, 74
    IRQI.ProtDS, 74
    IRQI.ProtVector, 74
    IRQI.RealIDS, 74
    IRQI.RealVector, 74
  IRQI.RealIDTOffset, 74
  IRQI.RealRouter, 74
  IRQI.SharedCount, 74

Keywords
  notational conventions, 6

Letters, lowercase, 17
Letters, uppercase, 17
List default command, DOS-resident debugger
  abbreviation, 55
  comments, 55
  purpose, 55
  syntax, 55
List Group command, DOS-resident debugger
  abbreviation, 45
  comments, 45
  purpose, 45
  syntax, 45

List Map command, DOS-resident debugger
  abbreviation, 45
  comments, 45
  purpose, 45
  syntax, 45
List Near command, DOS-resident debugger
  abbreviation, 46
  comments, 46
  purpose, 46
  syntax, 46
List Symbols command, DOS-resident debugger
  abbreviation, 46
  comments, 46
  purpose, 46
  syntax, 46
Loading symbol map files
  display message, 17
Logical operators
  notational conventions, 6
Lowercase letters, 17

m, list map command, 17
MAP, 60
Map file
  example, 60
  how to read, 60
  reading, 60
Map files, 59
  use in debugging, 59
mapsym program, 16
  generate symbolic map file, 16
  version number required, 16
Memory requirements
  development system, 9
Move command, DOS-resident debugger
  abbreviation, 46
  comments, 46
  purpose, 46
  syntax, 46
MS OS/2 device drivers
  developing, 79
  samples, 79
  testing, 79
MS OS/2 error messages, 5

Notational conventions, 6

Operators
  binary, 21
Operators *(continued)*
  unary, 22
Optional hardware
  development system, 9
Options
  notational conventions, 6
Output command, DOS-resident debugger
  abbreviation, 47
  comments, 47
  purpose, 47
  syntax, 47
Overview of ADD tests, 68
Overview of automated device driver tests, 68

Peripheral device controllers, 4
Physical address, 27
Protected-mode address, 27
Ptrace command, DOS-resident debugger
  abbreviation, 47
  comments, 47
  purpose, 47
  syntax, 47

readme file, 5
Real-mode address, 27
Reboot command, DOS-resident debugger
  abbreviation, 55
  comments, 55
  purpose, 55
  syntax, 55
Reference material, 6
References, 149
Register command, DOS-resident debugger
  abbreviation, 48
  comments, 48
  purpose, 48
  syntax, 48
Register names
  notational conventions, 6
Related publications, 149
Request packets, 85
  sample serial device driver, 86
ROM BIOS interlock, 109
Running ADD tests, 66
Running automated device driver tests, 66
Run-time selector address, 59
  how to get, 59

Sample device drivers
  disk device driver, 82
  serial device driver, 82
Sample serial device driver
  character queues, 87
  debugging, 100
  file organization, 84
  interrupt routine, 85, 95
  parameters, 83
  strategy routine, 88
Search command, DOS-resident
    debugger
  abbreviation, 50
  comments, 50
  purpose, 50
  syntax, 50
Segmented memory architecture, 4
Segment
  use with DOS-resident debugger, 18
Selector
  use with DOS-resident debugger, 18
Software requirements
  development system, 5, 10
  target system, 5
Strategy routine, 85, 109
  function in disk device driver, 111
  sample serial device driver, 88
String, DOS-resident debugger, 20
Summary of ADD tests, 70
Summary of automated device driver
    tests, 70
.sym extension, 16
  symbol map file, 16
Symbol map file, 16
  how to generate, 16
Symbol map files
  active, 17
  loading, 17
Symbol map file
  use with base device drivers, 17
  use with supplemental device drivers,
    17
symbolic debug support, 16
Symbolic debug support
  device drivers, 59
  DOS-resident debugger, 16
Syntax statements
  notational conventions, 6
System calls
  notational conventions, 6

Target system
  automated tests, 5
  configuring, 16
  use with DOS-resident debugger, 16
Terminal emulator, 18
  use with DOS-resident debugger, 18
Testing the system
  target system, 5
Trace command, DOS-resident
    debugger
  abbreviation, 51
  comments, 51
  purpose, 51
  syntax, 51

Unary operators
  list of, 22
Unassemble command, DOS-resident
    debugger
  abbreviation, 52
  comments, 52
  purpose, 52
  syntax, 52
Uppercase letters, 17

Variables
  notational conventions, 6

w, change map command, 17