intel

iRMX®
System Debugger Reference Manual

Order Number: 462920-001

iRMX®
System Debugger Reference Manual

Order Number: 462920-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are located directly after the reader reply card in the back of the manual.

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above	iLBX	iPSC	Plug-A-Bubble
BITBUS	i_m	iRMX	PROMPT
COMMputer	iMDDX	iSBC	Promware
CREDIT	iMMX	iSBX	QUEST
Data Pipeline	Insite	iSDM	QueX
Genius	int$_e$l	iSSB	Ripplemode
Δ i	Intel376	iSXM	RMX/80
i	Intel386	Library Manager	RUPI
I^2ICE	int$_e$lBOS	MCS	Seamless
ICE	Intelevision	Megachassis	SLD
iCEL	int$_e$ligent Identifier	MICROMAINFRAME	UPI
iCS	int$_e$ligent Programming	MULTIBUS	VLSiCEL
iDBP	Intellec	MULTICHANNEL	376
iDIS	Intellink	MULTIMODULE	386
	iOSP	OpenNET	386SX
	iPDS	ONCE	
	iPSB		

XENIX, MS-DOS, Multiplan, and Microsoft are trademarks of Microsoft Corporation. UNIX is a trademark of Bell Laboratories. Ethernet is a trademark of Xerox Corporation. Centronics is a trademark of Centronics Data Computer Corporation. Chassis Trak is a trademark of General Devices Company, Inc. VAX and VMS are trademarks of Digital Equipment Corporation. Smartmodem 1200 and Hayes are trademarks of Hayes Microcomputer Products, Inc. IBM, PC/XT, and PC/AT are registered trademarks of International Business Machines. Soft-Scope is a registered trademark of Concurrent Sciences.

REV.	REVISION HISTORY	DATE
-001	Original Issue.	03/89

PREFACE

INTRODUCTION

The iRMX® System Debugger is a memory-resident extension of the iSDM™ System Debug Monitor. The System Debugger gives you a static debugging tool that can recognize and display all iRMX objects. It enables you to examine your iRMX system interactively so you can find and correct errors.

READER LEVEL

This manual is intended for application engineers familiar with the concepts and terminology introduced in the *iRMX® II Nucleus User's Guide* or the *iRMX® I Nucleus User's Guide* and the system programmers implementing device drivers, object managers, and operating system extensions.

MANUAL OVERVIEW

This manual consists of the following chapters:

Chapter 1	INTRODUCTION--This chapter describes the features of the System Debugger, illustrates how the System Debugger relates to EPROM-based debugging tools, and explains how to use the System Debugger. Read this chapter if you are going through the manual for the first time.
Chapter 2	SYSTEM DEBUGGER COMMANDS--This chapter contains detailed descriptions of the System Debugger commands, presented in alphabetical order. When debugging your system, refer to this chapter for specific information about the format and parameters of the commands.
Chapter 3	SAMPLE DEBUG SESSION--This chapter shows in a step-by-step fashion how to use System Debugger features. The chapter contains a sample debugging session showing how to use iSDM monitor and System Debugger commands to locate an application-code error, correct it, and test the change. Separate examples showing additional debugging techniques are also included. Use this chapter as a hands-on introduction to the System Debugger.

Appendix A	iSDM MONITOR COMMANDS--This appendix briefly describes the function of all basic iSDM monitor commands. Use this appendix as a quick reference to the iSDM monitor. For more information see the *iSDM™ System Debug Monitor User's Guide*.
Appendix B	D-MON386 MONITOR COMMANDS--This appendix briefly describes the function of all basic D-MON386 monitor commands. For more information, refer to the *D-MON386 Debug Monitor for the 80386 User's Guide*.

CONVENTIONS

This manual uses the following format conventions:

- User input appears in one of the following forms:

```
as blue text
```

```
as bolded text within a screen
```

- The text <CR> appears where you must enter a carriage return. When pressing the carriage return key, the text <CR> does not appear on the console.
- Although all syntax diagrams show uppercase letters (e.g., VR), you can also use lowercase letters.
- All numbers unless otherwise stated are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).
- The term "iRMX II" refers to the iRMX II (iRMX 286) Operating System.
- The term "iRMX I" refers to the iRMX I (iRMX 86) Operating System.
- The terms "iRMX" or "iRMX Operating Systems" when used by themselves, refer to both iRMX I and II, that is, the text applies equally to both operating systems.

CONTENTS

Chapter 1. iRMX® System Debugger Overview

Chapter 2. System Debugger Commands

Chapter 3. Sample Debug Session

Appendix A. iSDM™ Monitor Commands

Appendix B. D-MON386 Commands

Appendix B. D-MON386 Commands (continued)

Index

Figures

iRMX® SYSTEM DEBUGGER OVERVIEW 1

1.1 INTRODUCING THE iRMX® SYSTEM DEBUGGER

When you develop application systems, you need debugging capabilities on your development system. Besides the iSDM™ System Debug Monitor, Intel provides the iRMX System Debugger (SDB) for debugging your iRMX-based application system.

NOTE

The remainder of this manual uses the term "monitor" to refer to the iSDM System Debug Monitor.

The System Debugger is a memory-resident extension of the monitor; therefore, you must have the monitor if you have the System Debugger configured into your system. The monitor provides code disassembly, execution breakpoints, memory display, and program download capabilities. The System Debugger extends the monitor's disassembly functions by interpreting iRMX calls, data structures, and stacks.

Monitor and System Debugger commands are entered in response to the iSDM Monitor's protected-mode prompt (..) or the iRMX I real mode prompt (.). When you invoke the monitor, both the operating system and your application system are frozen. As you use monitor commands to set breakpoints while the application code is executed, you can inspect system objects, change system call parameters and registers, and test changes. Refer to Appendix A for more information on iSDM Monitor commands and Appendix B for D-MON386 Monitor commands.

1.2 SUPPORTING THE SYSTEM DEBUGGER

To use the System Debugger, you must have one of the following hardware configurations with all the required support hardware:

- An Intel Microcomputer connected to an 8086-, 80186-, 80188-, 80286- or 386™-based board
- A terminal connected directly to an 8086-, 80186-, 80188-, 80286- or 386™-based board
- An Intellec® Development System connected to an 8086-, 80286- or 386™-based board

Besides the above hardware, you must have both of the following:

- The EPROM portion of the iSDM System Debug Monitor
- An iRMX operating system configuration

1.3 CONFIGURING THE SYSTEM DEBUGGER

You cannot use the System Debugger until you include it in your system through the Interactive Configuration Utility (ICU). To include the System Debugger, begin by invoking the ICU. Next, provide the following information the ICU requires to configure the System Debugger:

1. In the ICU's "Sub-Systems" screen, respond "yes" to the SDB prompt.
2. In the ICU's "System Debugger" screen, set the interrupt level you want to use to invoke the monitor manually (by pressing a hardware interrupt button).

   To use the Non-Maskable Interrupt (NMI) for debugging device drivers, see the *iRMX® II Hardware and Software Installation Guide* or the *iRMX® I Hardware and Software Installation Guide* .

For detailed information on configuring the System Debugger, consult the *iRMX® II Interactive Configuration Utility Reference Manual* or the *iRMX® I Interactive Configuration Utility Reference Manual.*

1.4 INVOKING THE SYSTEM DEBUGGER

You must enter the monitor to use the System Debugger. You can invoke the monitor in three ways:

1. Use a hardware switch physically connected to the interrupt level you specified during configuration. Activating this switch halts the application system, saves the system's contents, and passes control to the monitor.
2. Use the Human Interface DEBUG command. DEBUG loads your specified application program into main memory and transfers control to the monitor.
3. Use the Bootstrap Loader DEBUG switch. When you specify this switch, the monitor comes up after the system is loaded but before the system starts running. The CS:IP points to the first instruction of the application system. At this point the system has not been initialized; therefore, you can run only monitor commands. Using the MAP286 output (or MAP86 in iRMX I), you can identify where you want to insert breakpoints. (For more information on BIND, MAP, and OVL, see the *80286 Utilities User's Guide for iRMX® II Systems* or the *86, 88 Utilities Reference Manual* in iRMX I). Use the break address parameter in the monitor's GO (G) command to set breakpoints in the application system code. When you enter "G <CR>", the system starts and is initialized. The monitor is invoked when CS:IP reaches a breakpoint. For more information on booting with DEBUG, consult the *iRMX® Bootstrap Loader Reference Manual.*

When you invoke the monitor, the application system stops running and all system activity freezes. The appropriate prompt appears (the ".." for the iSDM Monitor, or a single "." in iRMX I), and you can begin entering System Debugger and monitor commands to examine system objects.

1.5 USING THE SYSTEM DEBUGGER

The System Debugger uses monitor procedures to parse the command line and to output to the console; therefore, you run both System Debugger and monitor commands from the monitor. The syntax for System Debugger commands is a "V" or "v" followed by another letter, an optional space, and an optional parameter.

The fifteen System Debugger commands (described in Chapter 2) fall into four categories:

1. Eight commands extend the monitor memory display functions by displaying iRMX data structures and objects.
2. Three commands extend the monitor disassembly functions by recognizing and displaying iRMX calls.
3. Three commands add the ability to display features of the Message passing Coprocessor (MPC) *(iRMX II only)*.
4. A help command provides a short description of all the commands.

All commands either display information as hexadecimal numbers or try to interpret the information. If the System Debugger cannot interpret the information, it displays the actual hexadecimal value, followed by two question marks.

iRMX II provides two features that enable you to leave the monitor without resetting your system: warm-start and CLI-restart. The warm-start feature reinitializes the system and returns control to the Human Interface at the login level. The CLI-restart feature deletes the current job then returns control to the Command Line Interpreter. Refer to Chapter 2 for more information on these features *(iRMX II only)*.

1.6 RETURNING TO YOUR APPLICATION

Use the monitor's GO command (G) to resume execution of the application

- When you finish debugging your application system with the System Debugger.
- If you want to test the changes you made to the application code.

SYSTEM DEBUGGER COMMANDS 2

2.1 INTRODUCTION

This chapter contains detailed descriptions of the iRMX System Debugger commands. Commands appear in alphabetical order, with the first occurrence of each command appearing in blue at the top of the page. A directory of the commands, divided into functional groups, precedes the command descriptions.

This chapter uses the following conventions:

- "CS:IP" is the Code Segment:Instruction Pointer--The pointer to the instruction that would be executed next if the application system were running. If you specify an IP value (one four-digit hexadecimal number) but not a CS value, the System Debugger uses the current CS as the default base.
- "SS:SP" is the Stack Segment:Stack Pointer--The pointer to the current stack location.
- Entering zero (0) as a value for an optional parameter is the same as omitting the parameter; the default value of the parameter is used.
- All terminal examples assume that the iSDM System Debug Monitor is being used. Thus, example input lines show the iSDM monitor prompt ".." (or a single "." in iRMX I).

2.2 CHECKING VALIDITY OF TOKENS

Many System Debugger commands use iRMX tokens as parameters or display tokens as part of the command output. The iRMX Operating Systems maintain tokens in doubly linked lists. When you enter a token as a parameter, the System Debugger checks the validity of the token by looking at the forward and backward links of the token.

If one of the links is bad, the System Debugger generates an error message along with the standard command output. The token you enter as a parameter always appears as the center value in each line of the token display. The displays for forward- and backward-link errors are as follows:

Forward link ERROR:	4111-->4E85	4111<--4E85-->4155	?FFFF<--4155
Backward link ERROR:	4111-->410F?	4111<--4E85-->4155	4E85<--4155

Arrows to the left indicate backward links; arrows to the right indicate forward links. A question mark before or after a value signifies a forward or backward link error, respectively.

If both links are bad, the System Debugger considers the token invalid. A token may also be invalid

- if it belongs to an object in the deletion process
- if an incorrect token is entered as a parameter in a system call
- if a deleted or unused token is entered as a parameter.

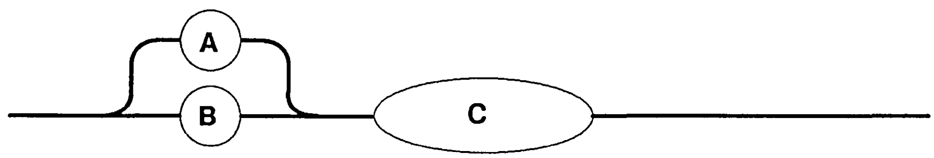
When the token is invalid, the System Debugger displays the following message:

```
*** INVALID TOKEN ***
```

A link error indicates that iRMX data structures have been corrupted. The most common reason for this problem is a task might have accidentally written over part of the system data structures. However, (in the case of the iRMX II Operating System) the iRMX II protection mode feature protects against such overwriting under normal circumstances. Data structure corruption can also occur if you are using the Non-Maskable Interrupt (NMI). The Nucleus may have been interrupted while it was setting up the links. (The NMI is a hardware interrupt. For more information on the NMI, see the *8086 Hardware Reference Manual*, the *80286 Hardware Reference Manual* or the *386™ Hardware Reference Manual.*)

2.3 PICTORIAL REPRESENTATION OF SYNTAX

This chapter uses a schematic device to illustrate command syntax. The schematic consists of what looks like an aerial view of a model railroad, with syntactic elements (appearing in circles) scattered along the track. To construct a valid command, imagine that a train enters the system at the far left, travels from left to right only (backing up is not allowed), chooses one branch at each fork, and finally departs at the far right. The command generated consists of the syntactic elements it encounters on its journey. The following schematic shows two valid sequences: AC and BC.



W-0940

These schematics do not show spaces as elements, but you may include one or more spaces between the command and parameter. For example, even though the syntax for VR is as follows:

W-0941

The following command is valid:

```
..VR xxxx   <CR>
```

The space between "VR" and "xxxx" is optional.

2.4 LEAVING THE MONITOR

iRMX I Note:	The discussion of warm-start and CLI-restart below applies to iRMX II only.

Two features enable you to leave the monitor without resetting your system: warm-start and CLI-restart. You will also leave the monitor when your application terminates normally.

The warm-start feature is the process of starting a system without reloading it from secondary storage. Warm-start reinitializes the system. It begins executing the application system at the same point where the Bootstrap Loader passes control to the system.

To warm-start an iRMX II system from the iSDM monitor, enter the following command:

```
..g 284:0a   <CR>
```

If no system code or data segments were corrupted, the system reinitializes. If segment corruption has occurred, the application system will not run; you must reboot the system.

If your system contains a Command Line Interpreter, and running your application program causes an exception that breaks to the monitor (for example, a General Protection exception), enter the following command to CLI-restart an iRMX II system from the iSDM monitor:

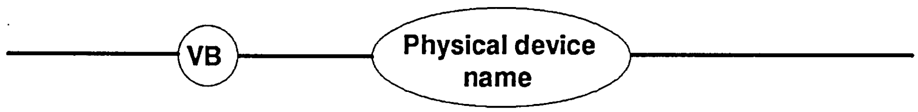
```
..g 284:14   <CR>
```

These commands causes the system to attempt to delete the job tree of the running task. If the running task is part of the application's job (not a subsystem task running for the job) control returns to the Command Line Interpreter. Otherwise, you must reboot the system.

2.5 COMMAND DIRECTORY

VB--DISPLAY DUIB INFORMATION

The VB command displays the DUIB information for the specified physical device. For additional information about Device-Unit Information Blocks (DUIBs), refer to Chapter 4 of the *iRMX® Device Drivers User's Guide*.



W-0942

Parameter

Physical device — The name of the physical device for which you want to view the DUIB information (e.g., WMF0). This device must be part of the system configuration.

Description

The VB command displays the DUIB information for the specified physical device. Figure 2-1 illustrates the output from the VB command.

```
Device name:          <physical device name>

Functs:               xx                  DUIB address      xxxx:xxxx
Dev$gran              xxxx                Max$buffers       xx
Dev$size              xxxxxxxx            Device            xx
Unit                  xx                  Dev$unit          xxxx
Device$info$p         xxxx:xxxx           Unit$info$p       xxxx:xxxx
Update$timeout        xxxx                Num$buffers       xxxx
Priority              xx                  Fixed$update      xx
Init$io               xxxx:xxxx           Finish$io         xxxx:xxxx
Queue$io              xxxx:xxxx           Cancel$io         xxxx:xxxx

Flags:                xx                  Valid
  Density             xxxxxx              Sides             xxxxxx
  Size                x                   Format            xxxxxxxx

File driver:          xxxx                Named             xxxx
  Physical            xxxx                Stream            xxxxx
```

Figure 2-1. Format of VB Output.

The fields displayed in Figure 2-1 are as follows:

Functs	A BYTE used to specify the I/O function validity for this device-unit.
DUIB address	The starting address in memory of the specified DUIB.
Dev$gran	A WORD that specifies the device granularity, in bytes. This parameter applies to random access devices, and to some common devices, such as tape drives. It specifies the minimum number of bytes of information that the device reads or writes in one operation.
Max$buffers	The maximum number of buffers that the EIOS can allocate for a connection to this device-unit when the connection is opened by a call to S$OPEN.
Dev$size	The number of bytes of information that the device-unit can store.
Device	The number of the device with which this device-unit is associated.
Unit	The number of this device-unit, which distinguishes this unit from other units of the device.
Dev$unit	The device-unit number, which distinguishes this device-unit from other device-units in the hardware system.
Device$info$p	A POINTER to a structure that contains additional information about the device. The common, random, and terminal device drivers require a Device Information Table in a specific format, for each device.
Unit$info$p	A POINTER to a structure that contains additional information about the unit. Random access, common device (such as tape drives), and terminal device drivers require this Unit Information Table in a specific format.
Update$timeout	The number of system time units that the I/O System must wait before writing a partial sector, after processing a write request for a disk device.
Num$buffers	The number of buffers of device-granularity size that the I/O System allocates.
Priority	The priority of the I/O System service task for the device.
Fixed$update	Indicates whether the fixed update option was selected for this device-unit when the application system was configured.
Init$io	The address of the Initialize I/O procedure associated with this unit.
Finish$io	The address of the Finish I/O procedure associated with this unit.
Queue$io	The address of the Queue I/O procedure associated with this unit.

Cancel$io	The address of the Cancel I/O procedure associated with this unit.
Flags	Specifies the characteristics of diskette devices.
Valid	Indicates whether the Flags field is "Valid" or "Not Valid" for this device.
Density	The density of the device. If the flags for this DUIB are invalid, this field is marked "N/A".
Sides	The number of media sides that the device can write to. If the flags for this DUIB are invalid, this field is marked "N/A".
Size	The physical size of the device (5 1/4-inch or 8-inch). If the flags for this DUIB are invalid, this field is marked "N/A".
Format	Indicates whether track 0 of a disk is to be formatted as a STANDARD diskette (128 bytes/sector) or as a UNIFORM diskette (all sectors formatted as specified). This parameter applies only to flexible diskettes. Hard disks are always specified as UNIFORM. If the flags for this DUIB are invalid, this field is marked "N/A".
File driver:	A WORD that indicates the BIOS file driver to which this connection is attached.
Named	Indicates whether this device is configured to use the Named file driver.
Physical	Indicates whether this device is configured to use the Physical file driver.
Stream	Indicates whether this device is configured to use the Stream file driver.

Error Messages

Syntax Error
: An error was made when entering the command. The correct syntax is VB <physical device>. Any other syntax produces this message.

VB not supported
: VB couldn't find the byte bucket DUIB entry in the BIOS code segment. If no DUIB entry for the byte bucket exists, VB is unsupported.

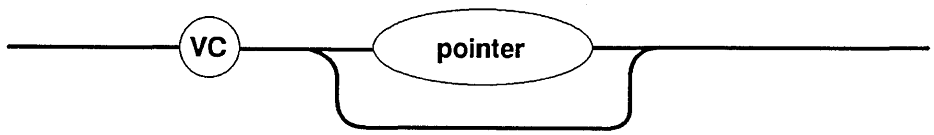
  If the BIOS has not been configured into the system, or if the BIOS code segment has execute-only attributes, this error message is returned.

DUIB not found
: VB returns this error message under these conditions:

  1. The DUIB is not configured into the system.
  2. The DUIB entry for the specified device is located before the byte bucket DUIB entry.
  3. The user made an error while entering the physical device name.

VC--DISPLAY SYSTEM CALL INFORMATION

The VC command checks to see if a CALL instruction is an iRMX system call. The VC command identifies system calls for all iRMX Operating System layers.



W-0943

Parameter

pointer — The address of the CALL instruction to be checked. This parameter can be any valid monitor address (two four-digit hexadecimal numbers separated by a colon).

If you are using the iSDM monitor and you do not supply a pointer (or you specify 0), this parameter defaults to the current CS:IP. If you specify an IP value (one four-digit hexadecimal number) but not a CS value, the System Debugger uses the current CS as the default base.

Description

If the CALL instruction is an iRMX system call, the VC command displays information about the CALL instruction as shown in Figure 2-2.

```
gate #NNNN
(subsystem)system call
```

Figure 2-2. Format of VC Output

The fields in Figure 2-2 are as follows:

gate #NNNN	The gate number associated with the iRMX system call at the address specified in the command.
(subsystem)	The iRMX Operating System layer corresponding to the system call.
system call	The name of the iRMX system call.

NOTE

The System Debugger uses the gate number to determine whether the CALL instruction represents a system call. Since the System Debugger does not disassemble the code, but rather examines a byte value at a particular offset from the CALL instruction, in rare cases a non-system call can be displayed as an iRMX system call. However, the System Debugger does recognize and display all iRMX system calls.

Error Messages

Syntax Error	An error was made in entering the command.
Not a system CALL	The parameter specified points to a CALL instruction that is not an iRMX system call.
Not a CALL instruction	The CS:IP specified does not point to any type of call instruction.

Examples

Suppose you disassembled the following code using the iSDM monitor's Display Memory (DX) command:

```
18A0:006D 50        PUSH    AX
18A0:006E E8AD1E    CALL    A = 1F1E         ;$+7856
18A0:0071 E8DD03    CALL    A = 0451         ;$+992
18A0:0074 B80000    MOV     AX,0
18A0:0077 50        PUSH    AX
18A0:0078 8D060600  LEA     AX,WORD PRT 006
18A0:007C 1E        PUSH    DS
18A0:007D 50        PUSH    AX
18A0:007E E8411E    CALL    A = 1EC2         ;$+7748
18A0:0081 A30000    MOV     WORD PTR 0000H,AX
```

If you use the VC command on the CALL instruction at address 18A0:006E by entering the following command:

```
..VC 18A0:006E  <CR>
```

The System Debugger displays the following information:

```
gate #0468
(Nucleus) set exception handler
```

Gate number 0468 corresponds to an RQSETEXCEPTION$HANDLER system call, a Nucleus call.

Now, suppose you want to see if the CALL instruction at 18A0:0071 is a system call. Enter the following command:

```
..VC 18A0:0071  <CR>
```

The System Debugger responds with the following:

```
Not a system CALL
```

Finally, if you use the VC command on the instruction at 18A0:0074, the System Debugger responds with the following:

```
Not a CALL instruction
```

The VD command displays a job's object directory.

W-0944

Parameter

job token — The token for the job having the object directory you want displayed. To obtain the job token, use the VJ command.

Description

If you specified a valid job token, the System Debugger displays the job's object directory, as shown in Figure 2-3.

```
Directory size:  xxxx                Entries used:  xxxx

name1          token1
name2          tasks waiting         token2...tokeni
.              .
.              .
.              .
namej          tokenj
namek          tokenk
.              .
.              .
.              .
namen          tokenn
```

Figure 2-3. Format of VD Output

Figure 2-3 shows these fields:

Directory size	The maximum number of entries this job can have in its object directory.
Entries used	The number of entries in the directory.
name1...namen	The names under which objects are catalogued. These names were assigned at the time the objects were catalogued with RQ$CATALOG$OBJECT.
token1...tokenn	Tokens for the catalogued objects.
tasks waiting	Signifies that one or more tasks have done an RQ$LOOKUP$OBJECT on an object not catalogued. The tokens following this field identify the tasks still waiting for the object to be catalogued.

For more information on object directories, see the *iRMX® II Nucleus User's Guide* or the *iRMX® I Nucleus User's Guide*.

Error Messages

Syntax Error	No parameter was specified for the command, or an error was made in entering the command.
TOKEN is not a Job	A valid token was entered that is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

Example

Suppose you want to look at the object directory of job "2280". Enter the following command:

```
..VD 2280  <CR>
```

The System Debugger responds with

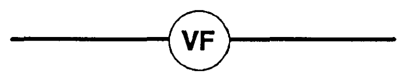
```
Directory size:  000A    Entries used:  0003

$              2228
R?IOUSER       2200
RQGLOBAL       2280
```

The symbols "$", "R?IOUSER", and "RQGLOBAL" are the names of objects the system creates; their respective tokens are 2228, 2200, and 2280. There are no waiting tasks or invalid entries.

VF--DISPLAY NUMBER OF FREE SLOTS

The VF command displays the number of free Global Descriptor Table slots available to the user.



W-0945

Parameters

The VF command has no parameters.

Description

The VF command displays the number of free Global Descriptor Table (GDT) slots available to the user, in the format shown in Figure 2-4.

```
Number of free slots = xxxxxxxx
```

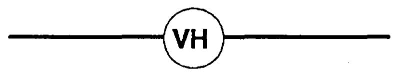
Figure 2-4. Format of VF Output

Error Messages

Syntax Error	An error was made in entering the command.

VH--DISPLAY HELP INFORMATION

The VH command displays and briefly describes the System Debugger commands (iRMX II displays 15 commands; iRMX I displays 12 commands.)



W-0946

Parameters

This command has no parameters.

Description

The VH command lists all the System Debugger commands, along with their parameters and descriptions.

Error Message

Syntax Error	An error was made in entering the command.

Example

If you enter the following command:

```
..VH  <CR>
```

The System Debugger responds as shown in Figure 2-5.

```
iRMX II SYSTEM DEBUGGER, Vx.y
Copyright <year> Intel Corporation

vb <Dev Name>       Displays DUIB for physical device.
vc [<POINTER>]      Display system call.
vd <Job TOKEN>      Display job's object directory.
vf                  Displays number of free slots available to user.
vh                  Display help information.
vj [<Job TOKEN>]    Display job hierarchy from specified level.
vk                  Display ready and sleeping tasks.
vo <Job TOKEN>      Display list of objects for specified job.
vr <Seg TOKEN>      Display I/O Request/Result Segment.
vs [<count>]        Display stack and system call information.
vt <TOKEN>          Display iRMX object.
vu <task TOKEN>     Unwind task stack, displaying system calls.
vmi [<msg #>] [,]   Display the MPC input message buffer.
vmo [<msg #>] [,]   Display the MPC output message buffer.
vmf                 Toggle the MPC fail-safe timeout.
```

Figure 2-5. Format of VH Output

< >	Angle brackets surround required variable fields.
[< >]	Square and angle brackets surround optional fields.

iRMX I Note: The last three lines in the display above apply only to iRMX II. They do not appear in the iRMX I display.

NOTE

The system uses default values if you specify zero (0) for any of the optional parameters in Figure 2-5. Using zero for required parameters causes the system to display the following message:

```
Syntax Error
```

The VJ command displays the portion of the job hierarchy that descends from the level you specify.

W-0947

Parameter

job token — The token of the job for which you want to display descendant jobs.

If you do not specify a job token, or you specify zero (0), VJ displays the root job and its descendant jobs.

If the job has more than 44 generations of job descendants, the System Debugger stops the display at the 44th descendant level, displays an error message, and prompts for another command.

Description

The VJ command displays the token of the specified job and the tokens of all its descendant jobs. It also displays the tokens of jobs (and their descendants) at the same level as the specified job. The tokens for descendant jobs are indented three spaces to show their job's position in the hierarchy. Figure 2-6 shows the format of the job hierarchy display.

iRMX® <I/II> Job Tree

$token_1$	*Root Job*
$token_2$	*Human Interface*
$token_3$	*Command Line Interpreter*
$token_4$	*Application*
$token_5$	*EIOS*
$token_6$	*iRMX-NET (if present)*
$token_7$	*BIOS*

Figure 2-6. Format of VJ Output

The fields in Figure 2-6 are

$token_1$	The token you specified as job token (recall that the root job token is the default).
$token_2 ... token_7$	The tokens for the descendant jobs of $token_1$.

In Figure 2-6, the Human Interface, EIOS, and BIOS Jobs are indented three spaces to signify that they are children of the Root Job. Similarly, the Command Line Interpreter Job is the child of the Human Interface Job (as are all first level user jobs), and the Application Job is the child of the Command Line Interpreter Job.

Error Messages

Syntax Error	An error was made in entering the command.
TOKEN is not a Job	A valid token was entered that is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).
SDB job nest limit exceeded	The specified job (or the default job) has more than 44 generations of job descendants.

Examples

If you want to examine the hierarchy of the root job, enter the following command:

```
..VJ   <CR>
```

Suppose the System Debugger responds with the following job tree:

```
              iRMX® <I/II> Job Tree

0258
    0F38
        1670
            2460
    0E88
    0E00
```

Figure 2-7 shows this job tree:

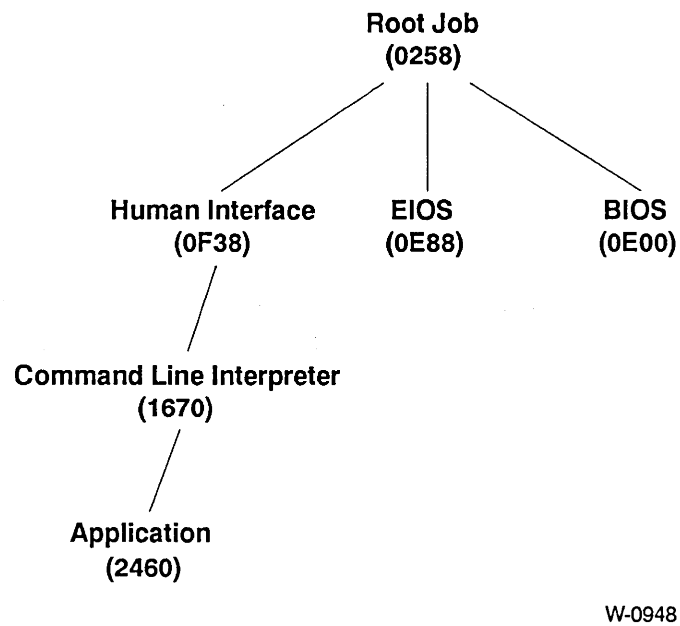


Figure 2-7. iRMX® Job Tree

If you want to display the descendant jobs of "0E88", enter the following command:

```
..VJ 0E88  <CR>
```

The System Debugger displays the following:

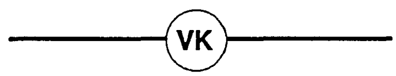
```
        iRMX® Job Tree

    0E88
    0E00
    0F38
        1670
            2460
```

Note that the tokens for all jobs at the same level as the specified token (0E00 and 0F38), and their descendants (1670 and 2460), are also displayed.

VK--DISPLAY READY AND SLEEPING TASKS

The VK command displays the tokens for tasks in the ready and sleeping states.



W-0949

Parameters

This command has no parameters.

Description

The VK command displays the tokens for tasks that are ready and asleep, in the format shown in Figure 2-8.

```
Ready tasks:          xxxx xxxx ...

Sleeping tasks:       xxxx xxxx ...
```

Figure 2-8. Format of VK Output

The fields in Figure 2-8 show the following:

Ready tasks	The tokens for all tasks in the ready state. The first token in this list represents the running task.
Sleeping tasks	The tokens for all tasks in the sleeping state.

Error Messages

Syntax Error	An error was made in entering the command.
Ready tasks: Can't locate	The system is corrupted.
Sleeping tasks: Can't locate	The most common reason for this type of error is not initializing the Nucleus. To recover from this error, reinitialize the system.

Example

To display a list of all the ready and sleeping tasks in your system, enter the following command:

```
..VK  <CR>
```

The System Debugger responds with the following:

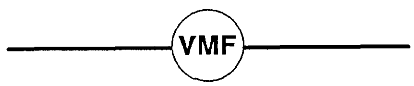
```
Ready tasks:      2F00

Sleeping tasks:   26F0   2588   26B8   2200   21B0   2090   25E8   2050
                  2020   1FF8   2698   2238   2118   2668   2638   2768
                  20D0   0300
```

VMF--TOGGLE MPC FAIL-SAFE TIMEOUT

iRMX I Note:	This command applies only to MULTIBUS® II systems (iRMX II).

The VMF command enables or disables the Message Passing Coprocessor (MPC) fail-safe timeout feature. This command can be used only in a MULTIBUS II system.



W-0608

Parameters

This command has no parameters.

Description

The VMF command enables and disables the fail-safe timer on the Message Passing Coprocessor (MPC). Multiple invocations of this command will alternately enable and disable the fail-safe timer.

The MPC fail-safe timer limits how long (about two seconds on an iSBC® 386/116 or iSBC® 386/120 board) the MPC will wait between sending a buffer request message and receiving a buffer grant or buffer reject message. This hardware timeout ensures that the MPC will not wait forever when trying to communicate with another host that has failed during the buffer negotiation phase. When debugging a message-passing application, it is useful to disable the fail-safe timer so either host may be stopped for indefinite periods while debugging commands are executing. When you are finished debugging, you must use the VMF command to re-enable the fail-safe timer before re-starting your application.

NOTE

The MPC fail-safe timer must be re-enabled before re-starting an application after debugging. Otherwise, your application may not function properly.

NOTE

To use the VMF command, you must specify at least one trace message on the Nucleus Communication Service screen in the ICU. For details on the Number of Trace Messages configuration parameter, see the *iRMX® II Interactive Configuration Utility Reference Manual.*

Error Messages

Syntax Error	An error was made in entering the command.

Example

If you enter the following command:

```
..VMF <CR>
```

one of these two messages will be displayed:

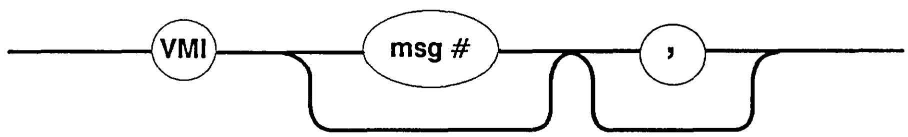
```
MPC Failsafe Timer Is Enabled
```

or

```
MPC Failsafe Timer Is Disabled
```

iRMX I Note:	This command applies only to MULTIBUS II systems (iRMX II).

The VMI command displays the contents of the messages received from the Message Passing Coprocessor (MPC). This command can be used only in a MULTIBUS II system.



W-0609

Parameters

msg # — The number of the message to display. If this parameter is omitted, the most recent message is displayed. If the comma (,) parameter is also entered, this field specifies the first message to display.

, — Specifies that you want to view more than one message in the input message buffer. When you specify this parameter, SDB displays the first message and then displays a special prompt, a dash (-), at the end of the line. If you enter another comma, SDB displays the next most recent message in the input message buffer. The debugger then issues another special prompt (-) and waits for you to either enter another comma or to end the command. You can end the VMI command by entering a carriage return in response to the special prompt (-).

Description

The VMI command displays the field values associated with the input messages received from the MPC input message buffer. These fields are used by the iRMX Nucleus Communication Service, an implementation of the MULTIBUS II Transport Protocol. This section briefly describes each field. For a more detailed description of the fields, refer to the *MULTIBUS® II Transport Protocol Specification and Designer's Guide*.

NOTE

The VMI command displays the most recent messages in the input message buffer. The number of messages you can display depends on how many trace messages you allocate on the Nucleus Communication Service screen in the ICU. For example, if you specify five trace messages, you will be able to display the five most recent messages. To use the VMI command, you must specify at least one trace message. For details on the Number of Trace Messages configuration parameter, see the *iRMX® II Interactive Configuration Utility Reference Manual.*

The format of the VMI output depends on the type of message. Figure 2-8.3 shows the fields that may be displayed.

```
## message type   req_id: xx src_hid:   xx dest_hid:   xx    len: xxxxxx
   trans control trans_id: xx src_pid: xxxx dest_pid: xxxx xmit_c: xx
   len: xxxxxxxx
```

Figure 2-8a. Format of VMI Output

The first line of the display contains hardware-level information about the message. The fields on this line are:

##	The message number.
message type	The type of message (hardware-level protocol). Possible values are Unsolicited, Broadcast, Buf Request, and Unknown Type.
req_id	Request Id. This ID defines a particular message transfer.
src_hid	Host ID of the sender of the message.
dest_hid	Host ID of the receiver of the message.
len	The length (in bytes) of the requested transfer. This field is only displayed for buffer request messages. For other types of messages, this field is blank.

The second line of the display contains software protocol information about the message. If the protocol of the message is not the data trasport protocol, the following is displayed:

```
Unknown Protocol
```

If the protocol being used is the data transport protocol, the following fields are displayed:

trans control
: A representation of the transaction control field of the message. If the message is not a request or response message, this field is blank; otherwise, this field indicates the type of request or response message. Possible values for this field are:

Resp/EOT	Response message, end-of-transaction (EOT). Indicates that this is the last fragment of a reply.
Resp/Not EOT	Response message, not end-of-transaction (EOT). Indicates that more fragments of the reply will follow.
Resp/Cancel	Response message with cancellation. Indicates that the sender of the reply (the server) is cancelling the transaction.
Resp/Reserved	Reserved type. Undefined at present.
Req/Frag Off	Request message with fragmentation disallowed. The request cannot be sent in fragments.
Req/Frag On	Request message with fragmentation allowed. The request can be sent in fragments, if necessary.
Req/Send Frag	Request message, send next fragment. The next fragment of a fragmented transfer can be sent.
Req/Next Frag	Request message containing the next fragment of a fragmented transfer.

trans_id
: Transaction ID. A number that uniquely identifies a transaction. This field will be zero for transactionless messages (unsolicited or solicited messages with no reply expected).

src_pid	The port ID of the sender of the message.
dest_pid	The port ID of the receiver of the message.
xmit_c	Transmission control. The high-order two bits of this field indicate the protection level of the message. Level 0 is the most privileged level and level 3 is the least.

If the trans control field indicates that the message is a Req/Send Frag message, the third line of the display contains the following field:

len	The length (in bytes) of the requested fragment.

Otherwise, the third line shows the user data portion of the control message in hexadecimal words. If the message type or software protocol are unknown, the entire message is displayed in hexadecimal words, beginning on the third line.

NOTE

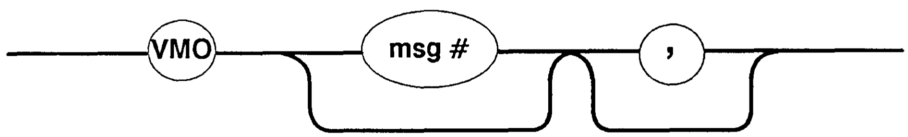
You cannot use the VMI command to view the contents of short-circuit messages.

Error Messages

Syntax Error	An error was made in entering the command.
Message Information Is Not Available	The system is not a MULTIBUS II system or no trace messages were specified during configuration. The number of trace messages is specified on the Nucleus Communication Service screen in the ICU. For details, refer to the *iRMX® II Interactive Configuration Utility Reference Manual.*

iRMX I Note:	This command applies only to MULTIBUS II systems (iRMX II).

The VMO command displays the contents of the output messages sent by the Message Passing Coprocessor (MPC). This command can be used only in a MULTIBUS II system.



W-0610

Parameters

msg # — The number of the message to display. If this parameter is omitted, the most recent message is displayed. If the comma (,) parameter is also entered, this field specifies the first message to display.

, — Specifies that you want to view more than one message in the output message buffer. When you specify this parameter, SDB displays the first message and then displays a special prompt, a dash (-), at the end of the line. If you enter another comma, SDB displays the next most recent message in the output message buffer. The debugger then issues another special prompt (-) and waits for you to either enter another comma or to end the command. You can end the VMO command by entering a carriage return in response to the special prompt (-).

Description

The VMO command displays the field values associated with the output messages sent by the MPC. These fields are used by the iRMX Nucleus Communication Service, an implementation of the MULTIBUS II Transport Protocol. This section briefly describes each field. For a more detailed description of the fields, refer to the *MULTIBUS® II Transport Protocol Specification and Designer's Guide.*

NOTE

The VMO command displays the most recent messages in the output message buffer. The number of messages you can display depends on how many trace messages you allocate on the Nucleus Communication Service screen in the ICU. For example, if you specify five trace messages, you will be able to display the five most recent messages. To use the VMO command, you must specify at least one trace message. For details on the Number of Trace Messages configuration parameter, see the *iRMX® II Interactive Configuration Utility Reference Manual.*

The format of the VMO output depends on the type of message. Figure 2-8.2 shows the fields that may be displayed.

```
## message type    req_id: xx  src_hid:   xx  dest_hid:   xx     YYYYYYY
   trans control  trans_id: xx  src_pid: xxxx  dest_pid: xxxx  xmit_c: xx
   len: xxxxxxxx
```

Figure 2-8b. Format of VMO Output

The first line of the display contains hardware-level information about the message. The fields on this line are:

##	The message number.
message type	The type of message (hardware-level protocol). Possible values are Unsolicited, Broadcast, Buf Request, Buf Grant, Buf Reject, and Unknown Type.
req_id	Request Id. This ID defines a particular message transfer.
src_hid	Host ID of the sender of the message.
dest_hid	Host ID of the receiver of the message.

YYYYYYY — This part of the first line is only displayed for buffer request, buffer grant, and buffer reject messages. It can consist of one of two fields. For buffer request messages, the following field is displayed:

len — The length (in bytes) of the requested transfer.

For buffer grant and buffer reject messages, this field is displayed.

l_id — Liaison ID. This ID binds a buffer grant or buffer reject message to a buffer request message.

The second line of the display contains software protocol information about the message. If the protocol of the message is not the data transport protocol, the following is displayed:

Unknown Protocol

If the protocol being used is the data transport protocol, the following fields are displayed:

trans control — A representation of the transaction control field of the message. If the message is not a request or response message, this field is blank; otherwise, this field indicates the type of request or response message. Possible values for this field are:

Resp/EOT	Response message, end-of-transaction (EOT). Indicates that this is the last fragment of a reply.
Resp/Not EOT	Response message, not end-of-transaction (EOT). Indicates that more fragments of the reply will follow.
Resp/Cancel	Response message with cancellation. Indicates that the sender of the reply (the server) is cancelling the transaction.
Resp/Reserved	Reserved type. Undefined at present.
Req/Frag Off	Request message with fragmentation disallowed. The request can not be sent in fragments.
Req/Frag On	Request message with fragmentation allowed. The request can be sent in fragments, if necessary.

	Req/Send Frag	Request message, send next fragment. The next fragment of a fragmented transfer can be sent.
	Req/Next Frag	Request message containing the next fragment of a fragmented transfer.

trans_id — Transaction ID. A number that uniquely identifies a transaction. This field will be zero for transactionless messages (unsolicited or solicited messages with no reply expected).

src_pid — The port ID of the sender of the message.

dest_pid — The port ID of the receiver of the message.

xmit_c — Transmission control. The high-order two bits of this field indicate the protection level of the message. Level 0 is the most privileged level and level 3 is the least.

If the trans control field indicates that the message is a Req/Send Frag message, the third line of the display contains the following field:

len — The length (in bytes) of the requested fragment.

Otherwise, the third line shows the user data portion of the message in hexadecimal words. If the message type or software protocol are unknown, the entire message is displayed in hexadecimal words, beginning on the third line.

NOTE

You cannot use the VMI command to view the contents of short-circuit messages.

Error Messages

Syntax Error	An error was made in entering the command.
Message Information Is Not Available	The system is not a MULTIBUS II system or no trace messages were specified during configuration. The number of trace messages is specified on the Nucleus Communication Service screen in the ICU. For details, refer to the *iRMX® II Interactive Configuration Utility Reference Manual.*

VO--DISPLAY OBJECTS IN A JOB

The VO command displays the tokens for the objects in the specified job.

W-0950

Parameter

job token — The token of the job for which you want to display objects.

Description

The VO command lists the tokens for a job's:

- child jobs
- tasks
- mailboxes
- semaphores
- regions
- segments
- extensions
- composites
- buffer pools

It uses the format shown in Figure 2-9.

```
Child Jobs:     xxxx    xxxx    xxxx    ...
Tasks:          xxxx    xxxx    xxxx    ...
Mailboxes:      xxxx    xxxx    xxxx    ...
Semaphores:     xxxx    xxxx    xxxx    ...
Regions:        xxxx    xxxx    xxxx    ...
Segments:       xxxx    xxxx    xxxx    ...
Extensions:     xxxx    xxxx    xxxx    ...
Composites:     xxxx    xxxx    xxxx    ...
Buffer Pools:   xxxx    xxxx    xxxx    ...
```

Figure 2-9. Format of VO Output

The fields in Figure 2-9 are as follows:

Child Jobs	The tokens for the specified job's offspring jobs.
Tasks	The tokens for the tasks in the specified job.
Mailboxes	The tokens for the mailboxes in the job. An "o" following a mailbox token means that one or more objects are queued at the mailbox. A "t" following a mailbox token means that one or more tasks are queued at the mailbox.
Semaphores	The tokens for the semaphores in the specified job. A "t" following a semaphore token means that one or more tasks are queued at the semaphore.
Regions	The tokens for the regions in the specified job. A "b" (busy) following a region token means that a task has access to information guarded by the region.
Segments	The tokens for the segments in the specified job.
Extensions	The tokens for the extensions in the specified job.
Composites	The tokens for the composites in the specified job. A "s" following a composite signifies a port with a signal waiting. An "m" signifies a port with a message waiting. A "t" signifies a port with a task waiting.
Buffer Pools	The tokens for the buffer pools in the specified job.

Error Messages

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
TOKEN is not a Job	A valid token was entered; however, it is not a job token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

Example

If you want to look at the objects in the job having the token "1670", enter the following command:

```
..VO 1670  <CR>
```

The System Debugger responds with the following:

```
Child jobs:     2460
Tasks:          1688    1778    17B8    1940    1950    2FF8
Mailboxes:      1720    1728    1738 t  1740 t  1760 t  1768 t
Semaphores:     17A0    17A8 t
Regions:
Segments:       16D8    1750    1958    1960    2FE8    2FC8
Extensions:
Composites:     1690    16F0    1710    1828    1848    1980
Buffer pools:
```

This display shows the tokens for the job, as described earlier. It also tells you that tasks are waiting at four mailboxes and one semaphore.

The VR command displays information about the iRMX Basic I/O System I/O Request/Result Segment (IORS) that corresponds to the segment token you enter.

W-0941

Parameter

Segment token — The token for a segment containing the IORS you want to display. If this segment is not an IORS, the VR command returns invalid information. To obtain a list of the segment tokens in a job, use the VO command.

Description

The VR command displays the names and values for the fields of a specific IORS. The contents of the IORS reflect the most recent I/O operation in which this IORS was used. The System Debugger ensures the specified segment is between 45 and 65 bytes long. It cannot tell whether the segment contains a valid IORS, so you must ensure that it does. If the parameter is a valid segment token for a segment containing an IORS, the System Debugger displays information about the IORS as shown in Figure 2-10. For more information on I/O Request/Result Segments, see the *iRMX® Basic I/O System User's Guide*.

For more detailed information about the IORS contents, see the *iRMX® Device Drivers User's Guide*.

```
I/O Request Result Segment

Status              xxxx       Unit status     xxxx
Device              xxxx       Unit            xx
Function            xxxxxxx    Subfunction     xxxxxxx
Count               xxxx       Actual          xxxx
Device location     xxxxxxxx   Buffer pointer  xxxx:xxxx
Resp mailbox        xxxx       Aux pointer     xxxx:xxxx
Link forward        xxxx:xxxx  Link backward   xxxx:xxxx
Done                xxxxx      Cancel ID       xxxx
Connection token    xxxx
```

Figure 2-10. Format of VR Output

The fields in Figure 2-10 are as follows:

Status — The condition code for the I/O operation.

Unit status — Additional status information. The contents of this field are significant only when the Status field is set to the E$IO condition (002BH). If the Status field is not set to E$IO, the Unit Status field displays "N/A".

Device — The number of the device for which this I/O request is intended.

Unit — The number of the unit for which this I/O request is intended.

Function — The operation done by the Basic I/O System. The possible functions are

Function	System Call
Read	RQAREAD
Write	RQAWRITE
Seek	RQASEEK
Special	RQASPECIAL
Att Dev	RQAPHYSICAL$ATTACH$DEVICE
Det Dev	RQAPHYSICAL$DETACH$DEVICE
Open	RQAOPEN
Close	RQACLOSE

If the Function field contains an invalid value, the System Debugger displays the actual value in this field, followed by a space and two question marks.

Subfunction	An added specification of the function that applies only when the Function field contains "Special" from the BIOS RQASPECIAL system call. Possible subfunctions and their descriptions are

Subfunction	Description
For/Que	Format or Query
Satisfy	Stream file satisfy function
Notify	Notify function
Device char	Device characteristics
Get Term Attr	Get terminal attributes
Set Term Attr	Set terminal attributes
Signal	Signal function
Rewind	Rewind tape
Read File Mark	Read file mark on tape
Write File Mark	Write file mark on tape
Retention Tape	Take up slack on tape
Set Font	Set character font
Set Bad Info	Set bad track/sector information
Get Bad Info	Get bad track/sector information
Get term status	Get terminal status
Cancel I/O	Cancel terminal I/O
Resume I/O	Resume terminal I/O

If the Function field doesn't contain "Special," then the Subfunction field contains "N/A." If the Subfunction field contains an invalid value, the System Debugger displays the value of the field followed by a space and two question marks.

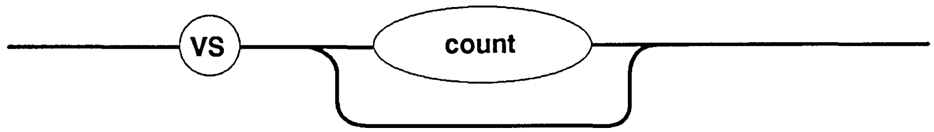
Count	The number of bytes of data called for in the I/O request.
Actual	The number of bytes of data transferred in response to the request.
Device location	The eight-digit hexadecimal address of the byte or logical block where the I/O operation began on the specified device.
Buffer pointer	The address of the buffer the Basic I/O System read from, or wrote to, in response to the request.
Resp mailbox	A token for the response mailbox to which the device sent the IORS after the operation.
Aux pointer	The pointer to the location of auxiliary data, if any. This field is significant only when the Function field contains "Special."
Link forward	The address of the next IORS in the queue where the IORS waited to be processed.
Link backward	The address of the previous IORS in the queue where the IORS waited to be processed.

Done	This field is always present but applies only to IORSs for I/O operations on random-access devices. When applicable, it indicates whether the I/O operation has been completed. The possible values are TRUE (0FFH) and FALSE (00H).
Cancel ID	A word used by device drivers to identify I/O requests that need to be canceled. A value of zero (0) indicates a request that cannot be canceled.
Connection token	The token for the file connection used to issue the request for the I/O operation.

Error Messages

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
TOKEN is not a SEGMENT	The token entered is valid but not a segment token.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).
SEGMENT wrong size - not an IORS	The specified segment is not between 45 and 65 bytes long, so it is not an I/O Request/Result Segment.

The VS command identifies system calls (as does the VC command) and displays the stack.



W-0952

Parameter

count — A decimal or hexadecimal value that specifies the number of words from the stack to be included in the display. A suffix of T, as in 16T, means decimal. No suffix or a suffix of H indicates hexadecimal.

If you do not specify a count, or you specify a count of zero (0), the number of words in the display depends on the number of parameters for the system call at the CS:IP. When CS:IP is not pointing to a system call, the entire contents of the stack are displayed.

Description

The VS command identifies iRMX system calls for all iRMX subsystems (as does the VC command). It interprets the system call parameters on the stack. If the stack does not contain a system call, the VS command displays either the number of stack elements you specify or all the stack contents, whichever is least. If a parameter is a string, the System Debugger displays the string. For additional system call information, see the appropriate iRMX Volume 3 system call manual.

The VS command interprets the CALL instruction at the current CS:IP. If you want to interpret a CALL instruction at a different CS:IP value, you must move the CS:IP to that value. To move the CS:IP using the iSDM monitor, use the GO (G) command or the EXAMINE/MODIFY REGISTER command (X with CS or IP specified as the 8086, 80286 or 386™ register).

If the instruction is not a CALL instruction, VS displays the contents of the words on the stack and no message. If the instruction is a CALL but not a system call (for example, a PL/M call to a procedure), VS displays the stack contents. It also displays a message telling you that the CALL was not a system call.

The VS command uses current values of the SS:SP (Stack Segment:Stack Pointer) registers to display the current stack values. If the instruction is an iRMX system call, VS displays the system call and the stack information as shown in Figure 2-11.

```
gate #NNNN
xxxx:xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx
xxxx:xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx   xxxx

(subsystem)system call

                       |parameters|
```

Figure 2-11. Format of VS Output

The fields in Figure 2-11 are as follows:

xxxx:xxxx	The contents of the SS:SP (stack memory addresses).
xxxx	Values (tokens) now on the stack. The number of stack values varies, depending on the number of parameters in the system call.
parameters	The names of the stack values. The parameters correspond to the stack values directly above them. The maximum number of displayed parameters is 24.

The three remaining fields in Figure 2-11 are the same as those in the VC command:

gate #NNNN	The gate number associated with the system call. *(iRMX II only)*.
(subsystem)	The iRMX Operating System layer that the system call is part of.
system call	The name of the iRMX system call.

Error Messages

Syntax Error	An error was made in entering the command.
Not a system CALL	The CS:IP is pointing to a CALL instruction that is not an iRMX system call.
Unknown entry code	This message indicates that one of two infrequent events has occurred. One is that the System Debugger has mistaken an operand belonging to some instruction in the object code for the FAR CALL instruction. The other event is that a software link from user code into iRMX code has been corrupted. To recover from system corruption, reboot the system.

Examples

Suppose you determine that the SS:SP is 1906:07CA (using the iSDM Monitor's X command, for example) then use the VS command by entering the following command:

```
..VS  <CR>
```

The System Debugger responds with the following:

```
gate #0360
1906:07CA   0B08   1980   1EA8   1980   1980   0000   0B00   1908
1906:07DA   19A0   0B20   0580   1EA8   1EA0   1EE8   0000   0000

(Nucleus) delete mailbox

            |..excep$p..|.mbox.|
```

The parameter names identify the stack values directly above them. That is, the "excep$p" parameter name signifies that the first two words represent a pointer (1980:0B08) to the exception code. Similarly, the "mbox" parameter signifies that the third word (1EA8) is the token for the mailbox being deleted.

Now, suppose that you move the SS:SP to 2906:07D0. If you invoke the VS command by entering the following command:

```
..VS  <CR>
```

The System Debugger displays the following stack and a message informing you that the instruction is a CALL instruction but not an iRMX system call:

```
2906:07D0   2980   2980   0000   0600   2908   29A0   0020   1580
2906:07E0   27C8   27C8   25C8   25C8   25C8   25C8   25C8   25C8

Not a system CALL
```

When an iRMX system call is executed, its parameters are pushed onto the current stack, and then a CALL instruction is issued with the appropriate stack address. If the call has more parameters than will fit on one line, the System Debugger automatically displays multiple lines of stack values. It shows corresponding multiple lines of parameter descriptions directly below them.

For example, suppose you use the VS command as follows:

```
..VS  <CR>
```

```
gate #0310
27CC:0F9A    0158   20C8   0000   20C8   20C8   0000   0600   17C8
27CC:0FAA    20E8   0028   0000   0000   20C8   00E0   2FF8   2FF8
27CC:0FBA    2608   1A58   1AF8   2608   0000   0000   0000   0000

(Nucleus) create job

             |...excep$p...|.t$flgs.|stksze|..sp..|..ss..|..ds..|..ip..|
             |..cs..|..pri.|.j$flgs.|.exp$info$p..|maxpri|maxtsk|maxobj|
             |poolmx|poolmn|.param..|dirsiz|
```

This display indicates that the CALL instruction is a Nucleus RQ$CREATE$JOB system call with 18 parameters. The names of these parameters are shown between the vertical bars (|). The words on the stack correspond to the parameters directly below them.

The following display shows that the CALL instruction is a Basic I/O System (BIOS) RQAATTACH$FILE system call with five parameters. The "subpath$p" parameter points to a string seven characters long: the word "example."

```
..VS  <CR>
```

```
gate #0500
27CC:0F4E    0F88   17C8   25F8   0000   2600   29A0   0000   2600
27CC:0F5E    2608   1C10   2600   1320   26D0   0F78   0DF8   2FF8

(BIOS) attach file

             |....excep$p...|.mbox.|..subpath$p..|.prefix|.user|
subpath--> 07'example'
```

The following display indicates that the CALL instruction at CS:IP is an Extended I/O System RQSRENAME$FILE system call with three parameters. Two of the parameters have strings: the "new$path$p" parameter points to a string four characters long ("XY70"); the "path$p" parameter points to another string four characters long ("temp").

```
..VS  <CR>
```

```
gate #06E8

27CC:0F98   0148   20C8   0858   20E8   06A0   20E8   0000   0600
27CC:0FA8   17C8   20E8   0028   1320   0000   20C8   0008   2600

(EIOS) rename file

            |..excep$p..|..new$path$p..|...path$p...|
new path--> 04'XY70'
path--> 04'temp'
```

NOTE

If a string is more than 50 characters long, the System Debugger displays only the first 50 characters. If the pointer is pointing to a nonreadable segment, the System Debugger does not display the string.

VT--DISPLAY iRMX® OBJECT

The VT command displays information about the iRMX object associated with the token you enter.

W-0953

Parameter

token — The token of the object for which you want to display information.

Description

The VT command determines the type of iRMX object represented by the token and displays information about that object. Both the information and the format in which the System Debugger displays the information depend on the type of object.

The following sections are divided into display groups illustrating the display format for these iRMX objects:

- Jobs
- Tasks
- Mailboxes
- Semaphores
- Regions
- Segments
- Extensions
- Composite objects (six types)
- Buffer Pools *(iRMX II only)*

Error Messages

Syntax Error	No parameter was specified for the command or an error was made in entering the command.
*** INVALID TOKEN ***	The value entered for the token is not a valid token (as defined in "Checking Validity of Tokens" earlier in this chapter).

Job Display

If the parameter you specify is a valid job token, the System Debugger displays information about the job having that token, as Figure 2-12 shows.

```
Object type = 1 Job

Current tasks    xxxx        Max tasks      xxxx    Max priority    xx
Current objects  xxxx        Max objects    xxxx    Parameter obj   xxxx
Directory size   xxxx        Entries used   xxxx    Job flags       xxxx
Except handler   xxxx:xxxx   Except mode    xx      Parent job      xxxx
Pool min         xxxxx       Pool max       xxxxx   Initial size    xxxxx
Borrowed         xxxxx

       Byte range  |  Number chunks    |  Largest chunk    |  Total memory
   ----------------|-------------------|-------------------|--------------
       22-44H      |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       44-84H      |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       84-200H     |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       200H-1K     |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       1K-2K       |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       2K-4K       |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       4K-8K       |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       8K-32K      |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
       +32K        |   xxxxxxxx        |   xxxxxxxx        |   xxxxxxxx
```

Figure 2-12. Format of VT Output: Job Display

The fields in Figure 2-12 (from left to right) are as follows:

Current tasks — The number of tasks currently existing in the job. If the Max tasks is not 0FFFFH (no limit), the number of Current tasks is equal to the Current tasks of this job plus all its children Max tasks.

Max tasks — The maximum number of tasks that can exist in the job simultaneously. This value was set when the job was created.

Max priority — The maximum (numerically lowest) priority allowed for any one task in the job. This value was set when the job was created.

Current objects — The number of objects currently existing in the job.

Max objects — The maximum number of objects that can exist in the job simultaneously. This value was set when the job was created.

Parameter obj	The token for the object that the parent job passed to this job. This value was set when the job was created.
Directory size	The maximum number of entries the job can have in its object directory. This value was specified by the first parameter when the job was created with the Nucleus RQ$CREATE$JOB system call or the RQE$CREATE$JOB system call *(iRMX II only)*.
Entries used	The number of objects now catalogued in the job's object directory.
Job flags	The job flags parameter specified when the job was created. It contains information the Nucleus needs to create and maintain the job.
Except handler	The start address of the job's exception handler. This address was set when the job was created.
Except mode	The value that indicates when control is to be passed to the new job's exception handler. This value was set when the job was created.
Parent job	The token for the specified job's parent.
Pool min	The minimum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Pool max	The maximum size (in 16-byte paragraphs) of the job's memory pool. This value was set when the job was created.
Initial size	The initial size (in 16-byte paragraphs) of the job's memory pool.
Borrowed	The current amount (in 16-byte paragraphs) of memory that the job has borrowed from its ancestor(s).
Free Space	All free memory in a job's pool is accounted for, via several double-linked lists. Each list contains a range of chunk sizes. A chunk is a piece of contiguous memory. Column one of the free space table shows the size ranges for the list. Column two shows the number of chunks on each list. Column three displays the largest chunk on each list. Column four shows the total amount of memory on each list.

Task Display

The System Debugger displays information about tasks in two different ways. Figure 2-13 shows the display for non-interrupt tasks, and Figure 2-14 shows the display for interrupt tasks.

```
Object type = 2 Task

Static pri       xx          Dynamic pri     xx      Task state      xxxxxxxxx
Suspend depth    xx          Delay req       xxxx    Last exchange   xxxx
Except handler   xxxx:xxxx   Except mode     xx      Task flags      xx
Containing job   xxxx        Interrupt task  no      K-saved SS:SP   xxxx:xxxx
```

Figure 2-13. Format of VT Output: Non-Interrupt Task

```
Object type = 2 Task

Static pri       xx          Dynamic pri     xx      Task state      xxxxxxxxx
Suspend depth    xx          Delay req       xxxx    Last exchange   xxxx
Except handler   xxxx:xxxx   Except mode     xx      Task flags      xx
Containing job   xxxx        Interrupt task  yes     Int level       xx
Master mask      xx          Slave mask      xx      Pending int     xx
Max interrupts   xx          K-saved SS:SP   xxxx:xxxx
```

Figure 2-14. Format of VT Output: Interrupt Task

The fields in Figures 2-13 and 2-14 (from left to right) are as follows:

Static pri — The maximum priority value of the task. This value was set by the max$priority parameter when the task's containing job was created with RQ$CREATE$JOB or RQE$CREATE$JOB. *(iRMX II only)*

Dynamic pri	A temporary priority that the Nucleus sometimes assigns to the task to improve system performance. For example, if a higher priority task wants control of a region that belongs to a currently executing lower priority task, the Nucleus assigns the lower priority task a priority equal to that of the higher priority task. This increasing of a task's priority improves the total system performance here.
Task state	The state of the task. The twelve possible states, as they are displayed, are

State	Description
ready	task is ready for execution
asleep	task is asleep
susp	task is suspended
aslp/susp	task is both asleep and suspended
deleted	task is being deleted
on exch Q	task is waiting at an exchange
aslp/exch	task is asleep waiting at an exchange
sl/xc/susp	task is asleep and suspended waiting at an exchange
on port Q	task is queued at a port
aslp/port	task is asleep waiting at a port
on trans Q	task is queued at a port on transaction queue
aslp/trans	task is asleep and queued at port on transaction queue

If this field contains an invalid value, the System Debugger displays the value followed by a space and two question marks.

Suspend depth	The number of RQ$SUSPEND$TASK system calls that have been applied to this task without corresponding RQ$RESUME$TASK system calls.
Delay req	The number of sleep units the task requested when it last specified a delay at a mailbox or semaphore, or when it last called RQ$SLEEP. If the task has not done any of these, this field contains zeros.
Last exchange	The token for the mailbox, region, or semaphore at which the task most recently began to wait.

Except handler	The start address of the job's default exception handler. This value was set either when the task was created with RQ$CREATE$TASK, RQ$CREATE$JOB, RQE$CREATE$JOB, or later with RQSETEXCEPTION$HANDLER.
Except mode	The value that indicates the exceptional conditions under which control is to be passed to the new task's exception handler. This value was set either when the task was created with RQ$CREATE$TASK, RQ$CREATE$JOB, RQE$CREATE$JOB, or later with RQSETEXCEPTION$HANDLER.
Task flags	The task flags parameter used when the task was created with RQ$CREATE$TASK. It contains information the Nucleus needs to create and maintain the job's initial task.
Containing job	The token of the job that this task belongs to.
Interrupt task	Indicates whether this task is an interrupt task. "No" signifies that the task is not an interrupt task. Here, only the K-saved field follows in the display. (See Figure 2-13.) "Yes" signifies that the task is an interrupt task. In this case, additional fields appear in the display. (See Figure 2-14.)
K-saved SS:SP	The contents of the SS:SP registers when the task last left the ready state.
Int level	The level that the interrupt task services. This level was set when this task called RQSETINTERRUPT.
Master mask	The value associated with the interrupt mask for the master interrupt controller. This value represents the master interrupt levels disabled by the interrupt level that the task services. For example, if the task services master interrupt level 68H, then master levels 6 and 7 are disabled, so the master mask field is 11000000B (=0C0H). For more information about interrupt levels, see the *iRMX® II Nucleus User's Guide* or the *iRMX® I Nucleus User's Guide*.
Slave mask	The value associated with the interrupt mask for a slave interrupt controller. This value represents the slave interrupt levels disabled by the level that the task services. For example, if the task services slave interrupt level 62H, then slave levels 2 through 7 are disabled, so the slave level field is 11111100B (=0FCH). For more information about interrupt levels, see the *iRMX® II Nucleus User's Guide* or the *iRMX® I Nucleus User's Guide*.
Pending int	The number of RQ$SIGNAL$INTERRUPT calls pending for the Int level.

Max interrupts — The maximum number of RQ$SIGNAL$INTERRUPT calls that can be pending for the Int level.

Mailbox Display

The System Debugger displays information about mailboxes in three different ways:

- Figure 2-15 shows the display when nothing is queued at the mailbox.
- Figure 2-16 shows the display when tasks are queued at the mailbox.
- Figure 2-17 shows the display when objects are queued at the mailbox.
- Figure 2-18 shows the display when data messages are queued at the mailbox.

```
Object type = 3  Mailbox

Mailbox type        xxxxxx     Task queue head     xxxx
Queue discipline    xxxx       Object queue head   0000
Containing job      xxxx       Object cache depth  xx
```

Figure 2-15. Format of VT Output: Mailbox with No Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx     Task queue head     zzzz
Queue discipline    xxxx       Object queue head   0000
Containing job      xxxx       Object cache depth  xx

Task queue          zzzz  xxxx  ...
```

Figure 2-16. Format of VT Output: Mailbox with Task Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head     xxxx
Queue discipline    xxxx        Object queue head   zzzz
Containing job      xxxx        Object cache depth  xx

Object cache queue      zzzz  xxxx  ...

Object overflow queue   xxxx  xxxx  ...
```

Figure 2-17. Format of VT Output: Mailbox with Object Queue

```
Object type = 3  Mailbox

Mailbox type        xxxxxx      Task queue head    zzzz
Queue discipline    xxxx        Data queue head    xxxx:xxxx
Containing job      xxxx

Data message queue  xxxx:xxxx   xxxx:xxxx   xxxx:xxxx
                    xxxx:xxxx   xxxx:xxxx   ...
```

Figure 2-18. Format of VT Output: Mailbox with Data Message Queue

The fields in Figures 2-15, 2-16, 2-17, and 2-18 are as follows:

Mailbox type — The type of mailbox: object or data. Mailbox type is either Object or Data. The mailbox type is defined when the mailbox is created.

Task queue head — The token for the task at the head of the queue. If the task queue for this mailbox is empty, this field contains the mailbox token.

Object queue head — The token for the object at the head of the queue. If the object queue for this mailbox is empty, this field contains "0000". If the mailbox type is "Data", this field contains "N/A".

Queue discipline	Indicates how tasks are queued at the mailbox. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the mailbox was defined when it was created with RQ$CREATE$MAILBOX. If the System Debugger can't interpret this field, it displays the actual value followed by a space and two question marks.
Object cache depth	The size of the high-performance cache portion of the object queue associated with the mailbox. This size was specified when the mailbox was created with RQ$CREATE$MAILBOX. If the mailbox type is "Data", this field contains "N/A".
Containing job	The token for the job that contains this mailbox.
Task queue	A list of tokens for the tasks queued at the mailbox in the order they are queued. If there are no tasks in the task queue, this field is not displayed.
Object cache queue	A list of tokens for the objects queued in the object cache queue, in the order they are queued. If there are no objects in the object cache queue or the mailbox type is Data, this field is not displayed.
Object overflow queue	A list of tokens for the objects queued in the object overflow queue, in the order they are queued. If there are no objects in the object overflow queue or the mailbox type is Data, this field is not displayed.
Data queue head	The pointer for the first data message at the head of the message queue.
Data message queue	Pointers for the data messages residing at the mailbox.

Semaphore Display

The System Debugger displays information about semaphores in two ways. The first display appears when no tasks are queued at the semaphore (Figure 2-19). The second appears when tasks are queued at the semaphore (Figure 2-20).

```
Object type = 4 Semaphore

Task queue head      xxxx    Queue discipline     xxxx
Current value        xxxx    Maximum value        xxxx
Containing job       xxxx
```

Figure 2-19. Format of VT Output: Semaphore with No Queue

```
Object type = 4 Semaphore

Task queue head      xxxx    Queue discipline     xxxx
Current value        xxxx    Maximum value        xxxx
Containing job       xxxx

Task queue       xxxx  xxxx  ...
```

Figure 2-20. Format of VT Output: Semaphore with Task Queue

The fields in Figures 2-19 and 2-20 are as follows:

Task queue head — The token for the task at the head of the queue. If the task queue is empty, this field contains zeros.

Queue discipline — Indicates how tasks are queued at the semaphore. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the semaphore was specified when it was created with RQ$CREATE$SEMAPHORE.

Current value — The number of units currently held by the semaphore.

Maximum value	The maximum number of units the semaphore can hold. This number was specified when the semaphore was created with RQ$CREATE$SEMAPHORE.
Containing job	The token for the job that the semaphore belongs to.
Task queue	A list of tokens for the tasks queued at the semaphore, in the order they are queued. If no tasks are queued, this list does not appear.

Region Display

If the parameter you supply is a valid token for a region, the System Debugger displays information about the associated region as shown in Figures 2-21 and 2-22.

```
Object type = 5 Region

Entered task        xxxx    Queue discipline        xxxx
Containing job      xxxx
```

Figure 2-21. Format of VT Output: Region with No Queue

```
Object type = 5 Region

Entered task        xxxx    Queue discipline        xxxx
Containing job      xxxx

Task queue      xxxx  xxxx  ...
```

Figure 2-22. Format of VT Output: Region with Task Queue

The fields in Figures 2-21 and 2-22 are as follows:

Entered task	The token for the task currently accessing information guarded by the region.
Queue discipline	Indicates how tasks are queued at the region. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the region was specified when it was created with RQ$CREATE$REGION.
Containing job	The token for the job that the region belongs to.
Task queue	Tokens for the tasks waiting to gain access to data guarded by the region. This line is displayed only if a task is already in the region and another task is waiting.

Segment Display

If the parameter that you supply is a valid token for a segment, the System Debugger displays information about the associated segment as shown in Figure 2-23.

```
Object type = 6 Segment

Segment size xxxx                Containing job      xxxx
```

Figure 2-23. Format of VT Output: Segment

The fields in Figure 2-23 are as follows:

Segment size	The number of bytes in this segment. The size of the segment was specified when the segment was created with RQ$CREATE$SEGMENT.
Containing job	The token for the job that the segment belongs to.

Extension Object Display

If the parameter that you supply is a valid token for an extension, the System Debugger displays information about the associated extension as shown in Figure 2-24.

```
Object type = 7 Extension

Extension type      xxxx    Deletion mailbox        xxxx
Containing job      xxxx
```

Figure 2-24. Format of VT Output: Extension Object

The fields in Figure 2-24 are as follows:

Extension type — The type code associated with composite objects licensed by this extension. This code was specified when the extension type was created with RQ$CREATE$EXTENSION. See the *iRMX® II Nucleus User's Guide* or the *iRMX® I Nucleus User's Guide* for more information about extension types.

Deletion mailbox — The token for the deletion mailbox associated with this extension. This mailbox was specified when the extension type was created with RQ$CREATE$EXTENSION.

Containing job — The token for the job that the extension belongs to.

Composite Object Display

The VT command displays the following kinds of composite information:

- All composites except those defined in the Basic I/O System (BIOS) and the port connection
- BIOS user objects
- BIOS physical file connections
- BIOS stream file connections
- BIOS named file connections
- BIOS remote file connections
- Port connection *(iRMX II only)*

Figure 2-25 shows the format for the display of non-BIOS objects.

```
Object type = 8 Composite

Extension type xxxx     Extension obj  xxxx     Deletion mbox    xxxx
Containing job xxxx     Num of entries xxxx

Component list   xxxx   xxxx    xxxx    xxxx     ...
```

Figure 2-25. Format of VT Output: Composite Object Other Than BIOS

The fields in Figure 2-25 are as follows:

Extension type	The code for the extension type of the extension object used to create this composite. This code was specified when the extension object was created with RQ$CREATE$EXTENSION.
Extension obj	The token for the extension object used to create this composite object.
Deletion mbox	The token for the mailbox to which this composite goes when the composite is to be deleted. This mailbox was specified when the extension was created with RQ$CREATE$EXTENSION.
Containing job	The token for the job that the composite object belongs to.
Num of entries	The number of component entries in the composite object.
Component list	The list of tokens for the components of the composite.

Figure 2-26 shows the format for the Basic I/O System user object display.

```
Object type = 8 Composite

Extension type  xxxx    Extension obj   xxxx    Deletion mbox    xxxx
Containing job  xxxx    Num of entries  xxxx

    BIOS USER OBJECT:
User segment  xxxx  Number of IDs    xxxx

User ID list    xxxx xxxx  ...
```

Figure 2-26. Format of VT Output: BIOS User Object Composite

Figure 2-26 uses the composite display described in Figure 2-25 as a base and appends the following fields:

User segment	The token for the segment containing the user IDs for the user object.
Number of IDs	The number of user IDs associated with the user object.
User ID list	List of the user IDs associated with the user object.

Figure 2-27 shows the format for a (file) connection to a physical file.

```
Object type = 8 Composite

Extension type  xxxx         Extension obj   xxxx         Deletion mbox   xxxx
Containing job  xxxx         Num of entries  xxxx

    T$CONNECTION OBJECT:
File driver     Physical     Conn flags      xx           Access          xxxx
Open mode       xxxxxx       Open share      xxxxxxx      File pointer    xxxxxxxx
IORS cache      xxxx         File node       xxxx         Device desc     xxxx
Dynamic DUIB    xxxxx        DUIB pointer    xxxx:xxxx    Num of conn     xxxx
Num of readers  xxxx         Num of writers  xxxx         File share      xxxxxxx
File drivers    xxxx         Device gran     xxxx         Device size     xxxxxxxx
Device functs   xxxx         Num dev conn    xxxx         Device name     xxxxxxxxxx
```

Figure 2-27. Format of VT Output: BIOS Physical File Connection

Figure 2-27 uses the composite display described in Figure 2-25 as a base and appends the following fields:

File driver	The BIOS file driver to which this connection is attached. The four possible values are Physical, Stream, Named, and Remote. If this field contains an invalid value, the System Debugger displays the value followed by a space and two question marks.

Conn flags

The flags for the connection. To determine how the flag is set, convert the hexadecimal value to binary. The following description shows the connection state when a bit (0 is the rightmost bit) is set to 1:

Bit	Condition
0	The connection is being detached
1	The connection is active and can be opened
2	This is a device connection
3	Reserved
4	The connection was forcibly detached
5-7	Reserved

Access

The access rights for this connection. This display uses a single character to represent each access right. If the connection has the access right, the character appears. If the connection does not have an access right, a hyphen (-) appears in the character position. The access rights and the characters that represent them are

```
                   ┌───────────────────── Delete
                   │     ┌─────────────── List
Directory files:   │     │     ┌───────── Add
                   │     │     │     ┌─── Change
                   D     L     A     C

                   D     R     A     U
                   │     │     │     └─── Update
    Data Files:    │     │     └───────── Append
                   │     └─────────────── Read
                   └───────────────────── Delete
```

Open mode

The mode established when this connection was opened. The possible modes are

Open Mode	Description
Closed	Connection is closed
Read	Connection is open for reading
Write	Connection is open for writing
R/W	Connection is open for reading and writing

	If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. If this value is Read, Write, or R/W, this value was specified when the connection was opened.
Open share	The sharing status established for this connection when it was opened. The sharing status for a connection is a subset of the sharing status of the file (see the File share field). The possible modes are

Share Mode	Description
Private	File cannot be shared
Readers	File can be shared with readers
Writers	File can be shared with writers
ALL	File can be shared with all users
0	Connection is not open

	If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. This probably indicates that the connection data structure has been corrupted.
File pointer	The current location of the file pointer for this connection.
IORS cache	The token for the segment at the head of the BIOS list of used IORSs. These IORSs are being saved for the RQ$WAIT$IO system call to use again. This list is empty if zeros appear in this field.
File node	The token for a segment that the operating system uses to maintain information about the connection. The information in this segment appears in the next two fields.
Device desc	The token for the segment that contains the device descriptor. The device descriptor is used by the operating system to maintain information about connections to a device.
Dynamic DUIB	Indicates whether a Device Unit Information Block (DUIB) was created dynamically when the device associated with this connection was attached.
DUIB pointer	The address of the DUIB for the device unit containing the file. See the *iRMX® Device Drivers User's Guide* for more information about DUIBs.
Num of conn	The number of connections to the file.
Num of readers	The number of connections now open for reading.
Num of writers	The number of connections now open for writing.

File share

The share mode of the file. This parameter defines how other connections to the file can be opened. The share mode of a file is a superset of the sharing status of each of the connections to the file (see the Open share field description). The possible modes are

Share Mode	Description
Private	File cannot be shared
Readers	File can be shared with readers
Writers	File can be shared with writers
All	File can be shared with all users

If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks. This probably means that the internal data structure for the file or the fnode for the file has been corrupted. See the *iRMX® Basic I/O System User's Guide* for more information about sharing modes for files and connections.

File drivers

The file drivers that connect the file. If the file can be connected to a given file driver, then the bit in the display is set to 1. Bit 0 is the rightmost bit.

Bit	Driver
0	Physical file
1	Stream file
2	Reserved
3	Named file
4	Remote file

Device gran

The granularity (in bytes) of the device. This is the minimum number of bytes that can be written to or read from the device in a single (physical) I/O operation.

Device size

The capacity (in bytes) of the device.

Device functs — Describes the functions supported by the device where this file is stored. Each bit in the low-order byte of the field corresponds to one of the possible device functions. If that bit is set to 1, then the corresponding function is supported by the device.

Bit	Function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE

Num dev conn — The number of connections to the device.

Device name — The 14-character (or fewer) name of the device where this file is stored.

Figure 2-28 shows the format for a (file) connection to a stream file.

```
Object type = 8 Composite

Extension type  xxxx    Extension obj    xxxx       Deletion mbox    xxxx
Containing job  xxxx    Num of entries   xxxx

    T$CONNECTION OBJECT:
File driver     Stream  Conn flags       xx         Access           xxxx
Open mode       xxxxxx  Open share       xxxxxx     File pointer     xxxxxxxx
IORS cache      xxxx    File node        xxxx       Device desc      xxxx
Dynamic DUIB    xxxxx   DUIB pointer     xxxx:xxxx  Num of conn      xxxx
Num of readers  xxxx    Num of writers   xxxx       File share       xxxxxxx
File drivers    xxxx    Device gran      xxxx       Device size      xxxx
Device functs   xxxx    Num dev conn     xxxx       Device name      Stream
Req queued      xxxx    Queued conn      xxxx       Open conn        xxxx
```

Figure 2-28. Format of VT Output: BIOS Stream File Connection

Figure 2-28 uses the physical display described in Figure 2-27 as a base and appends the following fields:

Req queued — The number of requests now queued at the stream file.

Queued conn — The number of connections now queued at the stream file.

Open conn — The number of connections to the stream file now open.

Figure 2-29 shows the format for a file connection to a named file.

```
Object type = 8 Composite

Extension type  xxxx    Extension obj   xxxx    Deletion mbox   xxxx
Containing job  xxxx    Num of entries  xxxx

        T$CONNECTION OBJECT:
File driver     Named         Conn flags       xx          Access          xxxx
Open mode       xxxxxx        Open share       xxxxxx      File pointer    xxxxxxxx
IORS cache      xxxx          File node        xxxx        Device desc     xxxx
Dynamic DUIB    xxxxx         DUIB pointer     xxxx:xxxx   Num of conn     xxxx
Num of readers  xxxx          Num of writers   xxxx        File share      xxxx
File drivers    xxxx          Device gran      xxxx        Device size     xxxxxxxx
Device functs   xxxx          Num dev conn     xxxx        Device name     xxxx
Num of buffers  xxxx          Fixed update     xxxx        Update timeout  xxxx
Fnode number    xxxx          File type        xxxxxxxxx   Fnode flags     xxxx
Owner           xxxxx         File/Vol gran    xxxx        Fnode PTR(s)    xxxx:xxxx
Total blocks    xxxxxxxx      Total size       xxxxxxxx    This size       xxxxxxxx
Volume gran     xxxx          Volume size      xxxxxxxx    Volume name     xxxxxx
```

Figure 2-29. Format of VT Output: BIOS Named File Connection

Figure 2-29 uses the physical display described in Figure 2-27 as a base and appends the following fields:

Num of buffers — The number of buffers allocated for blocking and unblocking I/O requests involving the device. A value of zero (0) indicates that the device is not a random-access device.

Fixed update — TRUE or FALSE indicates whether the device uses the fixed update timeout feature. For more information about update timeout, see the *iRMX® Basic I/O System User's Guide.*

Update timeout — The length of the time for the update timeout feature, measured in Nucleus time units. For more information about fixed updating, see the *iRMX® Basic I/O System User's Guide.*

Fnode number — The fnode number of this file. For more information about fnodes, see the *iRMX® Disk Verification Utility Reference Manual.*

File type	The type of named file. The possible values are

File type	Description
DIR	Directory file
DATA	Data file
SPACEMAP	Volume free space map file
FNODEMAP	Free fnodes map file
BADBLOCKMAP	Bad blocks file

If this field contains an invalid value, the System Debugger displays the value, followed by a space and two question marks.

Fnode flags — A word containing flag bits. If a bit is set to 1, the following description applies. Otherwise, the description does not apply. (Bit 0 is the rightmost bit.)

Bit	Description
0	This fnode is allocated
1	The file is a long file
2	Primary fnode
3-4	Reserved
5	This file has been modified
6	This file is marked for deletion
7-15	Reserved

Owner — The ID of the owner of the file. If this field has a value of 0FFFFH, then the field is displayed as "WORLD". See the *iRMX® Basic I/O System User's Guide* for more information about file ownership.

File/Vol gran — The granularity of the file (in volume granularity units).

Fnode PTR(s) — The addresses of the fnode pointers. See the *iRMX® Disk Verification Utility Reference Manual* for more information about fnode pointers.

Total blocks — The total number of volume blocks used for the file at present; this includes indirect blocks. See the *iRMX® Disk Verification Utility Reference Manual* for more information about blocks.

Total size — The total size (in bytes) of the file; this includes actual data only.

This size — The total number of bytes allocated to the file for data.

Volume gran — The granularity (in bytes) of the volume.

Volume size — The size (in bytes) of the volume.

Volume name — The name of the volume.

Figure 2-30 shows the format for a file connection to a remote file.

```
Object type = 8 Composite

Extension type  xxxx    Extension obj    xxxx     Deletion mbox    xxxx
Containing job  xxxx    Num of entries   xxxx

    T$CONNECTION OBJECT:
File driver     Remote  Conn flags       xx          Access          xxxx
Open mode       xxxxxx  Open share       xxxxxx      File pointer    xxxxxxxx
IORS cache      xxxx    File node        xxxx        Device desc     xxxx
Dynamic DUIB    xxxxx   DUIB pointer     xxxx:xxxx   Num of conn     xxxx
Num of readers  xxxx    Num of writers   xxxx        File share      xxxx
File drivers    xxxx    Device gran      xxxx        Device size     xxxxxxxx
Device functs   xxxx    Num dev conn     xxxx        Device name     xxxx
```

Figure 2-30. Format of VT Output: BIOS Remote File Connection

The fields in Figure 2-30 are the same as the fields in Figure 2-27, except for the File driver field, which is "Remote" rather than "Physical."

Figure 2-31 shows the display format for a port having signal protocol type.

```
Object type = 8   Composite

Extension type  xxxx   Extension obj   xxxx   Deletion mbox   xxxx
Containing job  xxxx   Num of entries  xxxx

    T$PORT OBJECT:
Protocol type   Signal  Queue discipline  xxxx  Signal count    xxxx
source id       xxxx

Task queue  xxxx   xxxx
```

Figure 2-31. Format of VT Output: Signal Protocol Port

Figure 2-31 uses the composite display described in Figure 2-23 as a base and appends the following fields:

Protocol type	The message protocol. This value is "Signal" to indicate signal service The type is determined when the port is created through RQ$CREATE$PORT.
Queue discipline	Indicates how tasks are queued at the port. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the port was specified when it was created with RQ$CREATE$PORT. If this field is uninterpretable, the actual BYTE value followed by a space and two question marks appears.
Signal count	The number of signals now waiting to be received at the port.
Source id	The board (agent) identification number for which this port was created to send messages to or receive messages from. This identification number matches the slot number of the remote board. The number is established through the "message$id" field when the port is created using the utility RQ$CREATE$PORT.
Task queue	The tokens for the list of tasks (if any) queued at the port.

Figure 2-32 shows the display format for a port having data transport protocol type.

```
Object type = 8 Composite

Extension type     xxxx    Extension obj        xxxx      Deletion mbox     xxxx
Containing job     xxxx    Num of entries       xxxx

    T$PORT OBJECT:
Protocol type      Data T  Queue discipline     xxxx      Buffer pool       xxxx
Fragmentation      xxx     Max Port Transctns   xxxx      Sink port         xxxx
Destination msg id xxxx    Destination port id  xxxx      Source port id    xxxx

Transaction id     xxxx    Task token           xxxx
Transaction id     xxxx    Message pointer      xxxx:xxxx

Message queue      xxxx:xxxx   xxxx:xxxx
```

Figure 2-32. Format of VT Output: Data Transport Protocol Port

```
Object type = 8 Composite

Extension type       xxxx    Extension obj        xxxx    Deletion mbox    xxxx
Containing job       xxxx    Num of entries       xxxx

    T$PORT OBJECT:
Protocol type        Data T  Queue discipline     xxxx    Buffer pool      xxxx
Fragmentation        xxx     Max Port Transctns   xxxx    Sink port        xxxx
Destination msg id   xxxx    Destination port id xxxx    Source port id   xxxx

Transaction id xxxx Task token  xxxx
Transaction id xxxx Message pointer xxxx:xxxx

Task queue     xxxx  xxxx
```

Figure 2-33. Format of VT Output: Data Transport Protocol Port

Figures 2-32 and 2-33 use the composite display described in Figure 2-23 as a base and append the following fields:

Protocol type	The message protocol. This value is "Data T" to indicate Data Transport service The type is determined when the port is created through RQ$CREATE$PORT.
Queue discipline	Indicates how tasks are queued at the port. Tasks are queued as "FIFO" (first-in-first-out) or by "PRI" (priority), depending on how the port was specified when it was created with RQ$CREATE$PORT.
Buffer pool	The token of the attached buffer pool (if any). The utility RQ$ATTACH$BUFFER$POOL attaches a buffer pool to a port.
Fragmentation	The fragmentation protocol. This value is either "Yes" if the port can handle message fragmentation, or "No" if the port does not handle message fragmentation. Port fragmentation protocol is defined through the utility RQ$CREATE$PORT.
Max Port Transctns	The maximum number of simultaneous outstanding transactions for the port. This limitation is defined when the port is created using RQ$CREATE$PORT.
Sink port	The token of the sink port (if any) associated with the port. Sink ports are connected to ports through the RQ$ATTACH$PORT utility.

Destination msg id	The host$id portion of the socket identifying the remote port that this port is connected. This value is established through the RQ$CONNECT utility.
Destination port id	The port$id portion of the socket identifying the remote port that this port is connected. This value is established through the RQ$CONNECT utility.
Source port id	The board (agent) identification number for which this port was created to send messages to or receive messages from. The number is established through the "port$id" field when the port is created using the utility RQ$CREATE$PORT.
Transaction id	Outstanding transaction identification numbers at this port.
Task token	The token(s) of the task or tasks with outstanding transactions at this port.
Message pointer	The pointer of the message(s) with outstanding transactions at this port.
Message queue	The list of pointers representing the messages queued at this port. This field appears only if the port has queued messages.

NOTE

Besides the display forms shown in Figures 2-32 and 2-33, the VT output for a Data Transport protocol port can appear with the following combinations of fields:

- Transaction information with no Message Queue or Task Queue information
- Message Queue information with no Transaction or Task Queue information
- Task Queue information with no Transaction or Message Queue information
- No Transaction, Message Queue, or Task Queue information

Buffer Pool Display

If the parameter that you supply is a valid token for a buffer pool, the System Debugger displays information about the buffer pool as shown in Figure 2-34.

```
Object type = 10    Buffer pool

Max Buffers xxxx    Total buffer count  xxxx    Total size count  xxxx
Containing job  xxxx    Data Chaining       xxx

Buffer pool contents:

Buffer size xxxx    Buffer count    xxxx
Buffer size xxxx    Buffer count    xxxx
                .
                .
                .
```

Figure 2-34. Format of VT Output: Buffer Pool

Figure 2-34 display fields are defined as follows:

Max buffers	The total number of buffers allowed in this buffer pool. This maximum value is determined when the buffer pool is created using RQ$CREATE$BUFFER$POOL.
Total buffer count	The number of buffers now in the buffer pool. This number is equivalent to the number of buffers created in the pool using RQ$CREATE$SEGMENT.
Total size count	The number of different buffer sizes in the buffer pool. The maximum number of different buffer sizes is eight.
Containing job	The token for the job that created this buffer pool.
Data Chaining	YES or NO indicates whether this buffer pool supports data chaining.
Buffer size	The available buffer sizes for this buffer pool. These sizes are determined when the individual buffers are created through RQ$CREATE$SEGMENT.
Buffer count	The number of buffers that are of the buffer size displayed in the field directly to the left.

The VU command displays (unwinds) the iRMX system calls in the stack of the task having the token you enter.

W-0954

Parameter

token — The token for the task having the stack to be searched for system calls.

Description

The VU command accepts a token for a task and then searches the task's stack for iRMX system calls, starting at the top of the stack. For each system call it finds in the stack, it displays

- The return address for the call. This is the address of the next instruction to be executed for the task after the system call has finished running.
- The VS display with two lines of stack values (or more if required for parameters of the system call). They are shown as if the CALL instruction for the system call were in the CS:IP register and the displayed stack values were at the top of the stack.

This command requires the task stack to be inside an iRMX segment.

The VU command uses internal iRMX data structures to get some of its information. The data structures are updated immediately after the system call at the top of the task's stack runs to completion. Since the monitor interrupt might come after the system call is completed, but before the data structures are updated, some of the information the VU command uses may be obsolete. Therefore, the first system call the VU command displays may not be valid.

Figure 2-35 illustrates the format of one system call display by the VU command. System calls can be nested, with one calling another, so some invocations of the VU command produce multiple displays of the type shown in the figure.

If the stack of the indicated task has no system calls, the VU command displays the following message:

```
No system calls on stack
```

```
gate #NNNN

Return cs:ip - yyyy:yyyy
xxxx:xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx
xxxx:xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx    xxxx

(subsystem)  system call

                     |parameters|
```

Figure 2-35. Format of VU Output

The fields in Figure 2-34 are as follows:

gate #NNNN	The gate number associated with the system call.
Return cs:ip	The return address for the system call of this display (yyyy:yyyy).
xxxx:xxxx	The address of the stack portion devoted to this call.
xxxx	Values now on the stack.
(subsystem)	The iRMX Operating System layer containing the system call.
system call	The name of the iRMX system call.
parameters	The parameter names associated with the stack values. The parameters correspond to the stack values directly above them. If one of the parameters is a string, it displays the string contents below the parameters.

Error Messages

Syntax Error	An error was made in entering the command.
*** INVALID TASK TOKEN ***	The value entered for the token is not a valid task token.
Stack not an iRMX segment	The stack of the task is not an iRMX segment, as is required.
TOKEN is not a TASK	The value entered for the token is valid; however, it is not a task token.

Example

This example shows how the VU command responds when system calls are nested. The task for the example has called RQSWRITE$MOVE of the Extended I/O System. RQSWRITE$MOVE has called RQ$A$WRITE of the Basic I/O System. RQAWRITE has called RQ$RECEIVE$MESSAGE to wait for the data transfer to be completed.

Suppose that before the message arrives signaling the completion of the transfer, you enter the System Debugger and invoke the following VU command:

```
..VU 21C8   <CR>
```

The System Debugger responds by displaying the following:

```
gate #0430

Return cs:ip -09B8:576A
216A:01B2     01C8   216A   01C8   216A   FFFF   1768   1760   1988
216A:01C2     1550   0000   2148   1FF8   1440   2558   2000   2050

(Nucleus) receive message

              |...excep$p....|....resp$p...|.time.|.mbox.|

gate #05B0

Return cs:ip -09D8:08E7
216A:01D4     01E8   216A   1F58   0400   0000   20E8   2098   2088
216A:01E4     1430   2048   01F8   20F8   1400   0218   0000   01F8

(BIOS) write

              |...excep$p...|..mbox.|.count|...buffer$p..|.conn.|

gate #0710

Return cs:ip -09F8:06FA
216A:0218     0020   19F0   0400   0030   19F0   2098   2080   2140
216A:0228     2058   0000   0000   20C8   20C8   20C8   20C8   20C8

(EIOS) write move

              |...excep$p...|..count|...buffer$p...|.conn.|
```

SAMPLE DEBUG SESSION 3

3.1 INTRODUCTION

This chapter provides a sample PL/M-286 program that was developed on an Intel 310 system running on an iSBC® 286/10 processor board with the iRMX II.4 Operating System. The terminal was a Hazeltine 1510. The code has compiled without errors; however, it does not run. The step-by-step process for using iSDM monitor and System Debugger commands to locate and fix the bug, then to test the corrected code is described in section 3.2. A scenario examining debugging techniques and additional commands is provided in section 3.3.

3.2 SAMPLE PROGRAM

This program includes three tasks.

- An initialization task (called Init) creates a mailbox and the other two tasks.
- Two tasks (called Alphonse and Gaston) exchange messages via mailboxes.

The source code is listed in Figures 3-1, 3-2, and 3-3. For information on compiling and binding this code, see the *iRMX® II Programming Techniques Reference Manual* or the *iRMX® I Programming Techniques Reference Manual.* The following description explains how the program is supposed to work.

The application code runs as a Human Interface (HI) program; therefore, the <name of the OBJECT file specified in BND286> is entered at the HI prompt. The task called Init runs first, creating a mailbox it catalogs in the root directory under the name "master." It creates the tasks Alphonse and Gaston then suspends itself.

When Gaston receives control, it gets the token for the mailbox created by Init (by looking up the name "master" in the root job's object directory). It then creates a segment (in which it will place a message) and a response mailbox (to which Alphonse will send a reply). Next it goes into a loop in which it places a message in the segment (after displaying it on the screen), sends the segment to the master mailbox, then waits at the response mailbox for a reply.

When Alphonse receives control, it also gets the token for the mailbox created by Init (by looking up the name in the root job's object directory). It then goes into a loop in which it waits at the mailbox for a message and checks to see if the token it received is a segment. If it is a segment, Alphonse places its own message in the segment (after displaying it on the screen), then sends the segment to the response mailbox. If it isn't a segment, Alphonse drops out of the loop and deletes itself.

By using the two mailboxes, the tasks Alphonse and Gaston are synchronized. Gaston sends a message to the first mailbox and waits at the second one before continuing. Alphonse waits at the first mailbox. When it receives a message, it sends a reply to the second mailbox and waits at the first for another message. This cycle continues for 6 messages.

After sending its sixth message, Gaston drops out of the loop. Instead of sending a segment to the master mailbox, Gaston displays a final message to the screen then sends the task token (the token for the Init task) to the mailbox. When Alphonse receives this token and finds it is not a segment, Alphonse drops out of its loop and deletes itself.

To finish the processing, Gaston causes the Init task to resume processing (remember, the Init task suspended itself earlier). When Init takes over, it deletes both offspring tasks and issues an EXITIOJOB system call to return control to the Human Interface level.

```
compact
init: DO;

DECLARE token                    LITERALLY          'SELECTOR';
DECLARE fifo                     LITERALLY          '0';
DECLARE self                     LITERALLY          '0';
DECLARE task$priority            BYTE;
DECLARE calling$task             TOKEN;
DECLARE calling$tasks$job        TOKEN;
DECLARE master$mbox              TOKEN;
DECLARE status                   WORD;
DECLARE init$task$token          TOKEN;
DECLARE gaston$task$token        TOKEN;
DECLARE alphonse$task$token      TOKEN;
DECLARE alphonse$start$add       POINTER;
DECLARE gaston$start$add         POINTER;
DECLARE gaston$ds                WORD EXTERNAL;
DECLARE alphonse$ds              WORD EXTERNAL;
DECLARE stack$pointer            POINTER;
DECLARE stack$size               WORD;
DECLARE task$flags               WORD;

gaston:
    PROCEDURE EXTERNAL;
END gaston;

alphonse:
    PROCEDURE EXTERNAL;
END alphonse;
```

Figure 3-1. Example PL/M-286 Application (Init)

```
$include(:rmx:inc/nuclus.ext)
$include(:rmx:inc/eios.ext)

    calling$tasks$job = SELECTOR$OF(NIL);   /* Directory obj cataloged in */
    calling$task = SELECTOR$OF(NIL);        /* Task whose priority will   */
                                            /* be gotten */
    gaston$start$add = @gaston;             /* Set up start addresses for */
    alphonse$start$add = @alphonse;         /* tasks */
    stack$pointer = NIL;                    /* Values for creating tasks  */
    stack$size = 500;
    task$flags = 0;

    init$task$token = RQ$GET$TASK$TOKENS(   /* Get token for init task    */
        self,
        @status);

    CALL RQ$CATALOG$OBJECT (                /* Catalog task token in      */
        calling$tasks$job,                  /* directory of calling       */
        init$task$token,                    /* task's job */
        @(4,'init'),
        @status);

    master$mbox = RQ$CREATE$MAILBOX (       /* Create mailbox tasks use   */
        fifo,                               /* to pass messages           */
        @status);

    CALL RQ$CATALOG$OBJECT (                /* Catalog mailbox in         */
        calling$tasks$job,                  /* directory of calling       */
        master$mbox,                        /* task's job                 */
        @(6,'master'),
        @status);

    task$priority = RQ$GET$PRIORITY (       /* Get priority of calling    */
        calling$task,                       /* task */
        @status);

    task$priority = task$priority + 1;      /* Pick lower priority for    */
                                            /* new tasks                  */
```

Figure 3-1. Example PL/M-286 Application (Init) (continued)

```
        alphonse$task$token = RQ$CREATE$TASK (  /* Create tasks             */
        task$priority,
        alphonse$start$add,
        SELECTOR$OF(@alphonse$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    gaston$task$token = RQ$CREATE$TASK (
        task$priority,
        gaston$start$add,
        SELECTOR$OF(@gaston$ds),
        stack$pointer,
        stack$size,
        task$flags,
        @status);

    CALL RQ$SUSPEND$TASK (                  /* Suspend self and let other */
        calling$task,                       /* tasks run                  */
        @status);

    CALL RQ$DELETE$TASK (                   /* Clean up and exit          */
        gaston$task$token,
        @status);

    CALL RQ$DELETE$TASK (
        alphonse$task$token,
        @status);

    CALL RQ$EXIT$IO$JOB (
        0,
        NIL,
        @status);

END;                                        /* Init                       */
```

Figure 3-1. Example PL/M-286 Application (Init) (continued)

```
$compact
alphonse$code: DO;

DECLARE token                       LITERALLY          'SELECTOR';

$include(:rmx:inc/nuclus.ext)
$include(:rmx:inc/eios.ext)
$include(:rmx:inc/hi.ext)

alphonse:
PROCEDURE PUBLIC;

DECLARE CR                          LITERALLY          '13';
DECLARE LF                          LITERALLY          '10';
DECLARE wait$forever                LITERALLY          '0FFFFH';
DECLARE FOREVER                     LITERALLY          'WHILE 1';
DECLARE calling$tasks$job           TOKEN;
DECLARE master$mbox                 TOKEN;
DECLARE response$mbox               TOKEN;
DECLARE status                      WORD;
DECLARE type$code                   WORD;
DECLARE time$limit                  WORD;
DECLARE count                       WORD;
DECLARE alphonse$ds                 WORD PUBLIC;
DECLARE seg$token                   TOKEN;
DECLARE seg$size                    WORD;
DECLARE display$message(*)          BYTE               DATA(
    CR,LF, 'After you, Gaston', CR, LF);

DECLARE message BASED seg$token     STRUCTURE(
                                        count          BYTE,
                                        text(25)       BYTE);

    time$limit = 25;                     /* Delay factor for message   */
                                         /* display                    */
    seg$size = 32;                       /* Size of message segment    */
    calling$tasks$job = SELECTOR$OF(NIL); /* Directory in which to look */
                                         /* up obj                     */
```

Figure 3-2. Example PL/M-286 Application (Alphonse)

```
    master$mbox = RQ$LOOKUP$OBJECT (            /* Look up message     */
        calling$tasks$job,                      /* mailbox             */
        @(6,'master'),
        wait$forever,
        @status);

    DO FOREVER;

        seg$token = RQ$RECEIVE$MESSAGE (        /* Receive response    */
            master$mbox,                        /* from Gaston         */
            wait$forever,
            @response$mbox,
            @status);

        type$code = RQ$GET$TYPE(                /* See what kind of    */
            seg$token,                          /* object it is        */
            @status);

        IF type$code <> 6 THEN                  /* If it isn't a       */
            CALL RQ$EXIT$IO$JOB (               /* segment, exit       */
                0,
                NIL,
                @status);

        message.count = 21;
        CALL MOVB(@display$message, @message.text, size(display$message));

        CALL RQ$C$SEND$CO$RESPONSE (            /* Send message to     */
            NIL,                                /*screen               */
            0,
            @message.count,
            @status);

        CALL RQ$SLEEP(                          /* Wait a while to     */
            time$limit,                         /* give user time to   */
            @status);                           /* see the message     */

        CALL RQ$SEND$MESSAGE (                  /* Send message to     */
            response$mbox,                      /* response mailbox    */
            seg$token,
            SELECTOR$OF(NIL),
            @status);

        END;                                    /* FOREVER             */
    END alphonse;                               /* Alphonse            */
END alphonse$code;
```

Figure 3-2. Example PL/M-286 Application (Alphonse) (continued)

```
$compact
gaston$code: DO;

DECLARE token                    LITERALLY         'SELECTOR';

$include(:rmx:inc/nuclus.ext)
$include(:rmx:inc/eios.ext)
$include(:rmx:inc/hi.ext)

gaston:
PROCEDURE PUBLIC;

DECLARE CR                       LITERALLY         '13';
DECLARE LF                       LITERALLY         '10';
DECLARE fifo                     LITERALLY         '0';
DECLARE wait$forever             LITERALLY         '0FFFFH';
DECLARE parent$task              TOKEN;
DECLARE calling$tasks$job        TOKEN;
DECLARE master$mbox              TOKEN;
DECLARE response$mbox            TOKEN;
DECLARE status                   WORD;
DECLARE time$limit               WORD;
DECLARE count                    WORD;
DECLARE final$count              WORD;
DECLARE gaston$ds                WORD PUBLIC;
DECLARE seg$token                TOKEN;
DECLARE seg$size                 WORD;
DECLARE main$message(*)          BYTE              DATA(
    CR,LF, 'After you, Alphonse', CR, LF);

DECLARE final$message(*)         BYTE              DATA(
    CR,LF, 'If you insist, Alphonse', CR, LF);

DECLARE message BASED seg$token  STRUCTURE(
                                     count         BYTE,
                                     text(27)      BYTE);

    count = 0;                             /* Initialize count          */
    final$count = 6;                       /* Set number of loops       */
    time$limit = 25;                       /* Delay factor for display  */
                                           /* to screen                 */
    seg$size = 32;                         /* Size of message segment   */
    calling$tasks$job = SELECTOR$OF(NIL);  /* Directory in which to look */
                                           /* up object                 */
```

Figure 3-3. Example PL/M-286 Application (Gaston)

```
master$mbox = RQ$LOOKUP$OBJECT (         /* Look up message mailbox   */
    calling$tasks$job,
    @(6,'master'),
    wait$forever,
    @status);

response$mbox = RQ$CREATE$MAILBOX (      /* Create response mailbox   */
    fifo,
    @status);

seg$token = RQ$CREATE$SEGMENT(           /* Create message segment    */
    seg$size,
    @status);

DO WHILE count < final$count;
    message.count = 23;
    CALL MOVW(@main$message, @message.text, SIZE(main$message));

    CALL RQ$C$SEND$CO$RESPONSE (         /* Send message to screen    */
        NIL,
        0,
        @message.count,
        @status);

    CALL RQ$SLEEP(                       /* Wait a while to give user */
        time$limit,                      /* time to see the message   */
        @status);

    CALL RQ$SEND$MESSAGE (               /* Send message to mailbox   */
        master$mbox,
        seg$token,
        response$mbox,
        @status);

    seg$token = RQ$RECEIVE$MESSAGE(      /* Receive response from     */
        response$mbox,                   /* Alphonse                  */
        wait$forever,
        NIL,
        @status);

    count = count + 1;
END;                                     /* WHILE                     */

message.count = 27;
CALL MOVB(@final$message,@message.text,SIZE(final$message));
```

Figure 3-3. Example PL/M-286 Application (Gaston) (continued)

```
        CALL RQ$C$SEND$CO$RESPONSE (          /* Send final message to  */
        NIL,                                  /* screen                 */
        0,
        @message.count,
        @status);

    CALL RQ$SEND$MESSAGE (                    /* Send token for mailbox */
        master$mbox,                          /* to mailbox.  This will */
        master$mbox,                          /* stop other task.       */
        SELECTOR$OF(NIL),
        @status);

    parent$task = RQ$LOOKUP$OBJECT(           /* Look up token for      */
        calling$tasks$job,                    /* calling task           */
        @(4,'init'),
        wait$forever,
        @status);

    CALL RQ$RESUME$TASK(                      /* Resume calling task    */
        parent$task,                          /* for cleanup            */
        @status);

    END gaston;                               /* Gaston                 */
END gaston$code;
```

Figure 3-3. Example PL/M-286 Application (Gaston) (continued)

3.3 DEBUGGING THE PROGRAM

Although it's a good idea to include error checking when developing programs, we did not include any in our sample program so we could demonstrate more features of the System Debugger. The sample program contains one error. We will show two approaches to finding and correcting it using the System Debugger.

The addresses and token values appearing in the following examples are those the system assigned in this debugging session. Most of these values will change from session to session. It's helpful to keep paper and pencil handy to note the various addresses and tokens.

When the iSDM monitor is invoked, both the application code and the operating system code freeze. However, by using iSDM monitor and System Debugger commands you can disassemble and execute the application instructions. Thus, in a debugging session you will move the CS:IP through your code, examining system objects, possibly changing stack or register values. These changes are valid for only one pass through the code. To re-execute the code, kill the current job by using the CLI-restart feature, then re-enter the iSDM monitor by using the Human Interface DEBUG command.

Example #1:

When <name of OBJECT file specified in BND286> runs, the system displays the following message:

```
Interrupt 13 at 2C38:0199 General Protection ECODE=0000
..
```

The values 2C38:0199 are where the CS:IP was pointing when the program halted. The protected-mode prompt (..) indicates that we are in the iSDM monitor. However, since the program has been executed, we must re-enter the iSDM monitor to re-execute the code. We can use the CLI-restart feature to return to the Command Line Interpreter. Enter the following command:

```
..g 284:14  <CR>
```

The system responds with the Human Interface prompt (-). Next, enter the following command:

```
-Debug <name of OBJECT file specified in BND286>  <CR>
```

The system responds with the following:

```
Interrupt 3 at 2A70:FFFF
..
```

Use the iSDM monitor's GO (G) command to set a breakpoint at the instruction where the program halted (remember the CS:IP value is given in the interrupt message displayed when the program halts). The code segment (CS) value will change each time you re-enter the iSDM monitor, but the instruction pointer (IP) will remain the same. Enter the following command:

```
..g,199  <CR>
```

To find out where we are in the code, use the iSDM monitor's D (DISPLAY MEMORY/DESCRIPTOR TABLES) command to display a disassembled block of code. Enter the following command:

```
..10 dx, <CR>
```

The system displays the following code:

```
2500:0199   F2A5       REP     MOVSW
2500:019B   B80000     MOV     AX,0000
2500:019E   8BD0       MOV     DX,AX
2500:01A0   52         PUSH    DX
2500:01A1   50         PUSH    AX
2500:01A2   680000     PUSH    0000
2500:01A5   8E063E00   MOV     ES,[003E]
2500:01A9   B80000     MOV     AX,0000
2500:01AC   06         PUSH    ES
2500:01AD   50         PUSH    AX -
```

The instruction at address 2500:0199 is a MOVE STRING WORD command. The only move word instruction in the sample program is the PL/M-286 MOVW call when Gaston enters the loop after creating the segment. The following display shows this section of code:

```
response$mbox = RQ$CREATE$MAILBOX  (     /* Create response mailbox */
    fifo,
    @status;

seg$token = RQ$CREATE$SEGMENT(           /* Create message segment */
    seg$size,
    @status;

DO WHILE count < final$count;
    message.count, = 23;

    CALL MOVW(@main$message, @message.text,   SIZE(main$message));

    CALL RQ$C$SEND$CO$RESPONSE (         /* Send message to screen */
        NIL,
        0,
        @message.count,
        @status);
```

Figure 3-4. MOVW in Gaston Code

If displaying the instruction doesn't provide enough information about why the program halted, we can look at the surrounding code by displaying forward or backward from the CS:IP. The comma we specified in the DX command enables us to enter just a comma (,) now to display forward another ten instructions from the current CS:IP. (Displaying backward from the CS:IP is shown in Example #2.) To see the register contents, enter the following command:

```
..x  <CR>
```

The system displays the following:

```
AX=0000   CS=2500   IP=0199    FL=0293   RGDT .BASE=002000 .LIMIT=2FFF
BX=0034   SS=2638   SP=01F2    BP=01F2   RIDT .BASE=005000 .LIMIT=03FF
CX=0017   DS=2530   SI=0042   MSW=FFFB
DX=2680   ES=2680   DI=0001    TR=0278   RLDT=02A0
..
```

To execute the MOVSW instruction, enter the following command:

`..n,    <CR>`

The system displays the following:

```
2500:0199           F2A5            REP MOVSW                   -
```

Enter a comma (,).

The system responds with the following:

```
Interrupt 13 at 2500:0199 General Protection ECODE=0000
..
```

To see how executing this instruction changed register contents, enter the following command:

`..x   <CR>`

The system displays the following:

```
AX=0000   CS=2800   IP=0199    FL=0293   RGDT .BASE=002000 .LIMIT=2FFF
BX=0034   SS=26D8   SP=01F2    BP=01F2   RIDT .BASE=005000 .LIMIT=03FF
CX=0006   DS=28B8   SI=0062   MSW=FFFB
DX=26C0   ES=26C0   DI=0021    TR=0278   RLDT=02A0
..
```

In the ASM286 Assembly language MOVSW instruction, DS:SI represents the source data is moving from; ES:DI is the destination. (For more information on MOVSW, see the *ASM286 Assembly Language Reference Manual.*) To check the limit of the ES register, enter the following command:

```
..ddt(es)  <CR>
```

The system displays the following:

```
GDT (1427T) DSEG BASE=090484 LIMIT=001F P=1 DPL=0 ED=0 W=1 A=1
SR=0000(ES)
```

The LIMIT parameter shows that the segment limit is 1FH (31 decimal). Since the system counts from zero, the limit is 32 decimal which is the value assigned to seg$size in Gaston. The DI register (shown in the previous display) contains 21H (33 decimal), indicating the system was trying to write past the segment limit when the program halted. This fact suggests the PL/M-286 MOVW call should be changed to MOVB. Here we could exit the iSDM monitor, change the PL/M-286 code, then recompile and run it.

However, we can use the iSDM monitor's EXAMINE/MODIFY REGISTERS (X) command to change a register value and the GO (G) command to execute the program. Making changes with the X and S (SUBSTITUTE MEMORY) commands enables us to test code without having to recompile and bind it.

The CX register contains the count of bytes or words moved. If we decrease the count in the CX register to 15 before we execute the MOVSW instruction, we should be able to move all the data. Re-enter the iSDM monitor and set a breakpoint at the MOVSW instruction by entering the following commands:

```
..g 284:14  <CR>
-debug <name of OBJECT file specified in BND286>  <CR>
..g,199  <CR>
```

Set the CX register to 15. Enter the following command:

```
..x cx=f  <CR>
```

Now, execute the rest of the program by entering the following command:

```
..g  <CR>
```

The system responds with the following:

```
After you, Alphonse

After you, Gaston

Interrupt 13 at 2A70:0199 General Protection ECODE=0000
..
```

Since our change was valid for one pass through the code, the first pass through the Gaston loop worked. The next pass failed. To return to the Command Line Interpreter, enter the following command:

```
..g 284:14  <CR>
```

This partially successful run shows that if we reduce the number of words moved, the program works. Therefore, to make a permanent fix, we should change the PL/M-286 MOVW call to MOVB in the sample code, then recompile and bind it.

Example #2:

We can also make changes in the disassembled code. Suppose we have run the program for the first time, and the system displayed the following message:

```
Interrupt 13 at 2A70:0199 General Protection ECODE=0000
..
```

Restart the system using the CLI-restart feature as you did in Example #1, then re-enter the iSDM monitor by entering the following command:

```
-Debug <name of OBJECT file specified in BND286>  <CR>
```

Set a breakpoint at the instruction that was executing when the program failed and display a block of disassembled code by entering the following commands:

```
..g,199  <CR>
..5 dx  <CR>
```

The system displays the following:

```
1258:0199   F2A5     REP     MOVSW
1258:019B   B80000   MOV     AX,0000
1258:019E   8BD0     MOV     DX,AX
1258:01A0   52       PUSH    DX
1258:01A1   50       PUSH    AX
```

To look at the instructions preceding MOVSW, enter the following command:

```
..15 dx cs:ip - 25  <CR>
```

The system displays the following code:

```
1258:0174   8B063800     MOV     AX,[0038]
1258:0178   3B063A00     CMP     AX,[003A]
1258:017C   7203         JB      A=0181
1258:017E   E97600       JMP     A=01F7
1258:0181   B117         MOV     CL,17
1258:0183   8E063E00     MOV     ES,[003E]
1258:0187   26880E0000   MOV     ES:[0000],CL
1258:018C   B500         MOV     CH,00
1258:018E   8E063E00     MOV     ES,[003E]
1258:0192   BF0100       MOV     DI,0001
1258:0195   BE4200       MOV     SI,0042
1258:0198   FC           CLD
1258:0199   F2A5         REP     MOVSW
1258:019B   B80000       MOV     AX,0000
1258:019E   8BD0         MOV     DX,AX
```

MOVSW is a repetitive move from DS:SI to ES:DI. Looking at the preceding instructions, we see 1258:0181 moves 17H into CL, which is the low-order register of CX. Remember that CX is the count of bytes or words moved. (For more information on the register set, see the *ASM286 Assembly Language Reference Manual*). If we display the ES register contents using "ddt(es) <CR>" as we did in the last example, we can check the limit. Since the limit is 32 (decimal) and the system is trying to write 17H words, the system fails when it tries to write past the segment limit. If we reduce this count we should be able to move the data. We must re-enter the iSDM monitor, then using the iSDM monitor's SUBSTITUTE (S) command, we can change the code at 1258:0181. Semicolons (;) precede the explanations in the following code; enter the information appearing in blue:

```
..g 284:14  <CR>
-Debug <name of OBJECT file specified in BND286>  <CR>
..s cs:181  <CR>                  ;enter monitor command to substitute
                                  ;memory at Ip=0181
1258:0181 B1 -  ,                 ;enter a comma to step to the count
1258:0182 17 -  f  <CR>           ;enter the new count
..g  <CR>                         ;re-start code execution
```

The system responds with six iterations of the following:

```
After you, Alphonse
After you, Gaston
.
.
.
```

After six iterations of the previous screen, the monitor displays the following:

```
If you insist, Alphonse
-
```

3.4 VIEWING SYSTEM OBJECTS

Consider that we have a deadlock problem. By looking at system objects at various stages of execution, we can observe how synchronization (or lack of it) is occurring.

We can view any object in a job using the VO command (specifying the job's token) to provide the broad picture of the system state, then the VT command to focus on individual elements. Suppose, we want to view the state of the objects before entering the loop in which Gaston and Alphonse exchange messages. Assume we have stepped through the code, verifying system calls until we located the CS:IP for the Nucleus create$segment system call in Gaston. Re-enter the iSDM monitor and set a breakpoint at this CS:IP by entering the following commands:

```
-Debug <name of OBJECT file specified in BND286>  <CR>
..g,16d  <CR>
```

To get the job token, enter the following command:

```
..vj  <CR>
```

The system displays the following:

```
iRMX®  <I/II>  Job Tree

0258
    0F38
        1670
            2460
    0E88
    0E00
```

Note that "2460" is the token for the application job. To view objects for this job, enter the following command:

```
..vo 2460  <CR>
```

The system displays the following:

```
Child Jobs:
Tasks:          26D0    26F0    1AC8    1900
Mailboxes:      25C0 t  1AB8
Semaphores:
Regions:
Segments:       25B0    25E8    25E0    2650    2528    2480   2478
Extensions:
Composites:     24A0

..
```

At this stage of program execution, two mailboxes exist. The "t" following mailbox 25C0 means one or more tasks are waiting at it (Alphonse was created first and is waiting for a message from Gaston). Examine mailbox 25C0 by entering the following command:

```
..vt 25C0  <CR>
```

The system responds with the following:

```
Object type = 3     Mailbox

Task queue head     1900    Object queue head       0000
Queue discipline    FIFO    Object cache depth      08
Containing job      2460

Task queue          1900
..
```

Use the System Debugger's VU command to view the waiting task's stack. To unwind the stack, enter the following command:

```
..vu 1900  <CR>
```

The system displays the following:

```
gate #0430

Return cs:ip -   1D18:029F
16C8:01E6        0086    1D28    0084    1D28    FFFF    17E0    0000

(Nucleus)receive message

                 |...excep$p...|....resp$p.....|.time.|.mbox.|

..
```

We can continue to examine objects or set a breakpoint at the return CS:IP. Setting the CS:IP (g, 29f <CR>) in the sample program causes the iSDM monitor to display the following:

```
Interrupt 13 at 21F0:0199 General Protection ECODE=0000
```

This message indicates that the program halts in Gaston and that 21F0:0199 is the instruction executing when it dies.

This chapter has shown two ways to find an error and two ways to make temporary fixes from the System Debugger. The message displayed when the program halts contains the CS:IP of the last instruction executing. If setting the CS:IP at this instruction and displaying the surrounding code doesn't give you enough information about where this point is in your application code, you can use combinations of VJ, VO, VT, VU, and VS to locate the running task. Then set the breakpoint at the CS:IP of the last executing instruction and display code, objects, and registers to determine how the system is executing that instruction.

iSDM™ MONITOR COMMANDS A

A.1 INTRODUCTION

This appendix briefly describes the iSDM System Debug Monitor commands in alphabetical order. A command directory listing the functional groups and page references precedes the command descriptions. For examples and more detailed information about the commands, see the *iSDM™ System Debug Monitor User's Guide*.

A.2 COMMAND DIRECTORY

This section provides a brief summary of all iSDM monitor commands listed by functions. Each entry in the following summary contains along with the command name a brief description of the command and a page reference where you can find more information on the command.

* *Command requires an attached development system.*

A.3 COMMAND DESCRIPTIONS

This section provides brief descriptions for iSDM monitor commands in an easily referenced alphabetical order. For more information on command parameters, syntax, and options, refer to the *iSDM™ System Debug Monitor User's Guide*.

A.3.1 B--Bootstrap Load

The B command passes control to the bootstrap loader to load absolute object code from secondary storage into your target system memory. The Bootstrap Loader loads the file into the target system at the memory address specified in the file. After the bootstrap loader finishes loading the file, the code begins executing. To use the B command correctly, you must be operating in real mode.

If either the file you specified or the default file does not exist, the bootstrap loader halts and takes action according to how it is configured.

A.3.2 C--Compare

The C command compares the contents of one block of memory defined by a range with the contents of another block of memory that begins at a destination address. The iSDM monitor expects the blocks to be equal in length. If the iSDM monitor encounters any mismatched bytes, it displays them in the following format:

```
aaaa:bbbb   xx   yy   aaaa:bbbb
```

where "aaaa:bbbb" are the addresses of the bytes that do not match and "xx" and "yy" are the bytes themselves.

A.3.3 D--Display Memory/Descriptor Tables/Disassembled Instructions

The D command is actually three commands in one. You can use it to display the contents of a specified block of memory, the contents of an 80286/386 descriptor table, or the contents of a specified block of memory in disassembled form. If you are operating in real mode, you cannot display descriptor table entries. However, if you are operating in protected mode, you can use both functions of this command.

A.3.4 E--Exit

The E command enables you to exit the loader program by returning control from the loader program to the development operating system. Upon return, the iSDM monitor loses all symbol information.

When using the E command, you must use it on a line by itself; do not use multiple commands on a line with the E command. Also, your system must include an attached development system before you can use this command.

When you reinvoke the iSDM monitor after exiting the loader program, one of two things happens:

- The iSDM monitor prints either a single or double prompt depending upon whether you were operating in real or protected mode when you exited.
- The iSDM monitor prints its usual sign-on message and re-initializes itself if you reset your target system between the time you exited the loader and the time you reinvoked the iSDM monitor.

A.3.5 F--Find

The F command searches the block of memory you specified to determine if it contains the sequence of hexadecimal digits you chose in the data parameter. Each time the iSDM monitor finds a match, it displays the address of the first matching byte.

A.3.6 G--Go

The G command instructs the iSDM monitor to begin executing your application program. In response to the G command, the iSDM monitor single steps the first instruction, then executes all succeeding instructions at full speed.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

With 80286 and 386 boards, a special situation arises when you execute the G command and you specify a breakpoint address but not a starting address. If the breakpoint is in an interrupt handler and the current CS:IP is at a software interrupt instruction (INT x, INTO, BOUND), the iSDM monitor single steps the interrupt instruction, executing the interrupt handler at full speed and bypassing the breakpoint you set. To get around this 80286/386 operational anomaly, make sure that the CS:IP is pointing to the (or any) instruction preceding the software interrupt instruction before you execute the G command.

A.3.7 I--Port Input

The I command retrieves and displays a byte or word from the port you specify. Byte and word formats are different. (See the *iSDM™ System Debug Monitor User's Guide* for byte and word format descriptions).

A.3.8 K--Echo File

The K command copies all console output to a development system file you specify. Repeating the K command without specifying a file causes the iSDM monitor to stop copying console output. Your system must include an attached development system in order to use this command.

A.3.9 L--Load Absolute Object File

The L command loads absolute 8086 or 80286 object files into target system memory. The iSDM monitor loads the data from the file into the memory location that you specified when you used the LOC86 or BLD286 commands. When loading the data, the iSDM monitor discards all previously loaded symbol information and loads the new symbol information, but it retains all user-defined symbols. If the file contains a register initialization record, the iSDM monitor sets the appropriate registers to the values the file specifies. Your system must include an attached development system in order to use this command.

The L command cannot load relocatable modules. If you are operating in real mode, you can load only 8086 absolute object files. If you are operating in protected mode, you can load only 80286 absolute object files.

When you load an 80286 object file, the iSDM monitor initializes the first 40 global descriptor table (GDT) entries for its own use. In addition, the iSDM monitor initializes any uninitialized interrupt descriptor table (IDT) entries. If the access byte is equal to zero, the iSDM monitor assumes that the descriptor table entry is not initialized. Refer to Intel's *Microprocessor and Peripheral Handbook*, *Microsystem Components Handbook*, or *80286 Operating System Writer's Guide* for more information about the descriptor tables.

A.3.10 M--Move

The M command copies the contents of a block of memory to a memory address you specify.

A.3.11 N--Execute Single Instructions

The N command displays and executes one or more disassembled instructions at a time. Going through your application line-by-line is called "single-stepping." Single-stepping allows you to begin at a CS:IP you specify and check your application for problems in an instruction-by-instruction manner.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

When you are single-stepping instructions, you should be aware of some special considerations. See the *iSDM™ System Debug Monitor User's Guide* for more information about these special considerations when using the N command.

A.3.12 O--Port Output

The O command allows you to enter data (a byte or word) at the console and send it to a port you select.

A.3.13 P--Print

The P command allows you to display either the value of an expression or the value of the base (or selector) and offset portions of an address. The values are displayed on your console terminal screen. The iSDM monitor always displays an address in hexadecimal form. If you enter "P" plus an expression, the iSDM monitor prints the value in hexadecimal. If you enter "PT" or "PS" plus an expression, the iSDM monitor prints the value in decimal or signed decimal form, respectively.

In this command, the comma acting as a separator also causes the iSDM monitor to add a space between the addresses or expressions it displays.

A.3.14 Q--Enable Protection (80286 or 386™ Only)

The Q command changes the 80286- or 386-based system from real mode to protected mode. The iSDM monitor displays the following message when you use the Q command:

```
Now in Protected Mode
```

When you invoke this command, the iSDM monitor initializes the entries it needs in the GDT and the IDT. The iSDM monitor then places itself at privilege level zero. If you are already operating in protected mode when you invoke this command, the iSDM monitor re-initializes the GDT and IDT entries. The only way you can return to real mode is to reset the 80286 or 386 hardware.

A.3.15 R--Load and Go

The R command is a combination of the Load command (L) and the Go command (G). This command loads an absolute object file from a development system into target system memory then executes this program. This command causes the iSDM monitor to discard all previously loaded symbol information and load new symbol information; however, the iSDM monitor retains all user-defined symbols. Your system must include an attached development system in order to use this command.

The iSDM monitor loads the data from the file into the memory location that you specified when you used the LOC86 or BLD286 commands. If the file contains a register initialization record, the iSDM monitor sets the appropriate registers to the values the file specifies.

The R command cannot load relocatable modules. If you are operating in real-addressing mode, you can load only 8086 absolute object files. If you are operating in protected mode, you can load only 80286 bootloadable (absolute) files.

When you load an 80286 object file, the iSDM monitor initializes the first 40 global descriptor table (GDT) entries for its own use. In addition, the iSDM monitor initializes any uninitialized interrupt descriptor table (IDT) entries. Refer to Intel's *Microprocessor and Peripheral Handbook*, *Microsystem Components Handbook*, or *80286 Operating System Writer's Guide* for more information about the 80286 component's descriptor tables.

After the iSDM monitor loads the file and sets the appropriate registers to the values the file specifies, it begins to execute the program at the location specified by the CS and IP registers.

Your application program must have at least 12 bytes of stack available for the iSDM monitor to use. If you are operating in protected mode, each task in your program must contain at least 12 bytes of stack at privilege level 0 for the iSDM monitor to use.

A.3.16 S--Substitute Memory/Descriptor Table Entry

The S command is actually two commands in one. You can use it to display and (optionally) modify either the contents of memory or the contents of descriptor table entries. If you are operating in real mode, you cannot display and modify descriptor table entries. However, if you are operating in protected mode, you can use both functions of this command.

If you enter the S command without an equal sign (=), the iSDM monitor displays a special hyphen (-) prompt. Then, it waits for you to enter either

- A continuation comma instructing the iSDM monitor to display the next memory location.
- A single expression or a list of expressions separated by slashes (/). By entering an expression (or expressions), you instruct the iSDM monitor to substitute these values in place of those already in the memory location you specified.

The iSDM monitor continues to issue hyphen prompts until you enter a carriage return.

A.3.17 X--Examine/Modify Registers

The X command allows you to examine and (optionally) modify the contents of your system's NPX and microprocessor registers.

If you use the X command with no parameters, the iSDM monitor displays all the 8086, 286, and 386 registers (except for control and debug registers).

If you use both the register name and an expression, (for example, CS = XXXX), the value you entered (XXXX) is placed in the specified register.

You can use the X command to set the 8086 family and NPX registers and the task state segment contents to any value. If you used any invalid values, the iSDM monitor reports them when you execute the application program.

A.3.18 Y--Symbols (80286 or 386™ Only)

The Y command allows you to display and define symbol information generated by 80286 translators. If you use the Y command with no parameters, the iSDM monitor displays all the symbols stored in the current domain module or in all modules if you set no domain. You can also choose to have the iSDM monitor display the symbols and their values in a particular module or you can use this command to define your own symbols. To use this command, you must be operating in protected mode, with an attached development system.

D-MON386 COMMANDS B

B.1 INTRODUCTION

This appendix briefly describes the 386™ Debug Monitor (D-MON386) commands in alphabetical order. A command directory listing the functional groups and page references precedes the command descriptions. For examples and more detailed information about the commands, see the *D-MON386 Debug Monitor for the 80386 User's Guide*.

B.2 ENTERING COMMANDS

To enter D-MON386 commands, follow the guidelines below:

- End a command line by pressing the ENTER key or the RETURN (<CR>) key. A command line can consist of one or more commands.
- Separate multiple commands on a single line using a semicolon (;).
- Continue commands from one line to another by entering the slash (/) just before terminating the line with the ENTER key or RETURN key.
- Enter commands using upper or lower case characters.
- Use CTRL-C (pressing the control key down while at the same time pressing the C key) to abort a command being constructed on the command line.

B.3 COMMAND DIRECTORY

This section provides a brief summary of all D-MON386 commands listed by functions. Each entry in the following summary contains along with the command name a brief description of the command and a page reference where you can find more information on the command.

B.4 COMMAND DESCRIPTIONS

This section provides brief descriptions for D-MON386 commands in an easily referenced alphabetical order. For on-line syntax help, refer to the HELP command. For more information on command parameters, syntax, and options, refer to the *D-MON386 Debug Monitor for the 80386 User's Guide*.

B.4.1 $

This command displays or modifies the current execution point via the execution address register (CS:EIP). The contents of CS:EIP determine which ASM386 statement executes next. Entering $ by itself displays the current contents of CS:EIP.

B.4.2 ASM

This command disassembles code into ASM386 opcode mnemonics. Using this command and the addresses you supply with it, you can disassemble from one to several lines of code. Disassembled code appears on the screen in column form. Each row of columns contains an address, a hexadecimal object value, an opcode mnemonic, any operands, and comments appended to the operands.

B.4.3 BOOT

This command invokes a user-supplied real mode interface program. The B command is intended primarily for including a bootstrap loader program.

B.4.4 BASE

This command displays or modifies the number base. Available number bases include binary, octal, decimal, and hexadecimal. The hexadecimal base is the monitor default base. Entering BASE by itself displays the current base. Entering BASE followed by an expression that evaluates to 2, 8, 10, or 16 (all decimal numbers) sets the base to binary, octal, decimal, or hexadecimal, respectively.

B.4.5 BYTE

This command displays or modifies partitions of memory using a byte format. You can specify the partition as a single byte or a range of bytes. Entering the command BYTE followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command BYTE as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.6 COUNT/ENDCOUNT

This command executes groups of D-MON386 commands in a specified order for a specified number of times. After entering COUNT expr, simply enter commands you wish to execute. After entering ENDCOUNT, one iteration of the commands will have already been executed. The entire group of commands then continues to execute for expr-1 number of times.

B.4.7 CREGS

This command displays the contents of the control registers and the EFLAGS register when the processor is in real mode. If the processor is in protected mode, the CREGS command also displays the system address registers TR and LDTR. The display appears using a hexadecimal number base.

B.4.8 DPORT

This command reads or writes a 32-bit port. Entering DPORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

B.4.9 DT

This command displays descriptors from either the LDT or the GDT depending upon the index supplied with the command.

B.4.10 DWORD

This command displays or modifies partitions of memory using a double word format. You can display a specific double word or a range of double words by entering DWORD followed by the single address or the range of addresses. Entering the DWORD command as an equation causes the partition of memory specified on the left-hand side of the equation to be replaced with the contents of memory or value of the right-hand side of the equation.

B.4.11 EVAL

This command evaluates the expression entered after the keyword EVAL. The results of the expression appear on the screen in binary, octal, decimal, hexadecimal, and ASCII formats.

B.4.12 FLAGS

This command displays the contents of the lower 16 bits of the EFLAGS register. The display appears in a mnemonic form. The presence of a mnemonic indicates a flag is set. The absence of a mnemonic in the display indicates a flag is not set.

B.4.13 GDT

This command displays the entire Global Descriptor Table (GDT) or individual GDT descriptors. Entering the keyword GDT by itself causes the entire GDT to appear. Entering GDT followed by an index expression causes a specific descriptor to appear.

B.4.14 GO

This command supplies high-level execution control. Use of the GO command enables you to begin and end program execution using specific points in the application. You can also clear and specify break conditions using the GO command.

B.4.15 Help

This command displays the major D-MON386 commands along with their general syntax. For examples and more detailed information about the commands, see the *D-MON386 Debug Monitor for the 80386 User's Guide.*

B.4.16 HOST

This command provides the capability for operation with PMON host software. When entering this command, be sure to press only the ENTER key or a carriage return <CR> immediately after HOST.

B.4.17 IDT

This command displays the entire Interrupt Descriptor Table (IDT) or individual IDT descriptors. Entering the keyword IDT causes the entire IDT to appear. Entering IDT followed by an index causes a specific descriptor from the IDT to appear.

B.4.18 INTn

This command displays or modifies partitions of memory using an integer format. When entering the command, you can substitute the numbers 1, 2, or 4 for n. Thus, the integer type(s) referenced in memory are either 1-, 2-, or 4-byte integers. You can specify the partition as a single INTn value or a range of INTn values. Entering the command INTn followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command INTn as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.19 ISTEP

This command does single-step execution. You can use this command to single-step through the executable code from one to 255 executable statements. ISTEP also provides the capability to begin execution from a point other than the current execution point.

B.4.20 LDT

This command displays the entire Local Descriptor Table (LDT) or individual LDT descriptors. Entering the keyword LDT causes the entire LDT to appear. Entering LDT followed by an index causes a specific descriptor from the LDT to appear.

B.4.21 N0-N9

This command displays or alters scratch registers zero through nine. Entering Nn (where n is a number 0 through 9) by itself causes the value of the appropriate register to appear on the screen. You can enter Nn followed by an equal sign and an expression to alter the contents of the appropriate scratch register.

B.4.22 ORDn

This command displays or modifies partitions of memory using an ordinal format. When entering the command, you can substitute the numbers 1, 2, or 4 for n. Thus, the ordinal type(s) referenced in memory are either 1-, 2-, or 4-byte ordinals. You can specify the partition as a single ORDn value or a range of ORDn values. Entering the command ORDn followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command ORDn as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.23 PD

This command examines the Page Table Directory and page tables. When paging is enabled, the 386 uses two levels of tables to translate a linear address into a physical address: the Page Table Directory and the page tables themselves. Entering the PD command by itself causes the entire 4K Page Table Directory to scroll to the screen. You can, however, supply an index with the PD command to view a particular directory entry within the Page Table Directory. Also, you can use the additional .PT option with an index to view a particular page table entry.

B.4.24 PORT

This command reads or writes a 8-bit port. Entering PORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

B.4.25 Register-name

D-MON386 enables you to display or alter the contents of 386™ registers. To gain register access, enter the name of the register Entering the name of the register only causes the contents of the register to appear on the screen. Entering the name of the register followed by an equal sign and a valid expression causes the contents of the register to be written with the value of the expression. For a complete list of register names, refer to the *D-MON386 Debug Monitor for the 80386 User's Guide*.

NOTE

Register modification is dependent on the current processor protection model. You cannot modify protected registers.

B.4.26 REGS

This command displays the contents of a set of registers as a group. The register set depends on which mode the processor is operating under (real or protected). The display is always in hexadecimal, and it provides less detail for the segment and control registers than the command that are specifically designed for those groups of registers, that is SREGS and CREGS, respectively.

B.4.27 SREGS

This command displays, in hexadecimal, the contents of the segment registers (CS, DS, SS, ES, FS, and GS).

B.4.28 SWBREAK

This command displays or sets code patch breaks. Entering SWBREAK by itself causes all current software break definitions to appear. If you enter SWBREAK followed by an equal sign and one or more addresses, the command sets a software break at the specified address or addresses.

NOTE

When specifying software break addresses, the address must be able to be written, present in physical memory, and on an instruction boundary. A maximum of 16 software breaks may be in effect at one time.

B.4.29 SWREMOVE

This command removes all or selected code patch breaks. Entering this command followed by ALL removes all current software breaks. If you supply one or more addresses with the command, the software breaks at those addresses alone are removed.

B.4.30 TSS

This command displays the contents of a task state segment. TSS supports both 386 and 80286 task state segments. Task state segments appear using the component names.

B.4.31 USE

This command specifies the default (16-bit or 32-bit code) for disassembling code from physical or linear addresses. When entering the command, the expression to the right of the equal sign must evaluate to either 16 or 32 (decimal).

B.4.32 VERSION

This command displays the version number of the D-MON386 software you are using.

B.4.33 WORD

This command displays or modifies partitions of memory using a word format. You can specify the partition as a single word or a range of words. Entering the command WORD followed by an address or range of addresses causes that partition of memory to appear on the screen. Entering the command WORD as an equation causes the partition of memory on the left side of the equation to be replaced with the contents of memory or value of the right side of the equation.

B.4.34 WPORT

This command reads or writes a 16-bit port. Entering WPORT with the physical input/output address space as a 16-bit unsigned quantity causes the specified port to be read and the contents to appear on the screen. If you supply an expression to the right of the equal sign when entering this command, the addressed port is written with the value the expression equals.

INDEX

A

B

C

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
Flower-Hill Shin-machi
1-23-9, Shinmachi
Setagaya-ku, Tokyo 15

NETHERLANDS
Intel Semiconductor (Netherland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor G.N.B.H.
Seidlestrasse 27
D-8000 Munchen

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of al Intel product users. This form lets you participate directly in the publication process. Your comment will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of thi publication. If you have any comments on the product that this publication describes, please contac your Intel representative.

1. Please describe any errors you found in this publication (include page number).

____

____

____

____

2. Does this publication cover the information you expected or required? Please make suggestion for improvement.

____

____

____

____

3. Is this the right type of publication for your needs? Is it at the right level? What other types o publications are needed?

____

____

____

____

4. Did you have any difficulty understanding descriptions or wording? Where?

____

____

____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ______________________________ DATE ____________

TITLE ____

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________ PHONE ( )____________

CITY ______________ STATE ______________ ZIP CODE ____________

(COUNTRY)

Please check here if you require a written reply. ☐

'E'D LIKE YOUR COMMENTS . . .

ıis document is one of a series describing Intel products. Your comments on the back of this form will ·lp us produce better manuals. Each reply will be carefully reviewed by the responsible person. All mments and suggestions become the property of Intel Corporation.

/ou are in the United States, use the preprinted address provided on this form to return your mments. No postage is required. If you are not in the United States, return your comments to the Intel les office in your country. For your convenience, international sales office addresses are printed on ǝ last page of this document.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
OMSO Technical Publications, MS: HF3-72
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9978

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051
(408) 987-8080