iRMX®
System Debugger Reference

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR   97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com

07-0581-01
December 1999

EPC, iRMX, INtime, Inside Advantage, and RadiSys are registered trademarks of RadiSys Corporation. Spirit, DAI, DAQ, ASM, Brahma, and SAIB are trademarks of RadiSys Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation and Windows 95 is a trademark of Microsoft Corporation.

IBM and PC/AT are registered trademarks of International Business Machines Corporation.

Microsoft Windows and MS-DOS are registered trademarks of Microsoft Corporation.

Intel is a registered trademark of Intel Corporation.

All other trademarks, registered trademarks, service marks, and trade names are property of their respective owners.

December 1999

Copyright © 1999 by RadiSys Corporation

All rights reserved.

Quick Contents

Chapter 1. Overview of Debugging Tools

Chapter 2. System Debugger (SDB) Commands

Chapter 3. System Debug Monitor (SDM) Commands

Appendix A. Console I/O Calls

Appendix B. Related Publications

Index

Notational Conventions

Most of the references to system calls in the text and graphics use C syntax instead of PL/M (for example, the system call **send_message** instead of **send$message**). If you are working in C, you must use the C header files, *rmx_c.h*, *udi_c.h* and *rmx_err.h*. If you are working in PL/M, you must use dollar signs ($) and use the *rmxplm.ext* and *error.lit* header files.

This manual uses the following conventions:

- Syntax strings, data types, and data structures are provided for PL/M and C respectively.
- All numbers are decimal in text and hexadecimal in commands, unless otherwise stated. Hexadecimal numbers include the `H` radix character (for example, `0FFH`). Binary numbers include the `B` radix character (for example, `11011000B`). Decimal numbers in commands include the `T` radix character (for example, `10T`).
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless otherwise stated.
- `Data structures and syntax strings appear in this font.`
- **System call names and command names appear in this font.**
- PL/M data types such as BYTE and SELECTOR, and iRMX data types such as STRING and SOCKET are capitalized. All C data types are lower case except those that represent data structures.
- The following OS layer abbreviations are used. The Nucleus layer is unabbreviated.

AL	Application Loader
BIOS	Basic I/O System
EIOS	Extended I/O System
HI	Human Interface
UDI	Universal Development Interface

- Whenever this manual describes I/O operations, it assumes that tasks use BIOS calls (such as **rq_a_read**, **rq_a_write**, and **rq_a_special**). Although not mentioned, tasks can also use the equivalent EIOS calls (such as **rq_s_read**, **rq_s_write**, and **rq_s_special**) or UDI calls (**dq_read** or **dq_write**) to do the same operations.

Contents

1 Overview of Debugging Tools

Debugging Tools 1
Hardware and Software Requirements 2
Starting SDM and SDB 3
Using SDB Commands 4
    Command Syntax 4
    Using Tokens as Command Parameters 6
    Entering Commands 6
Leaving the Monitor 7
    Warm-Starting a System (iRMX III OS and iRMX for PCs Only) 7
    CLI-Restarting a System 7
Returning to your Application 7

2 System Debugger (SDB) Commands

vb 11
vc 15
vd 18
vf 20
vh 21
vj 22
vk 25
vmf 26
vmi 27
vmo 30
vo 33
vr 35
vs 39
vt 45
    Job Display 46
    Task Display 48
    Mailbox Display 51
    Semaphore Display 53

Region Display ... 54
Segment Display ... 55
Extension Object Display ... 55
Composite Object Display ... 56
Display of Composite Objects Other Than BIOS or EDOS ... 56
Display of BIOS Composite User Object ... 57
Display of BIOS Physical File Connection ... 57
Display of BIOS Stream File Connection ... 62
Display of BIOS Named File Connection ... 62
Display of BIOS Remote File Connection ... 65
Display of BIOS EDOS File Connection ... 65
Display of Service Object ... 66
Display of a Generic Port ... 67
Display of a Signal Port (Nucleus Communications Service only) ... 69
Display of a Data Port (Nucleus Communications Service only) With Messages Queued ... 70
Heap Display ... 72
Buffer Pool Display ... 72
vu ... 74

3 System Debug Monitor (SDM) Commands

Command Structure ... 80
Entering Commands ... 81
Command Line Conventions ... 81
Command-Editing Keys ... 81
Command Line History ... 81
Multiple Commands on a Single Line ... 82
Combining Commands ... 82
Repeating Commands ... 82
Continuing Commands ... 82
Command Parameters ... 84
Byte, Halfword and Word Parameters ... 84
Term Parameters ... 85
Expression Parameters ... 86
Address Parameter ... 86
Numeric Parameters ... 87
NPX Integers ... 88
NPX Real Numbers ... 89
Packed Binary Coded Decimal (BCD) Numbers ... 89
NPX Number Format ... 90
Decimal Values ... 91
Nonnumeric Values ... 91

Special-Case Numeric Values 92
Error Messages 93
bc 94
bs 96
Reason Codes 98
No Breaks Available 98
Software Breakpoints 98
Hardware Breakpoints 99
Execution Breakpoints 99
Data Breakpoints and I/O Access Breakpoints 99
Breakpoint Display 100
c 102
d 103
f 107
g 108
i 110
m 112
n 113
o 115
pdbr 117
pdd 118
pdp 120
pdt 121
psd 123
pst 124
s 125
x 130

A Console I/O Calls

Using the Console I/O Calls 137
ci 138
co 139
csts 141

B Related Publications 143

Index 145

Tables

Table 2-1. SDB Commands 9
Table 3-1. SDM Commands 79
Table 3-2. CPU Registers (Protected Mode) 85
Table 3-3. NPX Registers 87
Table 3-4. NPX Data Types 88
Table 3-5. NPX Integer Types 88
Table 3-6. NPX Real Types 89
Table 3-7. Error Messages 93
Table 3-8. Descriptor Components and Types 105
Table 3-9. Descriptor Types 105
Table 3-10. NPX Registers 133
Table 3-11. Task State Segment 134
Table A-1. Console I/O Calls 137

Overview of Debugging Tools 1

The iRMX® Operating System provides tools for debugging iRMX applications and systems programs. This manual is a reference for using the System Debugger (SDB) and System Debug Monitor (SDM). SDB is an extension of SDM.

This manual is for system programmers who are implementing iRMX applications, device drivers, object managers, and operating system extensions. To use and understand SDB commands, you should be familiar with the concepts and terminology of the iRMX Nucleus. To use SDM commands, you should be familiar with the registers and addressing modes of the microprocessor.

See also: *System Concepts*
Debugging example, *Programming Techniques*

Debugging Tools

The iRMX development environment includes these debugging tools:

System Debug Monitor (SDM)

A hardware debug monitor. Use SDM to disassemble code, set and execute breakpoints, display or change microprocessor registers, and display or change the contents of memory. SDM is installed along with the OS. It is a 32-bit protected mode monitor that supports the Intel386™, Intel486™, and Pentium® microprocessors.

See also: SDM Commands, Chapter 3

iRMX System Debugger (SDB)

A system job that operates on top of SDM to retrieve information about iRMX objects such as jobs, tasks, and mailboxes. Use SDB to interpret data structures maintained by the OS; for example, iRMX system calls and stacks, the Global Descriptor Table (GDT), and messages sent across the backplane in Multibus II systems. To use SDB, first start SDM, then enter SDB commands at the SDM prompt.

See also: SDB commands, Chapter 2

Soft-Scope

The Soft-Scope debugger is a multitasking industry-standard debugger that features source-level and symbolic debugging of your iRMX applications. Use the Soft-Scope debugger for higher-level debugging tasks. SoftScope provides access to SDB commands, but you cannot enter SDM commands from SoftScope. It is a product of Concurrent Sciences, inc., and is not described further in this manual.

See also: *Soft-Scope Debugger User's Guide*

Note

Previous releases of the iRMX OS described an interface to the iM III Monitor, a low-level interface to the system hardware. The functionality of the iM III Monitor has been moved into SDM. For example, SDM now has a **bs** command to set breakpoints. For that reason, this manual no longer includes an appendix describing iM III Monitor commands.

Hardware and Software Requirements

You need the following hardware, firmware, and software to support SDM and SDB:

- A system connected to an Intel386, Intel486, or Pentium processor board
- A terminal connected directly to the processor board
- The iRMX OS

The DOSRMX and iRMX for PCs OSs include SDM by default. You can load SDB as a loadable system job. In the iRMX III OS, you can configure SDM and SDB by using the Interactive Configuration Utility (ICU).

See also: *sdb.job*, *System Configuration and Administration*
SUB, SDB, and SDM screens, *ICU User's Guide and Quick Reference*

Starting SDM and SDB

Since SDB is an extension of SDM, you must first start SDM, which displays a . . prompt. If you have loaded the SDB job, you can then enter either SDB commands or SDM commands at this prompt.

In the DOSRMX or iRMX for PCs OS, invoke SDM in one of these ways:

- Use the Human Interface **debug** command to load an application program into main memory and transfer control to SDM.
- Insert an Int3 instruction in your code at the point where you want to break to SDM.
- (DOSRMX only) While iRMX owns the system console, enter <Ctrl-Alt-Break>. If you are at the DOS prompt instead of the iRMX prompt, first press <Alt-Sys-Req> to toggle from DOS to the iRMX OS.

In the iRMX III OS, invoke SDM with one of the first two methods described above or with these techniques:

- Use the front panel interrupt button on your Multibus II system. This is a hardware switch physically connected to the Interrupt 3 level that invokes SDM. Activating this switch halts the application system, saves the system's context, and passes control to SDM.
- Load the OS using the Bootstrap Loader with the `debug` option. Once the system is loaded and the Nucleus has finished initializing, SDM begins execution. At this point you can use only SDM commands, since SDB has not been initialized. SDB is a first-level job that initializes right after the Nucleus.

  See also: Booting with debugging, *iRMX Bootstrap Loader Reference Manual*

When you invoke SDM with any of the above methods, the application system stops running and all system activity freezes. You can enter commands to set breakpoints, step through a program, view memory, inspect system objects, change system call parameters and register values, and test changes.

See also: SDM Commands, Chapter 3
SDB Commands, Chapter 2

When your application is running, control passes to SDM when there is a breakpoint at an address and CS:EIP (code segment and instruction pointer registers) reaches the breakpoint. Use SDM’s **g** (go) command to restart your application and set some initial breakpoints in your code. You can use the output from MAP386 to identify where to set breakpoints.

For example, to invoke the **g** command and set two breakpoints, at the SDM prompt (. .) enter:

```
.. g, 7fa, 1f0:e20
```

The application begins executing at the current CS:EIP. SDM is again invoked when CS:EIP reaches the first breakpoint.

See also: MAP utility, *Intel386 Family Utilities*

Using SDB Commands

There are four kinds of SDB commands:

- Commands that display iRMX data structures and objects
- Commands that disassemble code by recognizing and displaying iRMX calls
- Commands that display features of the Message Passing Coprocessor (MPC)
- A command that provides help with short descriptions of all the SDB commands

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

See also: Debugging example, *Programming Techniques*

Command Syntax

All SDB commands begin with a `v` followed by one or more characters that represent the command name and parameters. Enter any command or parameter in upper or lower case. Spaces are optional between the command name and parameters. Include any punctuation as shown below except brackets ([ ]), ellipses (. . .), and the vertical bar (|).

```
command substitute [optional] [choice= item1|item2]
      [repeated [item] [, repeated [item]]...]
```

`command`
Enter any item printed like this exactly as it is shown.

`substitute`
For variable items, enter the appropriate information, such as a token for an object.

`[optional]`
Items surrounded by brackets indicate an optional parameter. If you enter this parameter do not include the brackets.

`[choice= item1|item2]`

For items separated with a vertical bar, enter only one of the items. You may enter `choice= item1` or `choice= item2`, but not both.

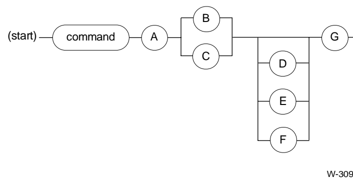
`[repeated [item] [, repeated [item]]...]`

Items followed by an ellipsis (`...`) indicate that the item may be repeated more times than it is shown. For this example, any of these would be valid entries:

```
repeated
repeated item
repeated, repeated, repeated
repeated item, repeated, repeated item
```

etc.

A few commands with many parameters have an additional syntax diagram. The parameters are listed along a track, as shown below. Enter the track at the top left and follow it through to the exit. Mandatory parameters are shown in line with the track. Optional parameters are shown below the track. You can follow the main track or follow the path through the option and return to the main track. Where you have a choice of parameters, the track branches through them.



In this example:

- A is a required parameter and you must enter it immediately after the command.
- Either B or C is required. Whichever parameter you enter must follow A.
- D, E, and F are all optional. You may select only one. If you select one of these parameters, enter it before or after G.

Using Tokens as Command Parameters

Many SDB commands use iRMX tokens as parameters or display tokens as part of the command output. The iRMX OS maintains tokens in doubly linked lists. SDB checks a token's forward and backward links to determine the token's validity.

A token is invalid if one or more of these is true:

- Both token links are bad
- The token belongs to an object being deleted
- An incorrect token is entered as a system call parameter
- A token is deleted or unused

SDB displays this message for an invalid token:

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

These are displays for forward and backward link errors:

```
Forward link ERROR:  4108-->4E88  4108<--4E88-->4158  ?FFFF<--4158

Backward link ERROR:  4108-->410F?  4108<--4E88-->4158  4E88<--4158
```

The token you entered appears as the center value in each line of the token display. Left arrows indicate backward links; right arrows indicate forward links. A question mark before a value indicates a forward link error; a question mark after a value indicates a backward link error.

A link error can happen when a task overwrites a token's data structure or when you use the Non-Maskable Interrupt (NMI) and the Nucleus is interrupted while setting up the links.

See also: NMI, in the hardware reference manual for your microprocessor

Entering Commands

Enter commands from a console attached to your target system. The command line examples in this chapter do not show spaces as elements, but you may include one or more spaces between the command and parameter. For example, these command lines are both valid; the space between `vr` and `xxxx` is optional:

```
..vr xxxx
..vrxxxx
```

To repeat an earlier SDB or SDM command without retyping it, enter <Ctrl-B> repeatedly to scroll back through previous commands. Enter <Ctrl-F> to scroll forward in the list. On a PC keyboard, you can also use the <UpArrow> and <DownArrow> keys for this purpose.

Leaving the Monitor

You can leave SDM/SDB without resetting your system by warm-starting or CLI-restarting your system. You also leave SDM when your application terminates normally.

Warm-Starting a System (iRMX III OS and iRMX for PCs Only)

The warm-start feature starts a system without reloading it from secondary storage. Warm-start reinitializes the system, but does not initialize memory. It begins executing the application system at the same point where the Bootstrap Loader passes control to the system.

To warm-start a system from the SDM prompt, enter:

```
..g 284:e
```

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. This command does not work in DOSRMX.

CLI-Restarting a System

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

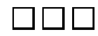
```
..g 284:1c
```

This command causes the system to attempt to delete the job subtree associated with 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 CLI. Otherwise, you must reboot the system.

Returning to your Application

Use SDM's `g` command to resume execution of the application in these cases:

- When you finish debugging your application system with SDB
- To test the changes you made to the application code



System Debugger (SDB) Commands 2

This chapter is a reference to the iRMX SDB commands, which appear in alphabetical order. Table 2-1 lists the SDB commands.

Table 2-1. SDB Commands

Command	Description
vb	Displays DUIB information
vc	Displays system call information
vd	Displays a job's object directory
vf	Displays number of GDT free slots
vh	Displays help information
vj	Displays job tree
vk	Displays tokens of ready and sleeping tasks
vmf	Enables or disables MPC fail-safe timeout
vmi	Displays input messages
vmo	Displays output messages
vo	Displays job objects
vr	Displays BIOS or EIOS IORS segment
vs	Displays stack and system call information
vt	Displays iRMX object information
vu	Displays system calls in a task's stack

This chapter uses these conventions:

- CS:EIP is the code segment and the instruction pointer to the next instruction to be executed.
- SS:ESP is the stack segment and stack pointer to the current stack location.
- Entering 0 as a value for an optional parameter omits the parameter, unless a default parameter value is used.
- When executing SDB commands from SDM, the input prompt is the SDM prompt (..).
- When executing SDB commands from the Soft-Scope debugger, the input prompt is the Soft-Scope prompt (ss>). SDM is not active.

vb

Displays the DUIB information for a physical device in the system configuration.

Syntax

```
vb device-name
```

Parameter

`device-name`
Name of a physical device in the system configuration, for example, the c_rmx3 device.

Additional Information

See also: DUIBs, *Driver Programming Concepts*

This is the format of the DUIB display:

```
Device name:      <physical device name>

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

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

File driver:      xxxx                Named         xxxx
  Physical        xxxx                Stream        xxxx
  EDOS            xxxx
  DOS             xxxx
```

The fields displayed are:

`<physical device name>`
The physical device name for which you are displaying information.

`Functs` A byte that indicates the I/O function for this physical device.

`DUIB address`
: The starting address in memory of the specified DUIB.

`Dev$gran` A 16-bit word that specifies the device granularity. Device granularity is the minimum number of bytes of information the device reads or writes in one operation. This parameter applies to random access devices and to some common devices, such as tape drives.

`Max$buffers`
: The maximum number of buffers that the EIOS can allocate for a connection to this physical device when the connection is opened by an **s_open** EIOS system call.

`Dev$size` The number of bytes the physical device can store.

`Device` The physical device number.

`Unit` The unit number that distinguishes the unit from other units in the device.

`Dev$unit` The physical device unit number that distinguishes the unit from other device units in the hardware system.

`Device$info$p`
: The pointer to the Device Information Table for the device unit. Common, random, and terminal device drivers require a Device Information Table for each device.

`Unit$info$p`
: The pointer to a Unit Information Table for the device. Common, random, and terminal device drivers require this Unit Information Table for each device.

`Update$timeout`
: The number of system time units the I/O System must wait before writing a partial sector and after processing a write request for a disk device.

`Num$buffers`
: The number of buffers allocated for the device by the I/O System.

`Priority` The device's service task priority set by the I/O System.

`Fixed$update`
: Indicates whether the fixed update option was selected for this physical device when the application system was configured.

`Init$io` The offset address of the Initialize I/O procedure associated with this unit.

`Finish$io` The offset address of the Finish I/O procedure associated with this unit.

`Queue$io` The offset address of the Queue I/O procedure associated with this unit.

`Cancel$io` The offset 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` Device density. If the `Flags` field is `Not Valid`, this field is marked `N/A`.

`Sides` The number of media sides to which the device can write. If the `Flags` field is `Not Valid`, this field is marked `N/A`.

`Size` The physical size of the device. If the `Flags` field is `Not Valid`, 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` field is `Not Valid`, this field is marked `N/A`.

`File driver`
A 16-bit word that indicates the BIOS file driver to which this connection is attached.

`Named` `TRUE` or `FALSE`. Indicates whether this device is configured to use the Named file driver.

`Physical` `TRUE` or `FALSE`. Indicates whether this device is configured to use the Physical file driver.

`Stream` `TRUE` or `FALSE`. Indicates whether this device is configured to use the Stream file driver.

`EDOS` `TRUE` or `FALSE`. Indicates whether this device is configured to use the EDOS file driver.

`DOS` `TRUE` or `FALSE`. Indicates whether this device is configured to use the DOS file driver.

Error Messages

`Syntax Error`

You made an error entering the command.

`VB not supported`

The command cannot 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 returns.

`DUIB not found`

The command returns this error message under these conditions:

- The DUIB is not configured into the system.
- The DUIB entry for the specified device is located before the byte bucket DUIB entry.
- You entered the physical device name incorrectly.

vc

Displays information for an iRMX system call.

Syntax

```
vc [pointer]
```

Parameter

`pointer`

A selector:offset pair that is the address of a CALL instruction. The address has this form:

```
ssss:oooooooo
```

The four-digit hexadecimal value `ssss` indicates the address selector. The eight-digit hexadecimal value `oooooooo` indicates the address offset.

If you do not supply a pointer or you specify 0, this parameter defaults to the current CS:EIP. If you specify an EIP value (one eight-digit hexadecimal number) but not a CS value, the default is the current CS.

Additional Information

If the CALL instruction is for an iRMX system call, information is displayed about the system call. If the CALL instruction is not an iRMX system call or the instruction is not a CALL instruction, a message is displayed to this effect.

This is the format of the iRMX system call information display:

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

The fields are:

`gate #NNNN`	Gate number for the iRMX system call
`(subsystem)`	iRMX OS layer of the system call
`system call`	iRMX system call name

The gate number determines if the CALL instruction is for an iRMX system call or a C Library call. Information for iRMX system calls is usually displayed, but occasionally information for a non-system call may be displayed.

Examples

Suppose you disassembled this code:

```
18a0:0000006d 50        push    ax
18a0:0000006e e8ad1e    call    a = 1f1e          ;$+7856
18a0:00000071 e8dd03    call    a = 0451          ;$+992
18a0:00000074 b80000    mov     ax,0
18a0:00000077 50        push    ax
18a0:00000078 8d060600  lea     ax,word prt 006
18a0:0000007c 1e        push    ds
18a0:0000007d 50        push    ax
18a0:0000007e e8411e    call    a = 1ec2          ;$+7748
18a0:00000081 a30000    mov     word ptr 0000h,ax
```

Use the **vc** command on the CALL instruction at address 18A0:0000006E by entering this command:

```
..vc 18A0:6E
```

This displays the iRMX system call name as:

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

The iRMX system call is the Nucleus system call **set_exception_handler** with the gate number 0468.

To see if the CALL instruction at 18A0:00000071 is a system call, enter:

```
..vc 18A0:71
```

This message indicates the call was not an iRMX system call:

```
Not a system CALL
```

To see if the instruction at 18A0:00000074 is a CALL instruction, enter:

```
..vc 18A0:74
```

This message indicates the instruction was not a CALL instruction:

```
Not a CALL instruction
```

Error Messages

`Syntax Error`

You made an error entering the command.

`Not a system CALL`

The specified parameter points to a CALL instruction that is not an iRMX system call.

`Not a CALL instruction`

The CS:EIP does not point to a CALL instruction.

vd

Displays the object directory for a job.

Syntax

```
vd job-token
```

Parameter

`job-token`

The token for a job you want information about. Use the **vj** command to obtain this token.

Additional Information

This is the format of the object directory display:

```
Directory size:  xxxx      Entries used:  xxxx

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

The display 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 cataloged. These names were assigned at the time the objects were cataloged with the Nucleus system call **catalog_object**.

`token1...tokenn`
The tokens for the cataloged objects.

`tasks waiting`
Signifies that one or more tasks have done a Nucleus system call **lookup_object** on an object not cataloged. The tokens following this field identify the tasks still waiting for the object to be cataloged.

See also: Object directories, *Introducing the iRMX Operating Systems and System Concepts*

Example

For example, to look at the object directory of job 2280, enter:

```
..vd 2280
```

The object directory display is:

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

$              2228
R?IOUSER       2200
RQGLOBAL       2280
```

In this display, the symbols `$`, `R?IOUSER`, and `RQGLOBAL` are the object names in the job's object directory for the respective tokens `2228`, `2200`, and `2280`.

Error Messages

`Syntax Error`
No parameter was specified for the command, or you made an error entering the command.

`TOKEN is not a Job`
You entered a token that is not a job token.

`*** INVALID TOKEN ***`
You entered a value that is not a valid token.

See also: Using tokens as command parameters, Chapter 1

vf

Displays the number of free Global Descriptor Table (GDT) slots available.

Syntax

```
vf
```

Additional Information

This is the format of the **vf** command display:

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

Error Messages

`Syntax Error`
You made an error entering the command.

vh

Lists the SDB commands with their parameters and descriptions.

Syntax

```
vh
```

Additional Information

In this example of the help display, angle brackets (< >) surround required variable fields; square and angle brackets ([< >]) surround optional fields.

```
iRMX III System Debugger, Vx.y
Copyright xxxx 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.
```

The system uses default values if you specify 0 for any of the optional parameters. Using 0 for required parameters causes the system to display a syntax error message.

Error Message

`Syntax Error`

You made an error entering the command.

vj

Displays the tokens in the job tree hierarchy beginning with a specified job token.

Syntax

```
vj [job-token]
```

Parameter

`job-token`
The token of the job from which you want to begin displaying the job tree hierarchy. If you omit this parameter or specify 0, the job tree display begins with the root job.

Additional Information

The tokens for descendant jobs in the job tree are indented to show their position in the tree hierarchy.

If the job tree has more than 44 descendant jobs, tokens are displayed up to the 44th entry and an error message is displayed.

This is the format of the job tree display:

```
        iRMX Job Tree

token1                                  (Root Job)
        token2                          (Human Interface)
                token3                  (Command Line  Interpreter)
                        token4          (Application)
        token5                          (EIOS)
        token6                          (iRMX-NET)
        token7                          (BIOS)
```

The fields in the job tree display are:

`token1` The token you specified as `job token`; the root job token is the default.

`token2...token7`
The tokens for the descendant jobs of `token1`.

The layer names in parentheses are not shown in the actual job tree display. There are comments added to show the layers for the default job tree.

The Human Interface, EIOS, and BIOS Jobs are indented three spaces indicating they are children of the Root Job. The Command Line Interpreter Job is the child of the Human Interface Job as are all first level user jobs. The Application Job is the child of the Command Line Interpreter Job.

Examples

To examine the hierarchy of the root job, enter:

```
..vj
```

This is an example job tree display:

```
          iRMX Job Tree

0258
   0f38
      1670
         2460
   0e88
   0e00
```

If you want to display the descendant jobs of `0e88`, enter:

```
..vj 0e88
```

This displays the job tree information:

```
          iRMX Job Tree

0e88
0e00
0f38
   1670
      2460
```

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.

Error Messages

`SDB job nest limit exceeded`
The job has more than 44 job descendants.

`Syntax Error`
You made an error entering the command.

`TOKEN is not a Job`
You entered a token that is not a job token.

`*** INVALID TOKEN ***`
You entered a value that is not a valid token.

See also: Using tokens as command parameters, Chapter 1

vk

Display the tokens for ready and sleeping tasks.

Syntax

```
vk
```

Additional Information

This is the format of the **vk** display of token information:

```
Ready tasks:    xxxx xxxx ...

Sleeping tasks: xxxx xxxx ...
```

The fields show:

`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`
: You made an error entering the command.

`Ready tasks:  Can't locate`
: The system is corrupted.

`Sleeping tasks:  Can't locate`
: Usually indicates the Nucleus is uninitialized. To recover from this error, reinitialize the system.

vmf

Enables or disables the Message Passing Coprocessor (MPC) fail-safe timeout feature. This command applies to Multibus II systems only.

Syntax

```
vmf
```

Additional Information

When the fail-safe timer is enabled, this command disables it. When the fail-safe timer is disabled, this command enables it.

When debugging a message passing application, disable the fail-safe timer. This allows stopping a host while debugging commands are executing. When finished debugging, enable the fail-safe timer before starting your application. Otherwise, the application may not function properly.

The MPC fail-safe timer limits how long the MPC waits between sending a buffer request message and receiving a buffer grant or buffer reject message. The wait is about two seconds on an iSBC 386/116 or iSBC 386/120 board. This ensures that the MPC will not wait forever when communicating with another host.

This command is available for preconfigured systems. On configurable systems, specify at least one trace message in the Number of Trace Messages option in the ICU's Nucleus Communication Service screen.

See also: NTM, *ICU User's Guide and Quick Reference*

Example

If you invoke the **vmf** command when the fail-safe timer is enabled, the fail-safe timer is disabled and this message is displayed:

```
MPC Failsafe Timer Is Disabled
```

If you invoke the **vmf** command when the fail-safe timer is disabled, the fail-safe timer is enabled and this message is displayed:

```
MPC Failsafe Timer Is Enabled
```

Error Messages

`Syntax Error`

You made an error entering the command.

vmi

Displays the most recent messages received from the Message Passing Coprocessor (MPC). This command applies to Multibus II systems only.

Syntax

```
vmi [message#][,]
```

Parameters

`message#`

Message number to display. If omitted, the most recent message is displayed. When the comma (,) parameter is entered, this specifies the first message to display.

`, (comma)`

Requests viewing more than one message in the input message buffer. The most recent message and a dash (-) at the end of the line are displayed. Repeat this process by entering another comma or end the command by entering a <CR> at the dash. The comma parameter is not supported when this command is entered from the Soft-Scope debugger.

Additional Information

This command is available for preconfigured systems. On configurable systems, specify at least one trace message in the Number of Trace Messages option in the ICU's Nucleus Communication Service screen.

The number of messages you can display depends on the number of trace messages configured in the system. For example, if the Number of Trace Messages is set to five you can display the five most recent messages.

See also: Number of trace messages, *ICU User's Guide and Quick Reference*

This 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.

See also: Message fields, *Multibus II Transport Protocol Specification and Designer's Guide*

This is the format of the **vmi** display:

```
## <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
```

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 of the requested transfer, in bytes. This field is only displayed for buffer request messages; otherwise 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 transport protocol, this is displayed:

```
Unknown Protocol
```

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

`<trans control>`
Indicates the type of request or response message. If the message is not a request or response message, this field is blank. These are the possible values:

Resp/EOT	Response message, end-of-transaction (EOT). This is the last fragment of a reply.
Resp/Not EOT	Response message, not end-of-transaction (EOT). More fragments of the reply will follow.
Resp/Cancel	Response message with cancellation. The server sending the reply is canceling the transaction.
Resp/Reserved	Reserved type.
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` A number that identifies the transaction ID. This field is 0 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.

`len` The length of the requested fragment, in bytes.

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

You cannot use the **vmi** command to view the contents of short-circuit messages. Short-circuit messages are messages passed between tasks that run on the same board.

Error Messages

`Syntax Error`

You made an error entering the command**.**

`Message Information Is Not Available`

The system is not a Multibus II system or no trace messages are specified in the system configuration.

vmo

Displays the most recent output messages sent by the Message Passing Coprocessor (MPC). This command can be used only in a Multibus II system.

Syntax

```
vmo [message#][,]
```

Parameters

`message#`

Message number to display. If omitted, the most recent message is displayed. When the comma (,) parameter is entered, this specifies the first message to display.

`, (comma)`

Requests viewing more than one message in the input message buffer. The most recent message and a dash (-) at the end of the line are displayed. Repeat this process by entering another comma or end the command by entering a <CR> at the dash. The comma parameter is not supported by the Soft-Scope debugger.

Additional Information

This command is available for preconfigured systems. On configurable systems, specify at least one trace message in the NTM option in the ICU's NCOM screen.

The number of messages you can display depends on the number of trace messages configured in the system. For example, if the NTM is set to five you can display the five most recent messages.

See also: NTM, *ICU User's Guide and Quick Reference*

This 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.

See also: Message fields, *Multibus II Transport Protocol Specification and Designer's Guide*

The format of the **vmo** output depends on the type of message. These are the possible fields for the output display:

```
## <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
```

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` Only this part of the first line is displayed for buffer request, buffer grant, and buffer reject messages. It can consist of one of two fields. For buffer request messages, this field is displayed:

`len` The length of the requested transfer, in bytes.

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, this message is displayed:

```
Unknown Protocol
```

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

`<trans control>`
Indicates the type of request or response message. If the message is not a request or response message, this field is blank. These are the possible values for this field:

Resp/EOT	Response message, end-of-transaction (EOT). The last fragment of a reply.
Resp/Not EOT	Response message, not end-of-transaction (EOT). More fragments of the reply will follow.
Resp/Cancel	Response message with cancellation. The sender of the reply (the server) is canceling the transaction.
Resp/Reserved	Reserved type.

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 is 0 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.

`len` The length of the requested fragment, in bytes.

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

You cannot use the **vmo** command to view the contents of short-circuit messages. Short-circuit messages are messages passed between tasks that run on the same board.

Error Messages

`Syntax Error`

You made an error entering the command.

`Message Information Is Not Available`

The system is not a Multibus II system or no trace messages were specified during configuration.

vo

Displays information about the objects in a job.

Syntax

```
vo job-token
```

Parameter

`job-token`
The token of the job for which you want to display object information.

Additional Information

This command lists the tokens for a job's child jobs, tasks, mailboxes, semaphores, regions, segments, extensions, composites, and buffer pools.

The format of the **vo** command is:

```
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    ...
```

The fields in the display are:

`Child Jobs`
The tokens for the child jobs.

`Tasks` The tokens for the tasks in the 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 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 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 job.

`Extensions`
The tokens for the extensions in the job.

`Composites`
The tokens for the composites in the job. An `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 job.

Example

To look at the objects in the job having the token 1670, enter:

```
..vo 1670
```

This displays:

```
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 job's tokens and that tasks are waiting at four mailboxes and one semaphore.

Error Messages

`Syntax Error`
You did not specify a parameter for the command or you made an error entering the command.

`TOKEN is not a Job`
You entered a token that is not a job token.

`*** INVALID TOKEN ***`
You entered a value that is not a valid token.

See also: Using Tokens as Command Parameters, Chapter 1

vr

Displays information about the BIOS and EIOS I/O Request/Result Segment (IORS) for a segment token.

Syntax

```
vr segment-token
```

Parameter

`segment-token`
The segment token for the IORS you want to display.

Additional Information

The IORS contains information about the most recent I/O operation.

If you do not enter a valid segment token for the IORS, the **vr** command returns invalid information. Use the **vo** command to obtain a list of the valid segment tokens in a job.

See also: IORS, *System Concepts* and *Driver Programming Concepts*

The display format for the IORS information is:

```
I/O Request Result Segment

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

The fields in the display are:

`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 BIOS. The possible functions are:

Function	**System Call**
Read	a_read
Write	a_write
Seek	a_seek
Special	a_special
Att Dev	a_physical_attach_device
Det Dev	a_physical_detach_device
Open	a_open
Close	a_close

If this field contains an invalid value, the actual value is displayed 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 **a_special** or EIOS **s_special** system calls. These are the possible subfunctions and their descriptions:

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 does not contain `Special`, then this field contains `N/A`. If this field contains an invalid value, the field value is displayed 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 BIOS 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`
: `TRUE` (0FFH) or `FALSE` (00H). This field indicates whether the I/O operation has been completed.

`Cancel ID`
: A word used by device drivers to identify I/O requests that need to be canceled. A value of 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`

You did not specify a parameter for the command or you made an error entering the command.

`TOKEN is not a SEGMENT`

You entered a token that is not a segment token.

`*** INVALID TOKEN ***`

You entered a value that is not a valid token.

See also: Using Tokens as Command Parameters, Chapter 1

`SEGMENT wrong size - not an IORS`

The specified segment is not between 54 and 70 bytes long, so it is not an I/O Request/Result Segment.

vs

Displays current information about the stack and system calls on the stack.

Syntax

```
vs [count]
```

Parameter

`count` A decimal or hexadecimal value for the number of words from the stack to display. A suffix of T, as in 16T, means decimal. No suffix or a suffix of H indicates hexadecimal.

By default, the number of words in the display depends on the number of parameters for the system call at the CS:EIP. When CS:EIP is not pointing to a system call, the entire contents of the stack are displayed.

Additional Information

If the stack does not contain a system call, the display is either the number of stack elements you specify or all the stack contents, whichever is least. If a parameter is a string, the string is displayed.

The current SS:ESP registers are used to display the current stack values. The current CS:EIP is used for system call and parameter information. To change the CS:EIP value, use the monitor's **g** or **x** command. For Soft-Scope, use the **go** or **reg** command.

If the current instruction is not a CALL instruction, the contents of the stack are displayed without a message. If the instruction is a CALL but not a system call, the stack contents are displayed with a message that the call is not a system call.

The gate number is displayed if the call is a C Library call.

This is the display format for system call information:

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

(subsystem) system call
```

```
|parameters|
```

The fields in the display are:

`gate #NNNN` The gate number associated with the system call.

`xxxx:xxxxxxxx`
The contents of the SS:ESP (stack memory addresses).

`xxxxxxxx` Values now on the stack. The number of stack values varies with the number of system call parameters and whether the code being debugged is 16- or 32-bits.

`(subsystem)`
The iRMX OS layer for the system call.

`system call`
The name of the iRMX system call.

`parameters`
The parameter names that correspond to the stack values directly above them. The maximum parameters displayed are 24.

Examples

1. To display stack information, enter:

```
..vs
```

This display is for a 16-bit application:

```
        gate #0360
1908:07ca   0b08   1980   1ea8   1980   1980   0000   0b00   1908
1908:07da   19a0   0b20   0580   1ea8   1ea0   1ee8   0000   0000

(Nucleus) delete mailbox

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

In this display, the CALL instruction has a stack address `1908:07CA`.

This display is for a 32-bit application:

```
     gate #0360
1908:000007ca 00000b08  00001980  00001ea8  00001980  00001980 00000000
1908:000007e4 00000b00  00001908  000019a0  00000b20  00000580 00001ea8
1908:000007fd 00001ea0  00001ee8  00000000  00000000  00000000 00000000

(Nucleus) delete mailbox

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

In this display, the CALL instruction has a stack address `1908:000007CA`. 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 (`1980:00000B08`) to the exception code. Similarly, the `mbox` parameter signifies that the third word (`1EA8`) is the token for the mailbox being deleted.

2. For system calls that are not iRMX system calls, SDB displays a stack display and a message that the CALL instruction is not for an iRMX system call.

   This display is for a 16-bit application:

```
2908:07d0    2980    2980    0000    0600    2908    29a0    0020    1580
2908:07e0    27c8    27c8    25c8    25c8    25c8    25c8    25c8    25c8

Not a system CALL
```

   This display is for a 32-bit application:

```
2908:000007d0   00002980 00002980 00000000 00000600 00002908 000029a0
2908:000007e8   00000020 00001580 000027c8 000027c8 000025c8 000025c8
2908:00000800   000025c8 000025c8 000025c8 000025c8 00000000 00000000

Not a system CALL
```

3. If a call has more parameters than will fit on one line of the display, SDB automatically displays multiple lines of stack values. Corresponding multiple lines of parameter descriptions are displayed directly below the stack values.

This display is stack information for the Nucleus system call **create_job** in a 16-bit application:

```
gate #0310
27c8:0f9a   0158   20c8   0000   20c8   20c8   0000   0600   17c8
27c8:0faa   20e8   0028   0000   0000   20c8   00e0   2ff8   2ff8
27c8:0fd4   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 is stack information for the Nucleus system call **create_job** in a 32-bit application:

```
gate #0310
27c8:00000f9a  00000158 000020c8 00000000 000020c8 000020c8 00000000
27c8:00000fb4  00000600 000017c8 000020e8 00000028 00000000 00000000
27c8:00000fca  000020c8 000000e0 00002ff8 00002ff8 00002608 00001a58
27c8:00000fd4  00001af8 000000c8 00000000 00000000 00000000 00000000

 (Nucleus) create job


|....excep$p......|.t$flgs.|.stksze.|...sp...|...ss...|
                     |...ds...|...ip...|...cs...|...pri..|.j$flgs.|.exp$in-

fo$p....|.maxpri.|.maxtsk.|.maxobj.|.poolmx.|.poolmn.|
                     |.param..|.dirsiz.|
```

This display indicates that the CALL instruction is a Nucleus system call **create_job** 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.

4. This display shows that the CALL instruction is a BIOS system call **a_attach_file** with five parameters. The `subpath$p` parameter points to a string seven characters long: the word `'example'`.

This display is for a 16-bit application:

```
gate #0500
27c8:0f4e   0f88   17c8   25f8   0000   2600   29a0   0000   2600
27c8:0f5e   2608   1c10   2600   1320   26d0   0f78   0df8   2ff8

 (BIOS) attach file

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

This display is for a 32-bit application:

```
gate #0500
27c8:00000f4e  00000f88 000017c8 000025f8 00000000 00002600 000029a0
27c8:00000f66  00000000 00002600 00002608 00001c10 00002600 00001320
27c8:00000f7e  000026d0 00000f78 00000df8 00002ff8 00000000 00000000

(BIOS) attach file


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

5. This display is for a 32-bit flat-model application calling the Nucleus **lookup_object** system call. Notice that the pointer parameters (`except$p` and `name$p`) are only 32 bits long, unlike pointers in the previous 32-bit examples. All pointers in a flat-model application are near, or offset only.

```
gate #01b0
679b:00446e1c  00446fde 0000000a 004202a7 00000000 00010001 4d00001d
679b:00446e34  61737365 74206567 6c206f6f 00676e6f 746f7250 6c6f636f
679b:00446e4c  6f727720 7420676e 53415405 4f52324b 6b634552 42007465

(Nucleus) lookup object

               |except$p|..time..|.name$p.|..job...|
name--> 04'TEST'
```

Error Messages

`Syntax Error`
: You made an error entering the command.

`Not a system CALL`
: The CS:EIP is pointing to a CALL instruction that is not an iRMX system call.

`Unknown entry code`

Indicates SDB has mistaken an instruction operand for a FAR CALL instruction or that a software link from user code into iRMX code has been corrupted. To recover from system corruption, reboot the system.

vt

Displays information about an iRMX object.

Syntax

```
vt token
```

Parameter

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

Additional Information

The **vt** command determines the type of iRMX object represented by the token and displays information about that object. The information and the format in which SDB displays the information depends on the type of object.

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

- Jobs
- Tasks
- Mailboxes
- Semaphores
- Regions
- Segments
- Extensions
- Composite objects
- Buffer Pools

Job Display

If the parameter you specify is a valid job token, SDB displays information about the job having that token, as the sample display 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:xxxxxxxx  Except mode  xx        Parent job      xxxx
Pool min         xxxxxxxx       Pool max     xxxxxxxx  Initial size    xxxxxxxx
Borrowed         xxxxxxxx
JOB name        xxxxxxx
       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
```

The table in the lower half of the display is the Free Space Table.

The display fields from left to right are:

`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 set when the job was created with the Nucleus system call **create_job** or **rqe_create_job**.

`Entries used`
The number of objects now cataloged in the job's object directory.

`Job flags` The `job flags` parameter value that was set when the job was created. A value of 0 indicates the Nucleus does parameter checking for system calls made from the job. The value 2H indicates parameter checking is not done unless a parent job has parameter checking set.

`Except handler`
The start address of the job's exception handler. This address was set when the job was created.

`Except mode`
Indicates when control is to be passed to the new job's exception handler. This value was set when the job was created.

Value	**When Control Passes**
0	Never
1	On programmer errors
2	On environmental conditions
3	On all exceptions

`Parent job`
The token for the specified job's parent.

`Pool min` The minimum size of the job's memory pool, in 16-byte paragraphs. This value was set when the job was created.

`Pool max` The maximum size of the job's memory pool, in 16-byte paragraphs. This value was set when the job was created.

`Initial size`
The initial size of the job's memory pool, in 16-byte paragraphs.

`Borrowed` The current amount of memory that the job has borrowed from its ancestor(s), in 16-byte paragraphs.

`Job name` The name of the job as contained in the jobs object directory cataloged as R?HCNAME. If the object does not exist, has a length of zero, or is a null token, this field is not displayed.

`Free Space`

A table that displays the amount of free memory in a job's memory pool. Column one of the free space table shows the byte ranges for contiguous free memory. Column two shows the number of chunks or free memory units in a specific byte range. Column three displays the largest chunk or free memory unit in a specific byte range. Column four shows the total amount of free space in a specific byte range.

Task Display

SDB displays information about tasks in different ways for interrupt tasks and for non-interrupt tasks.

This sample shows the display for non-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:xxxxxxxx  Except mode     xx    Task flags      xx
K-saved SS:SP   xxxx:xxxxxxxx  Containing job  xxxx  Interrupt task  no
```

The display for interrupt tasks is the same as for non-interrupt tasks but with this added:

```
Int level      xx              Master mask    xx    Slave mask     xx
Pending int    xx              Max interrupts xx
```

These are the fields for both interrupt and non-interrupt displays, from left to right:

`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 the Nucleus system calls **create_job** or **rqe_create_job**.

`Dynamic pri`

A temporary priority that the Nucleus assigns to the task when it controls a region and a higher priority task wants control.

`Task state`

The state of the task. These are the twelve possible states, as they are displayed:

State	Description
ready	Task is ready for execution
asleep	Task is asleep
susp	Task is suspended
aslp/susp	Task is asleep and suspended
deleted	Task is being deleted

State	**Description**
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 a port on the transaction queue

If this field contains an invalid value, the value followed by a space and two question marks is displayed.

`Suspend depth`
The number of **suspend_task** Nucleus system calls that have been applied to this task without a corresponding **resume_task** Nucleus system call.

`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 the **sleep** Nucleus system call. 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 with the Nucleus system calls **create_task**, **create_job**, **rqe_create_job**, or **set_exception_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 with the Nucleus system calls **create_task**, **create_job**, **rqe_create_job**, or **set_exception_handler**.

`Task flags`
The `task flags` parameter used when the task was created with the Nucleus system call **create_task**. The value 1H indicates the task contains floating-point instructions; the value 0 indicates it does not.

`K-saved SS:SP`
The contents of the SS:SP registers when the task last left the ready state.

`Containing job`
The token of the job to which this task belongs.

`Interrupt task`
Indicates whether this task is an interrupt task.

`No` signifies that the task is not an interrupt task. If so, this is the last field in the display. See the sample display for non-interrupt tasks.

`Yes` signifies that the task is an interrupt task. In this case, additional fields appear in the display. See the sample display for interrupt tasks.

`Int level` The level that the interrupt task services. This level was set when this task called the Nucleus system call **set_interrupt**.

`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, master levels 6 and 7 are disabled, so the master mask field is 11000000B (0C0H).

See also: Interrupt levels, *System Concepts*

`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).

`Pending int`
The number of **signal_interrupt** Nucleus system calls pending for the interrupt level.

`Max interrupts`
The maximum number of **signal_interrupt** Nucleus system calls that can be pending for the interrupt level.

Mailbox Display

SDB displays information about mailboxes in four ways:

- When nothing is queued at the mailbox
- When tasks are queued at the mailbox
- When objects are queued at the mailbox
- When data messages are queued at the mailbox

This is the format of the display when nothing is 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
```

When there are tasks queued at the mailbox, this line is added:

```
Task queue      zzzz  xxxx  ...
```

When there are objects queued at the mailbox, this line is added:

```
Object cache queue      zzzz  xxxx  ...

Object overflow queue   xxxx  xxxx  ...
```

When there are data messages queued at the mailbox, this line is added:

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

The field descriptions follow:

`Mailbox type`
: The type of mailbox: object or data. This 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`.

`Data queue head`
: The pointer for the first data message at the head of the message queue.

`Queue discipline`
: Indicates how tasks are queued at the mailbox. Tasks are queued as

`FIFO` (first-in-first-out) or by `PRI` (priority). This is specified when a mailbox is created with the Nucleus system call **create_mailbox**. If the field cannot be interpreted, the actual value is displayed followed by a space and two question marks.

`Containing job`
: The token for the job that contains this mailbox.

`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. If the mailbox type is `Data`, this field contains `N/A`.

`Data message queue`
: Pointers for the data messages residing at the 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. The tokens are listed 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. The tokens are listed 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.

Semaphore Display

Semaphore information is displayed either when no tasks are queued or when tasks are queued.

This is the format of the display when no tasks are queued:

```
Object type = 4 Semaphore

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

When no tasks are queued, this line is added:

```
Task queue    xxxx  xxxx  ...
```

The field descriptions follow:

`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 the Nucleus system call **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.

`Containing job`
: The token for the job to which the semaphore belongs.

`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 token parameter is a valid token for a region, information about the region is displayed with or without a task queue.

This is the format for the display of a region with no task queue:

```
Object type = 5 Region

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

When a region that has a task queue, this line is added:

```
Task queue    xxxx  xxxx  ...
```

The field descriptions follow:

`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 the Nucleus system call **create_region**.

`Containing job`
: The token for the job to which the region belongs.

`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, this information is displayed:

```
Object type = 6 Segment

Segment size xxxxxxxx     Containing job    xxxx
```

The fields are:

`Segment size`
: The number of bytes in this segment. The size of the segment was specified when the segment was created with the Nucleus system call **create_segment**.

`Containing job`
: The token for the job to which the segment belongs.

Extension Object Display

If the **vt** parameter is a valid token for an extension, this information is displayed:

```
Object type = 7 Extension

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

The fields are:

`Extension type`
: The type code associated with composite objects licensed by this extension. This code was specified when the extension type was created with the Nucleus system call **create_extension**.

See also: Extension types, *System Concepts*

`Deletion mailbox`
: The token for the deletion mailbox associated with this extension. This mailbox was specified when the extension type was created.

`Containing job`
: The token for the job to which the extension belongs.

Composite Object Display

The **vt** command displays these 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, stream file, named file, and remote file connections
- Port connection

See also: EDOS file driver, *Driver Programming Concepts*

Display of Composite Objects Other Than BIOS or EDOS

```
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    ...
```

The field descriptions follow:

`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 the Nucleus system call **create_extension**.

`Extension obj`
: The token for the extension object used to create this composite object.

`Deletion mbox`
: The token for the mailbox the composite goes to when the composite is to be deleted. This mailbox was specified when the extension was created with the Nucleus system call **create_extension**.

`Containing job`
: The token for the job to which the composite object belongs.

`Num of entries`
: The number of component entries in the composite object.

`Component list`
: The list of tokens for the components of the composite.

Display of BIOS Composite User Object

```
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    ...
```

In addition to the fields in the non-BIOS composite display, this display contains these 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.

Display of BIOS Physical File Connection

```
   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    xxxxxx  File pointer
xxxxxxxx
   IORS cache    xxxx          File node     xxxx    Device desc    xxxx
   DUIB pointer  xxxx:xxxxxxxx Dynamic DUIB  xxxxx   Num conns      xxxx
   Num readers   xxxx          Num writers   xxxx    File share
xxxxxxx
   File drivers  xxxx          Device gran   xxxx    Device size
xxxxxxxx
   Device name   xx            Device functs xxxx    Num dev conn   xxxx
```

This display contains these fields in addition to those in the non-BIOS composite display:

`File driver`

The file driver to which this connection is attached. The five possible values are `Physical`, `Stream`, `Named`, `EDOS`, `DOS`, and `Remote`. If this field contains an invalid value, the value is displayed 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. This description shows the connection state when a bit (0 is the rightmost bit) is set to 1:

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

`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 dash (-) appears in the character position. The iRMX access rights and the characters that represent them are:

Character	Directory File Access
D	Delete
L	List
A	Add
C	Change

Character	Data File Access
D	Delete
R	Read
A	Append
U	Update

The DOS file access rights are limited to two options: read-only (R) and full access (DRAU). For directories, DOS automatically grants full access (DLAC).

`Open mode` The mode established when this connection was opened. These are the possible modes:

Open Mode	**Description**
Closed	Connection is closed
Read	Connection is open for reading
Write	Connection is open for writing (not valid for DOS/EDOS)
R/W	Connection is open for reading and writing

If this field contains an invalid value, the value is displayed followed by a space and two question marks.

`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). These are the possible modes for iRMX:

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

These are the possible modes for DOS/EDOS:

Share Mode	**Description**
Readers	File can be shared with readers
ALL	File can be shared with all users
0	Connection is not open

If this field contains an invalid value, the value is displayed 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 IORS list. These IORSs are being saved for the BIOS system call **wait_io** 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 operating system uses the device descriptor to maintain information about connections to a device.

`DUIB pointer`

The address of the DUIB for the device unit containing the file.

See also: DUIBs, *Driver Programming Concepts*

`Dynamic DUIB`

Indicates whether a DUIB was created dynamically when the device associated with this connection was attached.

`Num of conns`

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. These are the possible modes for iRMX:

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

Only Readers and All modes are available for DOS/EDOS.

If this field contains an invalid value, the value is displayed followed by a space and two question marks. This probably means the file's internal data structure or fnode is corrupted.

See also: Sharing Modes for Files and Connections, *System Concepts*

`File drivers`

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

Bit	**Driver**
5	EDOS
4	Remote
3	Named
2	DOS
1	Stream
0	Physical

`Device gran`

The granularity of the device, in bytes. 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 name`

The device name where this file is stored. The device name has a maximum size of 14 characters.

`Device size`

The capacity of the device, in bytes.

`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**
7	f_close
6	f_open
5	f_detach_dev
4	f_attach_dev
3	f_special
2	f_seek
1	f_write
0	f_read

`Num dev conn`

The number of connections to the device.

Display of BIOS Stream File Connection

```
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
DUIB pointer  xxxx:xxxxxxxx  Dynamic DUIB   xxxxx    Num conns     xxxx
Num readers   xxxx           Num writers    xxxx     File share
xxxxxxx
File drivers  xxxx           Device gran    xxxx     Device size
xxxxxxxx
Device name   Stream         Device functs  xxxx     Num dev conn  xxxx
Req queued    xxxx           Queued conn    xxxx     Open conn     xxxx
```

This display contains these fields in addition to those in the BIOS physical file connection display:

`Req queued` The number of requests queued at the stream file.

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

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

Display of BIOS Named File Connection

```
   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    xxxxxxx    File pointer
xxxxxxxx
IORS cache    xxxx           File node     xxxx       Device desc   xxxx
DUIB pointer  xxxx:xxxxxxxx  Dynamic DUIB  xxxx       Num conns     xxxx
Num readers   xxxx           Num writers   xxxx       File share    xxxx
File drivers  xxxx           Device gran   xxxx       Device size
xxxxxxxx
Device name   xxxxx          Device functs xxxx       Num dev conn  xxxx
Num buffers   xxxx           Fixed update  xxxx       Upd timeout   xxxx
```

```
Fnode PTR(s)  xxxx:xxxxxxxx  Fnode number  xxxx      Fnode flags   xxxx
Owner         xxxx           File type     xxxxxxxxx File/Vol gran xxxx
Total blocks  xxxxxxxx       Total size    xxxxxxxx  This size
xxxxxxxx
Volume gran   xxxx           Volume size   xxxxxxxx  Volume name   xxxxxx
```

This display contains these fields in addition to those in the BIOS physical file connection display:

`Num buffers`
: The number of buffers allocated for blocking and unblocking I/O requests involving the device. A value of 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.

`Upd timeout`
: The length of the time for the update timeout feature, measured in system clock ticks.

See also: Update timeout and fixed updating, *System Concepts*

`Fnode PTR(s)`
: The addresses of the fnode pointers.

`Fnode number`
: The fnode number of this file.

See also: Fnodes, *Command Reference*

`Fnode flags`
: A word containing flag bits. If a bit is set to 1, this description applies. Otherwise, the description does not apply.

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

`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 also: File ownership, *System Concepts*

`File type` The type of named file. These are the possible values:

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 value is displayed followed by a space and two question marks.

`File/Vol gran`
The granularity of the file in volume granularity units.

`Total blocks`
The total number of volume blocks used for the file at present including indirect blocks.

`Total size`
The total size of the file in bytes, including actual data only.

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

`Volume gran`
The granularity of the volume, in bytes.

`Volume size`
The size of the volume, in bytes.

`Volume name`
The name of the volume.

See also: Fnodes, *Command Reference*

Display of BIOS Remote File Connection

```
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    xxxxxxx File pointer   xxxxxxxx
IORS cache    xxxx           File node     xxxx    Device desc    xxxx
DUIB pointer  xxxx:xxxxxxxx Dynamic DUIB   xxxxx   Num conns      xxxx
Num readers   xxxx           Num writers   xxxx    File share     xxxxxxx
File drivers  xxxx           Device gran   xxxx    Device size    xxxxxxxx
Device name   xxxx           Device functs xxxx    Num dev conn   xxxx
```

This display contains the same fields as those in the BIOS physical file connection display except for the `File driver` field, which is `Remote` rather than `Physical`.

Display of BIOS EDOS File Connection

```
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    EDOS           Conn flags    xx         Access         xxxx
Open mode      xxxxxx         Open share    xxxxxxx    File pointer   xxxxxxxx
IORS cache     xxxx           File node     xxxx       Device desc    xxxx
DUIB pointer   xxxx:xxxxxxxx  Dynamic DUIB  xxxx       Num conns      xxxx
Num readers    xxxx           Num writers   xxxx       File share     xxxx
File drivers   xxxx           Device gran   xxxx       Device size    xxxxxxxx
Device name    xxxxx          Device functs xxxx       Num dev conn   xxxx
Num buffers    xxxx           Fixed update  xxxx       Upd timeout    xxxx
Owner          xxxx           File type     xxxxxxxxx File/Vol gran  xxxx
Total blocks   xxxxxxxx       Total size    xxxxxxxx   This size      xxxxxxxx
EDOS conn ID    xxxxxx
```

The fields are the same as for a BIOS named file connection, except that information about volumes is omitted and information about the EDOS connection ID is added. The display for DOS is similar, but not exactly the same.

Display of Service Object

```
Object type = 8  Composite
Extension type xxxx       Extension obj  xxxx        Deletion mbox  xxxx
Containing job xxxx       Num of entries xxxx

     T$SERVICE OBJECT:
Name: eth0
Service type: Generic     Service flags: xxxx
Address size: xxxx   Max control message size: xxxx
Port IDs: valid from 0001 to 8FFF, else if 0000 allocate from 9000 to 9FFF
Alloc/max control buffers: xxxx/xxxx   Alloc/max transactions: xxxx/xxxx
Output queue: head=xxxx:xxxxxxxx  tail=xxxx:xxxxxxxx  count=xxxx
Input queue:  head=xxxx:xxxxxxxx  tail=xxxx:xxxxxxxx  count=xxxx
Handler entry points:
Initialize: xxxx:xxxxxxxx GetAttribs: xxxx:xxxxxxxx CreatePort: xxxx:xxxxxxxx
SendMsg:    xxxx:xxxxxxxx SetAttribs: xxxx:xxxxxxxx DeletePort: xxxx:xxxxxxxx
Update:     xxxx:xxxxxxxx Validate:   xxxx:xxxxxxxx Cancel:     xxxx:xxxxxxxx
Service:    xxxx:xxxxxxxx GetFrag:    xxxx:xxxxxxxx Finish:     xxxx:xxxxxxxx
xxxx ports:
xxxx    yyyy
```

This display shows the fields of a service object

Name
: The service name

Service type
: This field shows the level of support provided by the Nucleus for this service. This can include stubs for interrupt services and port services.

Service flags
: This field shows the current service flags, indicating the support the Nucleus provides for the service.

Address size
: The size in bytes of a hardware address for this service

Max control message size
: The maximum size allowed by this service for control messages

Port IDs
: This shows how port IDs are validated and allocated for this service. In the above example, a port ID in the range 1 to 07FFFh is valid for the creation for a port with that ID. If a port ID of 0 is specified, then the service allocates an ID from the range 09000h to 09FFFh, if available.

Any other value specified when creating or binding a port will result ina error.

Alloc/max control buffers
: Shows the number of service control buffers currently allocated, and the total available to the service

Alloc/max transactions
: Shows the number of transaction buffers currently in use by the service, and the total available to the service.

Input queue
: Shows the address of the first and last transaction on the service input queue, and the number of transactions on the queue. This might indicate the number of input buffer requests pending at a hardware interface, for example.

Output queue
: Shows the address of the first and last transaction on the service output queue, and the number of transactions on the queue. This might indicate the number of transmit requests still pending for a given service, for example.

Handler entry points
: Each address shown is the address of one of the service handlers. If the service does not support a given handler, that address is shown as 0000:00000000.

Ports
: Shows the number of ports currently created for this service, and the tokens for those ports.

Display of a Generic Port

```
Object type = 8  Composite

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

T$PORT OBJECT:
Service object:    xxxx     Name: xxxxxxxxxxxxxx
Queue discipline   FIFO     Buffer pool    xxxx
Fragmentation      xxx      Max port transctns  xx   Sink port      xxxx
Source port id xxxx    Source address xx xx xx xx xx xx
Dest. port id  xxxx    Dest. address  xx xx xx xx xx xx
Task queue    xxxx    xxxx    ...
```

```
Message queue      xxxx:xxxx  xxxx:xxxx    ...

Transaction id  nn   Task token  xxxx
Transaction id  nn+1 Task token  xxxx
Transaction id  nn+2 Task token  xxxx
Transaction id  nn+3 Task token  xxxx
Transaction id  nn+4 Task token  xxxx
Transaction id  nn+5 Task token  xxxx
Transaction id  nn+6 Task token  xxxx
Transaction id  nn+7 Task token  xxxx
Transaction id  nn+8 Task token  xxxx
Transaction id  nn+9 Task token  xxxx
```

This display shows the fields of a generic (non-Nucleus Communications Service) port.

Service object
: The token for the service where this port was created, or 0000 for a sink port.

Name
: The name of the service where the port was created, if not a sink port

Queue discipline
: The queue discipline for the task queue on this port.

Fragmentation
: The state of the allow-fragmentation flag on this port.

Max port transctns
: The size of the transaction queue on this port.

Sink port
: The token for the sink port for this port, if any.

Source port id
: The port id for this port

Source address
: For an addressed service, displays the address of this port.

Dest. port id
: If the port has been connected, shows the destination port id.

Dest. address
: If the port has been connected and the service processes addresses, shows the destination address.

Task queue
: A list of task tokens showing the tasks currently queued for messages.

Message queue
: A list of address of control messages currently queued on the port. Note that **either** the task queue **or** the message queue will be displayed.

Transaction id
: The transaction id of an outstanding RSVP transaction

Task token
: The token of the task queued on this transaction.

Display of a Signal Port (Nucleus Communications Service only)

```
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    ...
```

This format uses the composite display as a base and appends these fields:

`Protocol type`
: The message protocol. This value is `Signal` to indicate signal service. The type is determined when the port is created through the Nucleus system call **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 this was specified with the Nucleus system call **create_port**. If this field is uninterpretable, the actual byte value is displayed followed by a space and two question marks.

`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 specified in the `message$id` field using the Nucleus system call **create_port**.

`Task queue`
: The tokens for the list of tasks, if any, queued at the port.

Display of a Data Port (Nucleus Communications Service only) With Messages Queued

```
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    ...
```

The display format for a data port with tasks queued is the same as when messages are queued with these differences:

```
Task queue           xxxx  xxxx
```

The preceding two port formats use the composite display as a base and append these 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 the Nucleus system call **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.

`Buffer pool`
: The token of the attached buffer pool (if any). The Nucleus system call **attach_buffer_pool** attaches a buffer pool to a port.

`Fragmentation`
: `Yes` or `No` indicates whether the port can handle request message fragmentation. This is defined when the port was created.

`Max Port Transctns`
: The maximum number of simultaneous outstanding transactions for the port. This limitation is defined when the port is created.

`Sink port`
: The token of the sink port (if any) associated with the port. Sink ports are connected to ports through the Nucleus system call **attach_port**.

`Destination msg id`
: The `host_id` portion of the socket identifying the remote port to which this port is connected. This value is established through the Nucleus system call **connect**.

`Destination port id`
: The `port_id` portion of the socket identifying the remote port to which this port is connected. This value is established through the Nucleus system call **connect**.

`Source port id`
: The port ID number for this port. The number is established through the `port_id` field when the port is created.

`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 to 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.

`Task queue`
: The list of pointers representing the tasks queued at this port. This field appears only if the port has queued tasks.

In addition, the **vt** output for a data port can appear with these 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

Heap Display

```
Object type = 10  Heap

Containing job      xxxx   Total size  xxxxxxxx  Kernel Pool  xxxxxxxx
Pool size: xxxxxxxx    Pool available: xxxxxxxx  Pool largest area: xxxxxxxx
             Process Mappings
```

Displays the features of a heap object.

Total size
: The size of the segment containing the pool

Kernel pool
: The token for the associated RMK pool

Pool size
: The size of the RMK pool

Pool available
: The memory currently available in the pool

Pool largest area
: The largest remaining area in the pool.

Process mappings
: If the paging job is enabled, then flat-model applications can map the pool to their virtual segment in order to allocate buffers. The list of job VSEG mappings is displayed here.

Buffer Pool Display

If the parameter that you supply is a valid token for a buffer pool, the information about the buffer pool is:

```
Object type = 10  Buffer pool

Max buffers    xxxx  Total buffer count    xxxx  Total size count      xx
Containing job xxxx  Data Chaining          xxx

Buffer pool contents:
Buffer size   xxxx    Buffer count   xxxx
Buffer size   xxxx    Buffer count   xxxx
                    .
                    .
                    .
```

The fields in the buffer pool display are:

`Max buffers`
The total number of buffers allowed in this buffer pool. This maximum value is determined when the buffer pool is created using the Nucleus system call **create_buffer_pool**.

`Total buffer count`
The number of buffers now in the buffer pool. This number is equivalent to the number of buffers released to the buffer pool using the Nucleus system call **release_buffer**.

`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 the Nucleus system call **create_segment** and released to the buffer pool.

`Buffer count`
The number of buffers that are of the buffer size displayed in the field directly to the left.

Error Messages

`Syntax Error`
You did not specify a parameter for the command or you made an error entering the command.

`*** INVALID TOKEN ***`
You entered a value that is not a valid token.

See also: Using Tokens as Command Parameters, Chapter 1

vu

Displays the iRMX system calls in a task's stack.

Syntax

```
vu task-token
```

Parameter

`task-token` The token of the task whose stack will be searched for system calls.

Additional Information

This command searches a task's stack for iRMX system calls, starting at the top of the stack. The task’s stack must be inside an iRMX segment. For each system call it finds, **vu** 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 parameters of the system call. They are shown as if the CALL instruction for the system call were in the CS:EIP register and the displayed stack values were at the top of the stack.

If no system calls are found on the stack, vu attempts to find possible return CS:EIP values. There are two possible displays for this situation; the second can list one or more possible return values:

```
No system calls on stack
Return cs:eip - xxx:xxx
```

or

```
No system calls on stack
Possible Return cs:eip(s) - xxx:xxx
                            xxx:xxx
                            xxx:xxx
```

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

System calls can be nested, so some invocations of the **vu** command produce multiple displays of the type shown here. This is the display format for the **vu** system call information:

```
gate #NNNN

Return cs:eip - yyyy:yyyyyyyy
xxxx:xxxxxxxx   xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
xxxxxxxx
xxxx:xxxxxxxx   xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
xxxxxxxx
xxxx:xxxxxxxx   xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
xxxxxxxx

(subsystem)  system call

             |parameters|
```

The fields are as follows:

`gate #NNNN`
: The gate number associated with the system call.

`Return cs:eip`
: The return address for the system call of this display (`yyyy:yyyyyyyy`).

`xxxx:xxxxxxxx`
: The address of the stack portion devoted to this call.

`xxxxxxxx`
: Values now on the stack.

`(subsystem)`
: The iRMX OS 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.

Example

This example shows how the **vu** command responds when system calls are nested. The task for the example has called the EIOS system call **s_write_move**. **S_write_move** has called the BIOS system call **a_write**. **A_write** has called Nucleus system call **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 SDB and invoke the **vu** command:

```
..vu 21c8
```

This display is for a 16-bit application:

```
gate #0430

Return cs:ip -09b8:576a
2168:01b2   01c8   2168   01c8   2168   ffff   1768   1760   1988
2168:01c2   1550   0000   2148   1ff8   1440   2558   2000   2050

(Nucleus) receive message

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

gate #05B0

Return cs:ip -09d8:08e7
2168:01d4   01e8   2168   1f58   0400   0000   20e8   2098   2088
2168:01e4   1430   2048   01f8   20f8   1400   0218   0000   01f8

(BIOS) write

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

gate #0710

Return cs:ip -09f8:06fa
2168:0218   0020   19f0   0400   0030   19f0   2098   2080   2140
2168:0228   2058   0000   0000   20c8   20c8   20c8   20c8   20c8

(EIOS) write move

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

This display is for a 32-bit application:

```
gate #0430

Return cs:eip -09b8:0000576a
2168:000001b2  000001c8 00002168 000001c8 00002168 0000ffff 00001768
2168:000001da  00001760 00001988 00001550 00000000 00002148 00001ff8
2168:000001e2  00001440 00002558 00002000 00002050 00000000 00000000

(Nucleus) receive message

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

gate #05b0

Return cs:eip -09d8:000008e7
2168:000001d4  000001e8 00002168 00001f58 00000400 00000000 000020e8
2168:000001fa  00002098 00002088 00001430 00002048 000001f8 000020f8
2168:00000204  00001400 00000218 00000000 000001f8 00000000 00000000

(BIOS) write

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

gate #0710

Return cs:eip -09f8:000006fa
2168:00000218  00000020 000019f0 00000400 00000030 000019f0 00002098
2168:00000230  00002080 00002140 00002058 00000000 00000000 000020c8
2168:00000248  000020c8 000020c8 000020c8 000020c8 00000000 00000000

(EIOS) write move

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

Note

For a flat-model application, the parameters to system calls are displayed in reverse order. For example, if the **rq_write_move** system call displayed above was made from a flat-model application, the **vu** command would display the `conn` parameter first and the `excep$p` parameter last.

This difference in displays occurs because of the way the interface to the OS handles flat model applications. You do not invoke system calls any differently from a flat-model application.

Note

For a flat-model application, the pointer parameters are only 32 bits long, unlike pointers in the previous 32-bit examples. All pointers in a flat-model application are near, or offset only.

Error Messages

`Syntax Error`
: You made an error entering the command.

`*** INVALID TASK TOKEN ***`
: You entered a value that 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`
: You entered a value that is not a valid task token.

System Debug Monitor (SDM) Commands 3

This chapter is a reference for SDM commands. It includes command descriptions, parameter descriptions, and examples of commands. Most SDM commands have command characters that suggest the function of the command. For example, the **d** command displays information and the **s** command substitutes a value. Table 3-1 lists the SDM commands in alphabetical order.

Table 3-1. SDM Commands

Command	Description
bc (breakpoint clear)	Clears one or all breakpoints set previously with the **bs** command
bs (breakpoint set)	Sets one or more software or hardware breakpoints, which remain set until specifically cleared with the **bc** command
c (compare)	Compares the contents of one block of memory with another block
d (display)	Displays the contents of memory or a descriptor table
f (find)	Searches the specified block of memory to find a sequence of hexadecimal digits
g (go)	Begins executing the application program, with the option of setting one or more hardware breakpoints
i (input)	Inputs and displays a byte, halfword or word from a port
n (step)	Disassembles and executes one instruction at a time
m (memory copy)	Copies the contents of one block of memory to another
o (output)	Outputs a byte, halfword or word to a port
pd* (paging display)	A group of commands that display information about the paging subsystem
ps* (paging substitute)	Two commands that display and let you modify fields in the page directory and page tables
s (substitute)	Displays and lets you modify memory locations or components of the descriptor table entries
x (exchange)	Displays and lets you modify contents of registers or fields in a task state segment

Command Structure

This is the general structure of SDM commands:

```
[count] command [parameters]
```

Some commands have one or more parameters to indicate:

- Addresses
- Data
- Register names
- Punctuation symbols
- Numeric expressions
- Count

Though most parameters follow the command, the count parameters prefix the command character. `Count` indicates how many times you want SDM to repeat the command or how many consecutive bytes, halfwords, or words you want SDM to access. In some commands you can use `count` to control how many instructions SDM is to single step (go through line-by-line), which allows you to check your application for problems in each instruction.

Entering Commands

This section describes the conventions to use when you enter SDM commands. It also describes the SDM line editing features.

Command Line Conventions

When entering SDM commands, follow these conventions:

- You can enter commands in either upper- or lowercase characters.
- You can include spaces anywhere in the command line except within terms.
- You can enter command lines up to 128 characters long. If you exceed this limit, the terminal beeps and the command does not execute.

Command-Editing Keys

You can use these keys to edit the SDM command line:

Carriage Return
: Tells SDM to read the command line and execute the instruction.

Rubout or Backspace
: Deletes the character you entered most recently. It deletes the character from both the command line and the display. If you attempt to rubout the prompt, the terminal issues a beep.

<Ctrl-C> Aborts the current command and issues a prompt. However, if your application is running and it is in a loop, <Ctrl-C> has no effect.

<Ctrl-S> Suspends SDM's console output. SDM does not lose any output .

<Ctrl-Q> Resumes the console output suspended by <Ctrl-S>.

<Ctrl-X> Deletes the current command line.

Command Line History

To repeat an earlier command without retyping it, enter <Ctrl-B> repeatedly to scroll back through previous commands. Enter <Ctrl-F> to scroll forward in the list. On a PC keyboard, you can also use the <UpArrow> and <DownArrow> keys for this purpose.

Once you have displayed an earlier command in this way, you can edit it if you choose. Then execute the command by pressing <CR>.

Multiple Commands on a Single Line

There are three ways to designate multiple commands on a single line:

semicolon (;)	A semicolon between commands lets you put more than one type of command on a single line
n < command >	A decimal number and angle brackets lets you specify *n* repetitions of a single command on one line
comma (,)	A comma at the end of a command line lets you repeat the command line indefinitely

You can use one or a combination of these methods on a command line.

Combining Commands

Execute multiple commands on a single command line by separating them with semicolons (;). For example:

```
..g cs:3b7 ; d ds:4a
```

This executes the **g** command followed by the **d** command.

Repeating Commands

Repeat a command by preceding the command with a decimal number and enclosing the command in angle brackets. For example:

```
..12 <g, cs:3b7>
```

This executes the **g** command 12 times.

You can nest brackets up to three levels. By combining brackets with semicolons you can repeatedly execute multiple commands. For example:

```
..5 <12 <g, cs:3b7> ; d ds:4a>
```

This repeats the **g** command 60 times and the **d** command 5 times. The order of the commands is twelve **g** commands followed by one **d** command, then twelve more **g** commands and one **d** command, and so on.

SDM interprets repeat factors (those entered at the beginning of a command) as decimal numbers. This is the only case where you do not use a `T` suffix after a decimal number.

Continuing Commands

To execute a command or group of commands more than once, include a comma after the command or after the closing angle bracket in the command line. For example:

```
..10 <5 dw>,
```

SDM executes the **d** command 50 times and then displays the command followed by a dash prompt (-):

```
..10 <5 dw> -
```

If you enter another comma, the command line is executed again. If you do not want to execute the command line again, enter a <CR>.

Command Parameters

SDM commands have these parameter types:

- Byte, 16-bit halfword and 32-bit word parameters
- Term parameters
- Expression parameters
- Address parameters
- Numeric parameters

Byte, Halfword and Word Parameters

Byte parameters are 8-bit parameters, halfword parameters are 16-bit parameters, and word parameters are 32-bit parameters. These parameters can be hexadecimal numbers, decimal numbers, terms, expressions, registers, or ranges. SDM assumes that all byte, halfword and word values are hexadecimal. Omit the `H` suffix from hexadecimal values. To specify a decimal value, enter a `T` suffix immediately following the number.

Byte values range from 00H to 0FFH. Halfword values range from 0000H to 0FFFFH. Word values range from 00000000H to 0FFFFFFFFH. For halfword or word parameter values, enter the high byte first, then the low byte. For halfword and word value displays, the high byte value appears first followed by the low byte value. However, the system stores halfword and word values in memory with the low byte followed by the high byte.

These examples show how SDM displays byte, halfword, and word values. For example, the byte values C4, 26, F2, and 3D are in consecutive locations beginning at the address 2468:26. SDM displays these locations in bytes:

```
2468:00000026  C4  26  F2  3D
```

SDM displays these same locations in halfwords:

```
2468:00000026  26C4  3DF2
```

SDM displays these same locations in words:

```
2468:00000026  3DF226C4
```

Term Parameters

A term parameter is either a `number` or a `register`.

Where:

`number` A hexadecimal or decimal number. Valid values range from 0000 to 0FFFFFFFFH. For a decimal number, enter a `T` suffix.

`register` An abbreviation for any CPU register. Table 3-2 shows the registers for the Intel386, Intel486, and Pentium microprocessors.

Table 3-2. CPU Registers (Protected Mode)

	Abbreviation			
Register Name	**32-bit**	**16-bit**	**8-bit**	
General Registers	EAX	AX	AH	AL
	EBX	BX	BH	BL
	ECX	CX	CH	CL
	EDX	DX	DH	DL
	EBP	BP		
	ESI	SI		
	EDI	DI		
Stack Pointer	ESP	SP		
Segment Registers				
Code	CS			
Data Segment	DS			
Stack Segment	SS			
Extra Data Segments	ES			
	FS			
	GS			
Flag Register	EFL*	FL*		
Instruction Pointer	EIP*	IP*		
Control Registers	CR0*	MSW*		
	CR2*			
	CR3*			
	CR4* †			

* You cannot use these registers as terms. They are included here because you can use them in commands that use registers as parameters.

† Available only on the Pentium microprocessor

Expression Parameters

An expression is a combination of terms that includes one or more signed or unsigned values.

```
expression term [+ | - term] [...]
```

Where:

`term` A number or a register.

Address Parameter

An address parameter consists of a segment selector (base) and an offset:

```
segment-selector:offset
```

Where:

`segment-selector`
Segment selector with a range from 0000H to 0FFFFH.

`offset` The offset into the selected segment with a range from 0000H to 0FFFFFFFFH.

If you do not include an address in a command that requires one, SDM usually uses the contents of the data segment register (DS) as the selector and 0 as the offset.

See also: Detailed command descriptions for exceptions

Numeric Parameters

You can use SDM to access the Numeric Processor Extension (NPX), which is either a separate math coprocessor or a floating point unit built into the microprocessor. You can access eight NPX stack registers, the status word, the control word, the tag word, the instruction pointer, the data pointer, and the instruction opcode. Use the abbreviations listed in Table 3-3 to reference an NPX register in a command.

Table 3-3. NPX Registers

Register Name	Abbreviation	Register Name	Abbreviation
NPX State	N	Status Word	SW
Control Word	CW	Tag Word	TW
Instruction Pointer	IP	Data Pointer	DP
Instruction Opcode	OP	Stack Register 0	ST(0)
Stack Register 1	ST(1)	Stack Register 2	ST(2)
Stack Register 3	ST(3)	Stack Register 4	ST(4)
Stack Register 5	ST(5)	Stack Register 6	ST(6)
Stack Register 7	ST(7)		

Numeric parameters are data types that the NPX supports. The three types of numeric parameters are:

Integer — An integer type with three subtypes called word integer, short integer, and long integer.

Real — A real type with three subtypes called short real, long real, and temporary real.

BCD — Packed binary coded decimal.

The suffixes that you use when you enter NPX data types are different from the suffixes for byte, 16-bit halfword and 32-bit word parameters. If you do not enter a suffix after an NPX data type, SDM assumes the number is decimal.

Table 3-4 lists the NPX data types and describes their characteristics. All numbers in this table are listed in decimal form.

Table 3-4. NPX Data Types

Data Type	Explicit Suffix	Bits	Significant Digits	Approximate Range
Word Integer	H	16	4	$-32,768 < X < +32,767$
Short Integer	H	32	10	$-2x10^{9} < X < +2x10^{9}$
Long Integer	H	64	19	$-9x10^{18} < X < +9x10^{18}$
Short real	R	32	6-7	$8.43x10^{-37}$ |X| $3.37x10^{38}$
Long real	R	64	15-16	$4.19x10^{-307}$ |X| $1.67x10^{308}$
Temporary real	R	80	19	$3.4x10^{-4929}$ |X| $1.2x10^{4929}$
Packed Decimal (BCD)	H	80	18	$-99...99 < X < +99...99$ (17 digits + sign digit)

See also: Programmer's Reference Manual or User's Guide for your math coprocessor or floating-point unit.

NPX Integers

An NPX integer is either a signed whole number or a hexadecimal number followed by an `H` suffix that SDM can interpret as an integer. Table 3-5 lists the NPX integer types.

Table 3-5. NPX Integer Types

Data type	Example
signed whole number	12, -12
hexadecimal	4E2H

The numbers 1.375, -4.6, and 9.2E3 are not valid NPX integers.

If you enter a hexadecimal number with the trailing `H`, SDM places the number into memory exactly as you entered it at the console.

SDM recognizes three types of integers: word, short, and long. Table 3-4 includes the range of integer values.

NPX Real Numbers

SDM recognizes three types of real numbers: short, long, and temporary. The differences between these real number types are:

- The number of bits
- The number of significant decimal digits
- The range of decimal numbers

See also: Real numbers, Table 3-4

An NPX real number can be represented as a signed decimal number, a scientific number, or a hexadecimal followed by an `R` suffix. Table 3-6 lists the NPX real number types.

Table 3-6. NPX Real Types

Data type	Example
signed decimal number	12.454, -12.454
scientific number	3E2 same as 300 4E-3 same as .0040
hexadecimal number	3FF2R

A scientific number consists of a signed decimal number's significant digits times base 10 raised to an exponential power. Exponents range from -4930 to 4930. For example, the decimal number 300 in scientific notation is 3.0×10^2. Enter this number as `3E2` where `E` stands for exponent. Another example is the number .0040. Enter this number as `4E-3`.

Packed Binary Coded Decimal (BCD) Numbers

An NPX BCD number is either a signed decimal number or a hexadecimal number followed by an `H` suffix. Table 3-4 includes the range of BCD numbers.

NPX Number Format

SDM displays NPX data types by showing, in this order:

1. The memory address of the data type
2. The data type in hexadecimal
3. The decimal equivalent of the data type, if it has one

For example, SDM displays the long integer 11223344 as:

```
1111:0  0000000000AB4130H     11223344
```

The long real number 11223344 is displayed as:

```
1111:0  4165682600000000R     11223344
```

The BCD number 11223344 is displayed as:

```
1111:0  000000000000011223344T     11223344
```

When SDM displays data types, the contents of the most significant byte of the numeric memory location is in the left-most position of the hexadecimal display. The rest of the bytes follow in order of decreasing significance. SDM displays the least significant byte in the right-most position of the hexadecimal display.

When you enter NPX data types, in either hexadecimal or decimal form, enter the most significant digit first and the rest of the digits in order of decreasing significance. If you enter a number that has a smaller number of significant digits than that of the NPX data type's, SDM appends leading zeroes.

Decimal Values

SDM displays decimal values in four different ways depending on the number's range and value.

- Decimal numbers that are exact integers and under 12 digits long are displayed as integers with no trailing decimal point or additional zeroes. An example of a long real number display is:

```
0080:00000000 4206FEE0E1A80000R       12345678901
```

- Decimal values that are within the field size and not exact integers, but close to an integer, are displayed in the form `XXXXX.0`. The suffix `.0` indicates the value is close to but not exactly an integer. An example of this type of display is:

```
0080:00000000  41D26580B4CCCCCDR      1234567891.0
```

- Decimal numbers with a magnitude greater than or equal to 0.1 and less than 10^{12} are displayed in the form `XXXX.XXXX`. An example of this type of display is:

```
0080:00000000  40FE240CA0275254R     123456.7891
```

- Very large and very small decimal numbers are displayed in scientific notation. This format is `X.XXXXEY` where `X.XXXX` are the significant digits in the number, `E` is a notation that means the number which follows (`Y`) is an exponent. An example follows:

```
0080:00000000  492C2916217B84B7R     3.14E+44
```

Nonnumeric Values

If the value in memory is not numeric, SDM displays the memory address followed by the hexadecimal form of the value, the sign of the number (`+` or `-`), and either `NAN` (Not-A-Number) or `Infinity`. These nonnumeric values appear in place of a decimal equivalent of the value. Examples follow:

```
0080:00000000  FFFF000000000000R    -NAN
0080:00000008  7FF0000000000000R    +Infinity
```

Special-Case Numeric Values

SDM identifies these special-case numeric values:

- Pseudo-zero values such as negative zeroes and zero fractions with non-zero exponents.
- Unnormalized numbers, which are numbers containing a 0 in the integer bit. The integer bit is the most significant bit of the significand, and serves as the implicit decimal point.

See also: Special computational situations, *80387 Programmer's Reference Manual*

SDM displays pseudo-zero values as `-0` (negative zero) and in the form `0Eexp` where `exp` is the base 10 power equivalent of the binary exponent in the number.

SDM displays unnormalized numbers in NPX number format with a bit value rather than a decimal equivalent. An unnormalized number has a 0 in the integer bit. The NPX normalizes a number by converting it to scientific notation. This shifts the most significant bit to the left until a 1 is in the integer bit, and decrements the exponent by 1 for each shift left. After the number is normalized, the integer bit is implicit and is not actually stored. This is an example of a display using an unnormalized number:

```
0080:00000000  3FFF1999999999999999AR     .2 UNNORM 3 BITS
```

The `3 BITS` field indicates that the unnormalized number must be shifted three bits to the left to normalize it. The `.2` field is the decimal value of the number if it were normalized.

Error Messages

SDM displays error messages if the command is invalid or impossible to execute. If the command line that contains the errors consists of multiple commands, SDM executes any valid commands prior to the command that caused the error. Table 3-7 lists SDM error messages.

Table 3-7. Error Messages

Error Message	Explanation
Bad Command	You entered an invalid command.
Invalid NPX Number Format	You entered an NPX value incorrectly.
Mismatched < >	You entered an opening angle bracket (<) but omitted the closing angle bracket (>).
NPX Exponent is Out of Range	The exponent of the NPX number you entered is not in the range of -4930 exponent 4930.
NPX Not Available	You tried to access an NPX value when the system does not include a math coprocessor.
Syntax Error	You entered a command incorrectly.
Too Many Digits in NPX Number	You entered an NPX number (in decimal form) with more than 19 digits.
Undefined Command Extension	You issued the V extension command without initializing SDB.
Illegal Selector	An operation caused a reference to an invalid selector value. For example, an attempt was made to display registers with the TR register uninitialized.
No xxxx Component	You specified a component in a register or descriptor table entry that is invalid.
Outside Segment	An operation exceeded the limit of a segment. For example, an attempt was made to display a memory location that is beyond segment bounds or a program was executed with less than 12 bytes of level 0 stack available to SDM.

bc

Clears one or all previously set hardware or software breakpoints. Use the **bc** command to remove breakpoints set with the **bs** command.

Syntax

```
bc [address]
```

Parameters

`address`

The address of a specific breakpoint to clear. If you don't specify an address, the **bc** command clears all breakpoints. Use the linear or I/O address of a previously set breakpoint. Physical addresses are not allowed. In non-flat model iRMX applications linear addresses are always the same as physical addresses, but you must use the character l after the address to specify that it is linear.

Additional Information

The **bc** command clears both hardware and software breakpoints. When clearing a hardware breakpoint, it immediately updates the debug registers of the CPU. If you clear all breakpoints, **bc** does not display any information. If you clear a single breakpoint by specifying its address, **bc** displays the reason code assigned to that breakpoint when it was set. The reason code is a 32-bit hexadecimal number in the range of 200h-203h for hardware breakpoints or 1000h-101Fh for software breakpoints.

See also: **bs** command for details about breakpoint types and reason codes

The **bc** command returns no output if you try to clear a single breakpoint using a valid address that is not a breakpoint address. No breakpoints are cleared.

Clearing Redundant Breakpoints

You cannot set multiple software breakpoints at the same address. However, if you have set more than one hardware breakpoint at the same address you must clear each individually, or clear all breakpoints. Hardware breakpoints at the same address are cleared from lowest to highest numbered reason code.

To individually clear hardware and software breakpoints set at the same address, issue a **bc** `address` command for each breakpoint. SDM clears the software breakpoint on the first command and clears the hardware breakpoint on the second.

Examples

Suppose you have set a software breakpoint and a hardware breakpoint at the same address. When you display breakpoints with the **bs** command, those breakpoints might be listed as follows:

```
..bs
00000203 h 0000104e m b
00001000 s 0000104e
..
```

The first breakpoint is a hardware modify breakpoint with reason code 203H. The second is a software execution breakpoint with reason code 1000H. To clear the hardware breakpoint, you must first clear the software breakpoint.

When you issue the **bc** command for this address, SDM clears the software breakpoint and returns the reason code followed by the prompt (..):

```
..bc 104e l
00001000
..
```

When you issue a second **bc** command for this address, SDM clears the hardware breakpoint:

```
..bc 104e l
00000203
..
```

bs

Sets new software or hardware breakpoints, or displays the current breakpoints. You can set breakpoints in conventional memory or at an I/O address. To display current breakpoints, issue the **bs** command without any parameters.

Syntax

```
bs [address]

bs address e

bs address m b|h|w

bs address w b|h|w

bs address i b|h
```

Parameters

`address`

An address for the breakpoint. If you do not follow the address with any other parameters, **bs** sets a software execution breakpoint.

When you set a breakpoint in conventional memory (either a software breakpoint or a hardware breakpoint using an `e`, `m`, or `w` parameter), you should typically specify a segmented address such as `cs:offset`. You cannot specify a physical address for conventional memory. You can specify a linear address by entering a hexadecimal value followed by the character l. However, be aware that in a flat-model application using virtual memory, a linear address is not the same as a physical address. In that case, SDM will interpret the linear address using the page table mechanism to set the breakpoint at the correct location.

When you set an I/O breakpoint with the `i` parameter, specify a hexadecimal physical address in the range 0 - 0FFFFH. Do not follow the address with the letter h or l.

`e` Execution breakpoint: sets a hardware breakpoint to occur when the instruction at the specified address is ready to be executed. The address must be on an instruction boundary.

`m` Modify breakpoint: sets a hardware breakpoint to occur following a read or write at the specified address.

`w` Write breakpoint: sets a hardware breakpoint to occur following a write at the specified address.

`i` I/O breakpoint: sets a hardware breakpoint to occur following a read or write at the specified port address.

`b` Specifies that the breakpoint applies to one byte.

`h` Specifies that the breakpoint applies to a halfword (16 bits). The address must be aligned on the appropriate halfword boundary.

`w` Specifies that the breakpoint applies to a word (32 bits). The address must be aligned on the appropriate word boundary.

Additional Information

A breakpoint is a condition that you set to return control to SDM. When you issue the **g** (go) command, SDM returns execution to the program you are debugging. If you have breakpoints set in the program, execution continues until the first breakpoint is encountered, then control returns to SDM. At that point you can use SDM commands to examine or change the state of the program (processor registers, memory, etc.).

A software breakpoint is one that SDM manages itself by inserting an Interrupt 3 at a specified address. When you execute code at that address, the interrupt returns control to SDM, which then removes the interrupt and reinserts the original code. You can set up to 32 software breakpoints with the **bs** command.

A hardware breakpoint is managed by one of the debug registers in the processor. You can set a total of 4 hardware breakpoints, using a combination of the **g** and **bs** commands:

- Any breakpoints that you enter on the **g** command line are automatically hardware execution breakpoints. The breakpoints must be set in code to be effective. When the **g** command executes to the point where the first breakpoint is reached, control returns to SDM. At that point, SDM removes all breakpoints you specified on the **g** command line, freeing them for further use.
- Hardware breakpoints that you set with the **bs** command can be in code (execution breakpoints), data (modify or write breakpoints), or I/O space (modify breakpoints at a port address). The only way you can clear breakpoints set with the **bs** command is to use the **bc** command. This applies to both software and hardware breakpoints.

You can set multiple hardware breakpoints at the same address, or set a software breakpoint at the same address as a hardware breakpoint. However, you cannot set multiple software breakpoints at the same address.

➠ **Note**

If you intend to set breakpoints in the **g** command, you must limit the hardware breakpoints set with **bs** to fewer than four. For example, if you intend to set two breakpoints on the **g** command line, you can only set two hardware breakpoints with **bs**. Then when you issue the **g** command with two breakpoints, the total of four hardware breakpoints is used. The **n** command with the **p** option also uses a hardware breakpoint, so you can set a maximum of three hardware breakpoints with **bs** before issuing an **np** or **npr** command.

The breakpoints set with **bs** remain set until you specifically clear them with **bc**. Breakpoints set on the **g** command line are cleared the next time you enter SDM, regardless of how you enter. For example, assume that you set one breakpoint with **bs** and two others in a **g** command. It doesn't matter which breakpoint causes the break to SDM; at the next SDM prompt the breakpoint set with **bs** will still be set, but the two set in the **g** command will be cleared.

Reason Codes

SDM assigns a number to each breakpoint when the breakpoint is set. This number is called a reason code. Software breakpoints are assigned reason codes 1000H - 101FH. Hardware breakpoints are numbered 200h through 203h. When you set a breakpoint or display breakpoints, the **bs** command displays the reason code, which is a 32-bit number in hexadecimal format. The **bc** command displays the same reason code when you clear an individual breakpoint.

No Breaks Available

If all the available breakpoints of one type (hardware or software) are set and you attempt to set another breakpoint of that type, SDM displays the message "No Breaks Available." You must clear a previous breakpoint with the **bc** command to be able to set a new breakpoint.

Software Breakpoints

You can set as many as 32 software breakpoints. Software breakpoints are execution breakpoints: a break occurs when the code executes to the specified address. The instruction at the address is not executed.

CAUTION

When setting a software breakpoint, make sure the address is the first byte of an instruction, including prefixes. Software breakpoints temporarily replace the byte at the specified address with an interrupt instruction. If the address does not point to an instruction boundary, the substituted instruction never causes a break. Instead, your program contains a byte which is interpreted as an unintended command or data. The result is unpredictable and the program could crash.

Hardware Breakpoints

When you set a hardware breakpoint, the monitor changes the contents of the debug registers in the processor. The registers affected are DR0, DR1, DR2, DR3, and DR7. Register DR6 is set to 0 each time the monitor relinquishes control to your program.

See also: Debug registers in the reference manual for your Intel386, Intel486, or Pentium microprocessor

There are four different types of hardware breakpoints: execution, modify, write, and I/O. You selected them with an `e`, `m`, `w`, or `i` parameter.

Execution Breakpoints

A break caused by an execution-type breakpoint (parameter `e`) occurs only when the address on the command line is an instruction to be executed. The break occurs before the instruction is executed.

It is important that you set the address correctly for an execution-type breakpoint. To cause a break on an instruction, the specified address must point to the first byte of the instruction, including any prefixes. If the address does not point to an instruction boundary, the break never occurs.

Data Breakpoints and I/O Access Breakpoints

Breakpoint types modify, write, and I/O (parameters `m`, `w`, and `i`) cause breaks when a location is accessed. These breaks occur immediately after the instruction causing the access. A modify breakpoint causes a break whenever memory at the specified address is written or read, but not when an instruction is fetched from memory. A write breakpoint causes a break only when there is a write at the address. An I/O breakpoint causes a break when an I/O port is read or written.

The modify, write, and I/O breakpoints require that you specify a length parameter following the `m`, `w`, or `i` on the command line. The choices of length parameter are byte, halfword, and word. If you try to set these breakpoints without a length parameter, SDM displays a syntax error message.

The length parameter determines the length of the breakpoint field, which begins at the specified address. A break occurs when a memory access in your program overlaps any part of the breakpoint field. For example, assume that you set a word breakpoint of type `m` (modify) at address B000H, which defines a four-byte breakpoint field beginning at B000H. A byte read or write of any address B000H through B003H causes a break. Likewise, a word read or write at address B002H causes a break, even though only two of the four bytes overlap the field you defined.

Note
The address on the command line should be a multiple of the specified length parameter (byte, halfword, or word). If you specify a byte, use any address. If you specify a halfword, use an address that is a multiple of two. If you specify a word, use an address that is a multiple of four. If the address is not a multiple of the length parameter, the program may break in unexpected places.

Breakpoint Display

When you issue the **bs** command without parameters, SDM displays the breakpoints currently set. For each breakpoint the output is:

`reason code`	The reason code assigned to the breakpoint
`s` or `h`	Indicates a software or hardware breakpoint
`address`	The breakpoint address, returned as a linear address
`e`, `m`, `w`, or `i`	Execution, modify, write, or I/O (only for hardware breakpoints)
`b`, `h`, or `w`	Byte, halfword, word (only for modify, write, or I/O breakpoints)

Displayed software breakpoints look like this:

```
00001000 s 00020af0
```

Displayed hardware breakpoints have two forms, either as below:

```
00000200 h 0000fffe e
```

or with a `b`, `h`, or `w` parameter:

```
00000201 h 000d1240 m h
```

Examples

Assume that you want your program to break when it reads or writes any memory location from 1FF3h through 1FF7h. Set two hardware breakpoints:

```
..bs 1ff3 l m b
00000200
..bs 1ff4 l m w
00000201
..
```

The first breakpoint is a modify breakpoint covering one byte in memory. SDM assigns it reason code 200h. The second breakpoint is a modify breakpoint covering four bytes in memory. SDM assigns it reason code 201h. To display the breakpoints, issue the **bs** command with no parameters:

```
..bs
00000200 h 00001ff3 m b
00000201 h 00001ff4 m w
..
```

c

Compares the block of memory that begins at the source address with the block of memory that begins at the destination address.

Syntax

```
[size] c source-address,destination-address
```

Parameters

`size` The decimal number of sequential bytes you want SDM to compare.

`source-address`
The beginning address of the source block of memory.

`destination-address`
The beginning address of the destination block of memory.

Additional Information

The size of the memory blocks must be at least as large as the number of bytes entered with the `size` parameter. SDM displays any mismatched bytes in this format:

```
aaaa:bbbb xx yy cccc:dddd
```

In this format, `aaaa:bbbb` and `cccc:dddd` are the addresses of the bytes that do not match and `xx` and `yy` are the bytes themselves.

Example

To compare two blocks of memory, each of which is 16 bytes long, enter:

```
..16c cs:118, cs:1a4-5
```

SDM responds with these three mismatches:

```
0200:00000118 6E 28 0200:0000019F
0200:0000011A 67 4E 0200:000001A1
0200:00000123 3C 2D 0200:000001AA
```

d

Displays either the contents of a specified block of memory (first syntax) or the contents of a descriptor table (second syntax).

Syntax

```
[count] d [data-type] [source-address] [,]
```

```
[count] d table-type [(expression)[.component]]
```

Parameters

`count` The number of command repetitions, in decimal. Each successive repetition is performed on successive items. For descriptor tables, `count` specifies how many table entries to display, beginning at a specific entry. In this case, `expression` is a required parameter. For example, `5dgdt(40t)` displays 5 entries from the GDT beginning at entry number 40.

If you don't specify `count`, the default is to display a screenful of entries until you type Q to quit. You can increment the display one screen at a time by typing <Space>, or one line at a time by typing <CR>.

`data-type`

The format type for displaying the block of memory. The data types and display formats are:

Enter This Data Type	To Specify This Display Format	
H	Halfword	(16 bits, hex)
W	Word	(32 bits, hex)
I	Word integer	(16 bits, hex & decimal)
LI	Long integer	(64 bits, hex & decimal)
LR	Long real	(64 bits, hex & decimal)
SI	Short integer	(32 bits, hex & decimal)
SR	Short real	(32 bits, hex & decimal)
T	Binary coded decimal	(10 bytes)
TR	Temporary real	(10 bytes)
X	Disassembled instruction	

If you omit this parameter, the block of memory is displayed in both byte and ASCII characters. Refer to previous sections in this chapter for more detailed information about data types.

`source-address`

The address in memory to display.

`, (comma)`

A comma at the end of the command line displays the block of memory you specified and a dash (-) prompt. Entering another comma displays the next block of memory that is equal in size to the one you specified in the original command. SDM then issues another dash and waits for you to enter another comma, or to terminate the command by entering a <CR>.

Note

You cannot use continuation commas when you are displaying descriptor tables.

`table-type`

The type of descriptor table you want SDM to display. The table types and the descriptor tables they reference are as follows:

Enter This Data Type	To Specify This Display Format
DT	Generic descriptor table. If you use this table type, include an "(expression)" parameter.
GDT	Global descriptor table
IDT	Interrupt descriptor table
LDT	Local descriptor table

If you specify only the table type, SDM displays the entire table.

See also: Descriptor tables, *Microprocessors Handbook* and *Introducing the iRMX Operating System*

`(expression)`

An expression enclosed in parentheses that references the descriptor table entry to display.

If `DT` is your `table-type`, use a segment register mnemonic (for example, `CS`) or selector enclosed in parentheses to designate the entry in the descriptor table to display. SDM uses the selector to decide which descriptor table you are referencing. Therefore, when you are debugging an application and you know the selector, you can examine the entry in the corresponding descriptor table without knowing whether it is a local or global descriptor table.

If `GDT`, `LDT`, or `IDT` is your table type, use sequential entry numbers with a `T` suffix to specify the table entry.

`.component`

Component name of the descriptor table entry to display. Include a period (.) before the component name so SDM can recognize the name. Table 3-8 lists the descriptor components and the descriptor types that apply to each component. Table 3-9 lists and describes the descriptor types. SDM uses these abbreviations when it displays the descriptor types.

Table 3-8. Descriptor Components and Types

Component	Definition	Descriptor Type
.base	segment base	dseg16, dseg32, eseg16, eseg32, tss286, dtable
.limit	segment limit	dseg16, dseg32, eseg16, eseg32, tss286, dtable
.wcnt	word count	callg286, callg386
.ssel	segment selector	callg286, callg386, trapg286, trapg386, intg286, intg386, taskg, tss286
.soff	segment offset	callg286, callg386, trapg286, trapg386, intg286, intg386
.dpl	descriptor privilege level	dseg 16, dseg32, eseg16, eseg32, callg286, callg386, trapg286, trapg386, intg286, intg386, taskg, tss286, tss386, dtable
.ed	expand down	dseg16, dseg32
.w	writable	dseg16, dseg32
.a	accessed	dseg16, dseg32, eseg16, eseg32
.c	conforming	eseg16, eseg32
.r	readable	eseg16, eseg32
.p	present	eseg16, eseg32
.b	big	dseg32
.g	granularity	dseg32, eseg32

Table 3-9. Descriptor Types

Descriptor Type	32-bit Descriptor	16-bit Descriptor
Data Segment	dseg32	dseg16
Executable Segment	eseg32	eseg16
Call Gate	callg386	callg286
Trap Gate	trapg386	trap286
Interrupt Gate	intg386	intg286
Task Gate	taskg	
Task State Segment	tss386	tss286
Descriptor Table	dtable	

See also: Descriptor types, *Programmer's Reference Manual* for your microprocessor

Examples

1. For example, to display fourteen disassembled instructions at the CS:EIP, enter:

```
..14dx
```

The display is similar to:

```
0F00:0000012A 55          PUSH   BP
0F00:0000012B 8BEC        MOV    BP, SP
0F00:0000012D 8BF5        MOV    SI, BP
0F00:0000012F 83C608      ADD    SI, 8
0F00:00000132 CDB8        INT    0B8H
0F00:00000134 85C9        TEST   CX,CX
0F00:00000136 7403        JZ     A = 013BH    [destination offset of
0F00:00000138 E82700      CALL   A = 0162H    [near jump and call
0F00:0000013B C45E04      LES    BX,[BP+04H]
0F00:0000013E 26890F      MOVES:[BX],CX
0F00:00000141 5D          POP    BP
0F00:00000142 C20600      RET    6
0F00:00000145 C8          ??                  [invalid opcode
```

The comments preceded by brackets ([) are not part of the display; they are included to help you understand the display.

2. To display five word integers at DS:0, enter:

```
..5di
```

The display is similar to:

```
1000:00000000 1ABDH  6845   0929H  2345
1000:00000004 FFFFH  -1     FBB6H  -1098
1000:00000008 115CH  4444
```

3. To display the fourth entry in the global descriptor table, enter:

```
..dgdt(4t)
```

The display is similar to:

```
GDT (4T) ESEG32  BASE=00FFC240 LIMIT=36BB P=1 DPL=0 A=1 C=0 R=1 G=0
```

4. To display the descriptor table entry which corresponds to the selector in the DS register, enter:

```
..ddt(ds)
```

The display is similar to:

```
LDT (5T) DSEG32  BASE=00000140 LIMIT=03FF P=1 DPL=0 ED=0 W=1 A=1 G=0
```

f

Searches a specified block of memory to find a selected sequence of hexadecimal numbers.

Syntax

```
[count] f source-address, data
```

Parameters

`count` The number of successive bytes in decimal for SDM to search.

`source-address`
The beginning address of the block of memory that you want SDM to search to find the hexadecimal number you specify in the `data` parameter.

`data` The sequence of hexadecimal digits you want SDM to find. SDM can use from one to 32 digits. SDM scans memory in byte units, with one byte being the smallest unit for which it can search. Therefore, specify either an even number of digits or only a single digit. For single digits, SDM adds a 0 before the digit. For example, you can specify the digits 1, 12, and 54455354, but not 544 or 54455. The first digit in the sequence is the least significant byte.

Additional Information

Each time SDM finds a match, it displays the address of the first matching byte.

Example

To have SDM find the number 54455354 in a memory block of 2000 bytes, starting from address 0, enter:

```
..2000f cs:0, 54455354
```

SDM displays the addresses where it found the specified number:

```
0200:00000118
0200:000001A4
0200:00000212
```

g

Executes your application program.

Syntax

```
g [start-address][, break-address][,break-address] [,break-
      address]
```

Parameters

`start-address`
: The address at which you want the application to begin executing. If you omit this parameter, the application begins executing at the address specified by the code segment (CS) and instruction pointer (EIP) registers.

`break-address`
: An address at which a breakpoint is set. Specify up to three break addresses within an application.

Additional Information

The **g** (go) command single-steps the first instruction and then executes all succeeding instructions at normal speed.

Breakpoints set with the **g** command are hardware execution breakpoints. This allows you to set breakpoints in ROM code.

When SDM hits a breakpoint, it displays the breakpoint information in this format:

```
break at xxxx:yyyyyyyy
```

This indicates the application stopped at the address `xxxx:yyyyyyyy`. After SDM displays this information, it issues a prompt.

Once a breakpoint is hit, it and any other breakpoints set in that **g** command are cleared.

A special situation arises when you specify a breakpoint address but not a starting address. If the breakpoint is in an interrupt handler and the current CS:EIP points to a software interrupt instruction (INT x, INTO, BOUND), SDM single-steps the interrupt instruction and executes the interrupt handler code at full speed. This bypasses the breakpoint. A workaround is to make sure the CS:EIP is pointing to an instruction preceding the software interrupt instruction before executing the **g** command.

See also: **bc** and **bs** commands for related breakpoint information

Example

This example tells SDM to go from the current CS:EIP and stop executing at CS:000007FA or 1F0:00000E20, whichever comes first.

```
..g, 7FA, 1F0:E20
```

i

Obtains and displays a byte, 16-bit halfword, or 32-bit word from the port you specify. The **ii** command accesses interconnect space on Multibus II systems.

Syntax

```
[count] i[h|w] port-address,expression
```

```
[count] ii [slot:]register
```

Parameters

`count` The number of bytes, 16-bit halfwords or 32-bit words in decimal that you want to obtain from the port and display. If you omit this parameter, SDM assumes one byte or word.

`h` Displays data from the port in halfword form. The data format defaults to bytes.

`w` Displays the port input in word form. Enter `w` immediately following the **i** command character, as `iw`.

`i` Specifies that you want SDM to access interconnect space. Enter `i` immediately following the command character **i**, as `ii`.

`port-address`
The address of the port from which you want to obtain data. Valid port addresses range from 0000 to 0FFFFH. Do not use a base portion of the address in this parameter.

`slot:` The decimal number of the slot containing the board whose interconnect register you want to access. If you do not include this parameter, the slot defaults to the slot containing the CPU board.

`register`
The interconnect register you want to access.

Examples

1. To display five bytes from the port that has the address 2FA, enter:

   ```
   ..5i2fa
   ```

   SDM responds with the five bytes you requested. The display on the specified port is similar to:

   ```
   FF
   FF
   FF
   FF
   FF
   ```

2. To display the first five interconnect registers of the board on which SDM is running, enter:

   ```
   ..5ii 0
   ```

   This example shows the contents of Register 0, the vendor ID of the CPU board.

m

Copies the contents of a block of memory to a memory address you specify.

Syntax

```
[count] m source-address,destination-address
```

Parameters

`count` The number of bytes in decimal that you want to copy from the source address in memory. If you omit this parameter, SDM assumes one byte.

`source-address`
The beginning address of the block of memory from which you want SDM to copy memory.

`destination-address`
The beginning address of the memory block to which you want SDM to copy.

Examples

This command tells SDM to copy 15 bytes of memory from a starting address of CS:2CD to a starting address of 200:4A.

```
..15 m cs:2cd, 200:4a
```

n

Displays single instructions in a disassembled form and then executes those instructions.

Syntax

```
[count] n[p][r] [start-address][,]
```

Parameters

`count` The number of instructions you want SDM to single step, in decimal.

`p` Tells SDM to treat any CALL routines as a single instruction. SDM displays the CALL instruction, executes the routine, and displays the next instruction. Specify the `p` immediately after the **n** command character, as `np`.

`r` Tells SDM to continue single step execution of instructions until a call instruction is encountered. If the `r` option is used, the `count` parameter is ignored. Specify the `r` immediately after the previous character, as `nrp` or `npr`.

`start-address`
The address at which you want SDM to begin executing single instructions. If you do not specify an address, SDM begins executing disassembled single instructions at the current CS:EIP.

`comma (,)`
Adding a comma (,) at the end of this command causes SDM to display the instruction you specified and a dash (-), at the end of the line. If you enter another comma, SDM executes the instruction and then displays the next disassembled instruction with another dash (-). SDM then waits for you to enter another comma or to terminate the command with a <CR>.

Examples

1. If you enter this command, SDM displays the instructions in disassembled form starting at the current CS:EIP and continuing until a CALL instruction. It then displays the CALL instruction and the dash prompt while waiting for further input.

   ```
   ..nr,
   ```

   If you enter a continuation comma now, SDM continues to execute the displayed instruction and display the next instruction in disassembled form until another CALL instruction is encountered. If you enter additional commas, SDM repeats this process.

2. If you enter this command, SDM displays and executes the first 23 instructions beginning at CS:4. It then displays the 24th instruction and waits for additional input from you.

   ```
   ..24n 4,
   ```

3. If your enter this command, SDM steps to the next call instruction, displays the stack, and, for system calls, the parameters. The “p” insures that that SDM does not step through a call instruction if it started from one.

   ```
   ..nrp; vs
   ```

o

The first command form enters data at the console and sends it to a port. The second command form (`oi`) accesses the interconnect space on a Multibus II system.

Syntax

```
[count] o[h|w] port-address,expression

[count] oi [slot:]register,expression[, expression] ...
```

Parameters

`count` The number of bytes, 16-bit halfwords, or 32-bit words in decimal you want SDM to send to the port. The default count is one.

`h` Sends data to the port in halfword form. The data format defaults to bytes.

`w` Sends data to the port in word form. The data format defaults to bytes.

`port-address`
The address of the port to which you want to send data. Valid port addresses range from 0000 to 0FFFFH. Do not use a base portion of the address in this parameter.

`expression`
The value to send to the port.

`i` Specifies that you want SDM to access interconnect space.

`register`
The interconnect register you want to access.

`slot:` The slot number in decimal containing the board whose interconnect register you want to access. The default slot is the slot for the board executing SDM.

Examples

1. This command outputs the value of AX + 1B9 as a word to port number 4.

   ```
   ..ow 4, ax + 1b9
   ```

2. This command sends the value 1 as a byte value to port number 2CDE one hundred times.

   ```
   ..100 o 2cde, 1
   ```

3. To write the value 40H to the interconnect register 0:25, the General Control register of the CSM board in slot 0, enter:

   ```
   ..oi 0:25,40
   ```

   The value 40H enables an agent that is not on the CPU board to interrupt the CPU.

4. To write multiple values to respective consecutive interconnect registers, enter:

   ```
   ..oi 3:10,1,2,3
   ```

   This writes to the interconnect registers for the board in slot 3, beginning at register 10. It writes 01H to register 10, 02H to register 11, and 03H to register 12.

pdbr

Displays the physical base address of the page directory, which is the highest-level page table. This is the same as the contents of register CR3.

Syntax

```
pdbr
```

pdd

Displays the page directory, which is a table containing entries about the secondary page tables. See the description of the `.component` parameter for the meaning of each entry in the display.

See also: Paging information in the user's manual or programmer's reference for your microprocessor

Syntax

```
[count] pdd [(index) [.component]]
```

Parameters

`count` A decimal number specifying how many entries from the page directory to display. If you don't specify `count` or `(index)`, the default is to display a screenful of entries until you type Q to quit. You can increment the display one screen at a time by typing <Space>, or one line at a time by typing <CR>. If you don't specify `count` but do specify `(index)`, the default value for `count` is 1.

`(index)`

The index into the directory to begin the display. For example, to display 5 entries beginning with entry 11 (index 10, since the index of page tables begins with 0), enter:

```
5pdd(10t)
```

`.component`

To display only a single component of a specific directory entry, after the `(index)` enter a period (.) followed by one of these component abbreviations:

ADDR	Base address of the second-level page table
AV	Available bits for use by the OS
S	(Pentium-specific) Size bit: 0 means 4 KB page
A	Accessed bit: 1 means read or write to page table
PCD	Page-level Cache Disable bit: 1 means caching is disabled
PWT	Page-level Writethrough bit: 1 is writethrough, 0 is writeback
U	User/Supervisor bit: 1 is application code and data, 0 is OS memory
W	Read/Write bit: 1 is read/write, 0 is read-only
P	Present bit: 1 means the page table is present in memory

Examples

1. The following command displays entries about the first six page tables. See the description of the `component` parameter above for the meaning of each field. The fifth entry (index 4) is not present in memory.

```
..6pdd(0)
PDIR(0T) ADDR=00273000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
PDIR(1T) ADDR=00274000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
PDIR(2T) ADDR=00275000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
PDIR(3T) ADDR=00276000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
PDIR(4T) ffc01ffe         Not Present
PDIR(5T) ADDR=006c1000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
```

2. The following command displays just the base address of the fourth page table.

```
..pdd(3).addr
ADDR=00276000
```

pdp

Given a pointer (segmented address), this command displays information about the physical address and indicates which page table and page in memory hold the address.

Syntax

```
pdp pointer
```

Parameters

`pointer`

A full pointer to memory, including a selector:offset.

Additional Information

A flat-model application's code and data reside in a virtual segment managed by the paging subsystem. The SDM **ddt** command does not display the actual physical address for memory in a virtual segment. Use the **pdp** command to find the memory location for addresses within virtual segments.

Examples

1. Assume that you issued an **npr** command to step to the next `call` instruction, which is displayed as:

```
c71b:0041001c e8bf0b0000       call   $+00000bc4 ;a=00410be0
```

The call is to address `00410be0` in segment `c71b`. To get the physical memory location of the address being called, issue the following command. It shows that this pointer is to physical address `006c6be0` and resides in the 6th page table (index 5T), at the 17th memory page in that table (index 16T).

```
..pdp c71b:00410be0
LINEAR ADDRESS=01410be0  PHYSICAL ADDRESS=006c6be0
PDIR(5T) ADDR=006c1000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(16T) ADDR=006c6000 AV=0 D=0 A=1 PCD=0 PWT=0 U=1 W=0 P=1
```

See the `component` parameter descriptions in the **pdd** and **pdt** commands for the meaning of the fields displayed in the PDIR and PTBL lines above.

pdt

Displays one or more entries from a page table, which points to memory pages holding a flat-model application's code and data. See the description of the `.component` parameter for the meaning of each entry in the display.

See also: Paging information in the user's manual or programmer's reference for your microprocessor

Syntax

```
[count] pdt [(index) [.component]] physical_base
```

Parameters

`count` A decimal number specifying how many entries from the page table to display. If you don't specify `count` or `(index)`, the default is to display a screenful of entries until you type Q to quit. You can increment the display one screen at a time by typing <Space>, or one line at a time by typing <CR>. If you don't specify `count` but do specify `(index)`, the default value for `count` is 1.

`(index)`
The index into the table to begin the display. For example, assume that the page table's base address is 25A000H. To display 5 entries beginning with entry 10, you would enter:

```
5pdt(10t) 25a000
```

`.component`
To display only a single component of a specific table entry, after the `(index)` enter a period (.) followed by one of these component abbreviations:

ADDR	Base address of the page
AV	Available bits for use by the OS
D	Dirty bit: 1 means memory in the page has been written
A	Accessed bit: 1 means read or write to the page
PCD	Page-level Cache Disable bit: 1 means caching is disabled
PWT	Page-level Writethrough bit: 1 is writethrough, 0 is writeback
U	User/Supervisor bit: 1 is application code and data, 0 is OS memory
W	Read/Write bit: 1 is read/write, 0 is read-only
P	Present bit: 1 means the page is present in memory

`physical_base`
The base address of the page table, as displayed by the **pdd** command.

Examples

1. Assume that a **pdd** command displayed the base address of a page table as 00276000. To get information about the memory pages in that page table, use the following command.

```
..pdt 276000
PTBL(0T) ADDR=00c00000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(1T) ADDR=00c01000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(2T) ADDR=00c02000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(3T) ADDR=00c03000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(4T) ADDR=00c04000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
PTBL(5T) ADDR=00c05000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
   .
   .
   .
PTBL(21T) ADDR=00c15000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
                    Enter <Space> or <CR> or Quit -
```

2. To get just the base address of the 6th page (index 5) in that page table, use the following command.

```
..pdt(5t).addr 276000
ADDR=00c05000
```

psd

Displays and lets you modify a field from a page directory entry. Use the **pdd** command to display one or more complete entries.

Syntax

```
psd (index).component
```

Parameters

`(index)`
The directory entry to modify. The first entry is index 0.

`.component`
The field in the entry to modify, from the following list:

ADDR	Base address of the second-level page table
AV	Available bits for use by the OS
S	(Pentium-specific) Size bit: 0 means 4 KB page
A	Accessed bit: 1 means read or write to page table
PCD	Page-level Cache Disable bit: 1 means caching is disabled
PWT	Page-level Writethrough bit: 1 is writethrough, 0 is writeback
U	User/Supervisor bit: 1 is application code and data, 0 is OS memory
W	Read/Write bit: 1 is read/write, 0 is read-only
P	Present bit: 1 means the page table is present in memory

Examples

1. Use the following commands to display the sixth entry (index 5) of the page directory and change the Accessed bit of that entry.

```
..pdd(5t)
PDIR(5T) ADDR=006c1000 AV=0 S=0 A=1 PCD=0 PWT=0 U=1 W=1 P=1
..psd(5t).a
1 - 0
..pdd(5t)
PDIR(5T) ADDR=006c1000 AV=0 S=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
```

pst

Displays and lets you modify a field from a page table entry. Use the **pdt** command to display one or more complete entries.

Syntax

```
pst (index).component physical_base
```

Parameters

`(index)`

The directory entry to modify. The first entry is index 0.

`.component`

To display only a single component of a specific table entry, after the `(index)` enter a period (.) followed by one of these component abbreviations:

ADDR	Base address of the page
AV	Available bits for use by the OS
D	Dirty bit: 1 means memory in the page has been written
A	Accessed bit: 1 means read or write to the page
PCD	Page-level Cache Disable bit: 1 means caching is disabled
PWT	Page-level Writethrough bit: 1 is writethrough, 0 is writeback
U	User/Supervisor bit: 1 is application code and data, 0 is OS memory
W	Read/Write bit: 1 is read/write, 0 read-only
P	Present bit: 1 means the page is present in memory

`physical_base`

The base address of the page table, as displayed by the **pdd** command.

Examples

1. Use the following commands to display the second entry (index 1) of the page table at address 00276000 and change the Read/Write bit of that entry.

```
..pdt(1) 276000
PTBL(1T) ADDR=00c01000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=1 P=1
..pst(1).w 276000
1 - 0
..pdt(1) 276000
PTBL(1T) ADDR=00c01000 AV=0 D=0 A=0 PCD=0 PWT=0 U=1 W=0 P=1
```

s

The first command form displays or modifies memory locations. The second and third command forms display or modify the components of descriptor table entries.

Syntax

```
[count] s [data-type][address][=expression][/expression] ...

s table-type (expression)=descriptor-type

s table-type (expression).component[=expression]
```

Parameters

`count` The number of times in decimal you want SDM to repeat the command.

`data-type`
The format in which you want SDM to display the memory locations you specify. The data types and the display formats they specify are:

Enter This Data Type	To Display This Format
H	Halfword
I	Word integer
L	Long integer
LR	Long real
S	Short integer
SR	Short real
T	Binary coded decimal
TR	Temporary real
W	Word

If you omit the `data-type` parameter, SDM displays the memory locations in both byte and ASCII characters.

See also: NPX data types, Table 3-4

`address`
The address of the memory location you want to display. If you omit this parameter, SDM displays the memory locations beginning at the value corresponding to the segment selector value in the DS register, with an offset of 0.

If you do not include any further parameters, SDM displays the contents of the address followed by a dash (-). At this point, if you wish to change the contents of the memory location, enter the new value followed by a <CR>. If you do not want to change the contents, enter a <CR> or a continuation comma. If you enter a comma, SDM displays the contents of the next location and prompts for more input.

`=expression`
The value with which you want to replace the value referenced by the `address` parameter. The equal sign (`=`) preceding the `expression` parameter instructs SDM to place the value indicated by the expression in memory.

`/expression`
The slash (`/`) preceding the second `expression` parameter separates any subsequent expressions and indicates the values you want to substitute in memory. SDM places these values in the locations immediately following the one you specified in the `address` parameter.

You can use the **s** command to display and optionally modify descriptor table entries. If you want to display descriptor table entries, use the second or third command form with the parameters described in this section.

`table-type`
The descriptor table you want SDM to display. The table types and the descriptor tables they reference are:

Enter This Table Type	To Display This Descriptor Table
DT	Generic descriptor table.
GDT	Global descriptor table
IDT	Interrupt descriptor table
LDT	Local descriptor table

See also: Descriptor tables, *Programmer's Reference Manual* for your microprocessor

`(expression)`
References the descriptor table entry you want to display.

If you chose `DT`, use a segment register mnemonic or a selector enclosed in parentheses to designate the entry in the local or global descriptor table you wish to display. SDM uses the selector to decide which descriptor table you are referencing. Therefore, when you are debugging an application and you know the selector, you can examine and modify the entry in the corresponding descriptor table without knowing whether it is a local or global table.

If `GDT`, `LDT`, or `IDT` is your table type, use sequential entry numbers with a `T` suffix to specify the table entry desired.

`=descriptor-type`

The abbreviation for the type of descriptor you want to use in place of the entry in the descriptor table referenced by the (`expression`) parameter. Place an equal sign (`=`) before the `descriptor-type` parameter. This causes SDM to initialize the entry you specified as the descriptor type. SDM then initializes all other fields of the descriptor as 0. Table 3-8 lists the descriptor types and the abbreviations you enter for them.

`.component`

The name of the of the descriptor table entry component you want to display. Include a period (.) before the component name so SDM can recognize the name. Table 3-8 lists the components associated with each type of descriptor, and Table 3-9 lists the descriptor types.

If you do not include any further parameters, SDM displays the contents of the descriptor table entry indicated by the expression followed by a dash (-). At this point, if you wish to change the contents of the descriptor table entry, enter the new value followed by a <CR>. If you do not want to change the descriptor table entry value, enter a <CR>.

See also: Descriptor types, *Programmer's Reference Manual* for your microprocessor

`=expression`

An expression that corresponds to a value to use in place of the value referenced by the `.component` parameter. The equal sign (`=`) preceding the `expression` parameter instructs SDM to place in memory the value indicated by the expression.

Additional Information

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 enter the **s** command without an equal sign (`=`), SDM displays a dash (-) prompt. Then, it waits for you to enter either:

- A continuation comma instructing SDM 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 SDM to substitute these values in place of those already in the memory location you specified.

SDM continues to issue dash prompts until you enter a <CR>.

Examples

1. This example illustrates how to use the **s** command to display and then modify contents of a memory location. Suppose you enter the **s** command by itself:

```
..s
```

SDM responds with the contents of the byte at DS:0:

```
0170:00000000  40 -
```

At this point, SDM issues a dash prompt and waits for you to enter an expression or a comma. Suppose you enter:

```
0170:00000000  40 - 1/2/3, Five bytes are modified in
                           this example.
0170:00000003  20 - 4/15
```

SDM replaces the existing data in the memory locations with the expressions you entered. It then displays the next memory location because you entered a continuation comma. If you enter a <CR> after this substitution, you return to the prompt and can enter any SDM command.

If you want to replace the data in a given memory location without checking to see what it contains, you can enter:

```
..100sw ds:ax=ffff
```

2. This example shows how to display and modify the conforming bit component of a descriptor table entry. The CS register indicates where the descriptor table is located.

```
..sdt(cs).c
```

SDM responds with this line and waits for you to substitute an expression:

```
0 -
```

This line illustrates a similar example:

```
..sgdt(4t).base
```

SDM responds with the contents of the base component of the fourth global descriptor table entry and waits for you to enter an expression:

```
123456 -
```

3. This example sets the 45th global descriptor table entry to a data segment descriptor with all fields set to 0:

```
..sgdt(45t)=dseg32
```

4. You can use register names to specify address selectors and offsets. For example, to examine the word at an address with a selector value contained in the AX register, enter:

```
..swax:0
```

x

Allows you to examine and optionally modify the contents of NPX and CPU registers such as the task state segment.

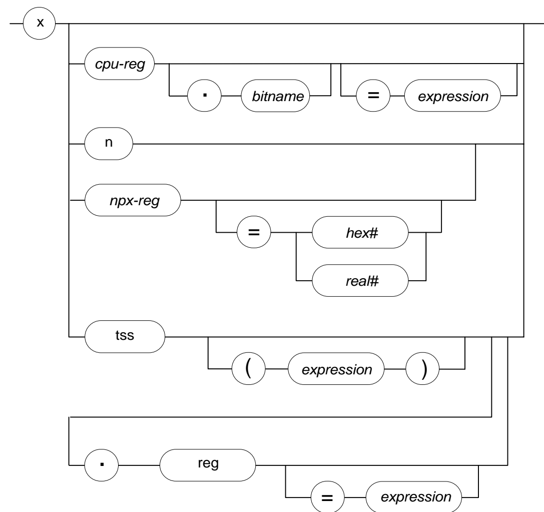
Syntax

```
x [n]

x [[cpu-reg][.bitname][=expression]]

x [[npx-reg][=hex#|real#]]

x tss [(expression)][.reg[=expression]]
```



Parameters

If you use the **x** command with no parameters, SDM displays the current contents of the CPU registers with respect to your application at the time the fault, trap, or break occurred.

`n` The `n` option instructs SDM to display the contents of the NPX registers and NPX stack registers.

`cpu-reg`

The abbreviation for the Intel386, Intel486, or Pentium microprocessor register that you want to display or modify.

If you do not include any other parameters, SDM displays the contents of the register or bit followed by a dash (-) prompt. At this point, to change the register or bit value, enter the new value followed by a <CR>. If you do not want to change the value, enter a <CR>.

The register abbreviations you can enter are listed below. Not all registers may be accessible on your CPU. For example, CR4 is not defined on the Intel386 microprocessor.

Register Type	32-bit Abbreviation	16-bit	8-bit	
General Registers	EAX	AX	AH	AL
	EBX	BX	BH	BL
	ECX	CX	CH	CL
	EDX	DX	DH	DL
	EBP	BP		
	ESI	SI		
	EDI	DI		
Stack Pointer	ESP	SP		
Code Segment	CS			
Data Segment	DS			
Stack Segment	SS			
Extra Data Segments	ES			
	FS			
	GS			
Flag Register	EFL	FL		
Instruction Pointer	EIP	IP		
Control Registers	CR0	MSW		
	CR1			
	CR2			
	CR3			
	CR4			

The FL, EFL, MSW, CR0, CR3, and CR4 registers are special registers containing bit fields. SDM displays the contents of these registers first with the binary values and a mnemonic for each bit field, then as a hexadecimal word value. For example, the display of EFL might be as follows, where 0 or 1 as a mnemonic indicates a reserved bit field:

```
ID VIP VIF AC VM RF  0 NT IOPL OF DF IF TF SF ZF  0 AF  0 PF  1 CF
 0  0   0   0  0  0  0  0  00   0  0  1  0  0  1  0  0  0  1  1  0
00000246 -
```

The mnemonics for bit fields in the special registers are listed below

EFL & FL	Bit Names	CR0 & MSW	Bit Names
AC	Alignment Check	AM	Alignment Mask
AF *	Auxiliary Carry Flag	CD	Cache Disable
CF *	Carry Flag	EM **	Emulation Mode (Coprocessor)
DF *	Direction Flag	ET **	Extension Type
IF *	Interrupt Enable Flag	MP **	Math Present (Monitor Coprocessor)
ID	Identification Flag	NE	Numerics Exception
IOPL *	I/O Privilege Level (2 bits)	NW	Not Write-Through
NT *	Nested Task Flag	PE **	Protection Enable
OF *	Overflow Flag	PG	Paging Enable
PF *	Parity Flag	TS **	Task Switch
RF	Resume Flag	WP	Write Protect
SF *	Sign Flag	**CR3**	**Bit Names**
TF *	Trap Enable Flag	PCD	Page Cache Disable
VIF	Virtual Interrupt Flag	PWT	Page Write-Through
VIP	Virtual Interrupt Pending	(no mnemonic)	Physical base address of page directory table in bits 12-31
VM	Virtual 8086 Mode	**CR4**	**Bit Names**
ZF *	Zero Flag	DE	Debugging Extensions
		MCE	Machine Check Enable
		PSE	Page Size Extensions
		PVI	Protected Mode Virtual Interrupt
		TSD	Time Stamp Disable
		VME	Virtual 8086 Mode Extensions

* Bit fields in the Flags register (FL), which is a subset of EFL

** Bit fields in the Machine Status Word (MSW), which is a subset of CR0

`.bitname`

To display and change a single bit field from one of the special registers, follow the `cpu-reg` name with a period (.) and the mnemonic for the bit field. For example, to display only the Carry Flag from the Extended Flags register, you would enter:

```
x efl.cr
```

`=expression`

An expression that corresponds to a value you want to place in the register or TSS you specified. The equal sign (=) preceding the `expression` parameter instructs SDM to place the value in the register.

`npx-reg`

The abbreviation for the math coprocessor (NPX) register you want to display and optionally modify. Table 3-10 lists the NPX registers and the abbreviations known to SDM.

Table 3-10. NPX Registers

Register Name	Abbreviation	Register Name	Abbreviation
NPX State	N	Status Word	SW
Control Word	CW	Tag Word	TW
Instruction Pointer	IP *	Data Pointer	DP *
Stack Register 0	ST(0)	Stack Register 4	ST(4)
Stack Register 1	ST(1)	Stack Register 5	ST(5)
Stack Register 2	ST(2)	Stack Register 6	ST(6)
Stack Register 3	ST(3)	Stack Register 7	ST(7)

* These registers cannot be modified.

If you do not include any further parameters, SDM displays the contents of the NPX register followed by a dash (-) prompt.

At this point, to change the register value, you can enter a value. Enter stack register values as real numbers; enter all other NPX registers values as words. You cannot modify the IP and DP registers for the NPX. If you do not want to change the NPX register value or display any further NPX register contents, enter a <CR>.

`=hex#`

The hexadecimal number you want to place in the NPX register you specified.

`=real#`

The real number you want to place in the NPX stack register you specified.

`tss` The task state segment (TSS) option instructs SDM to display the contents of the task state segment. The contents of the TSS are listed in Table 3-11.

To access TSS registers, use this parameter. If you do not include any further parameters, SDM displays the current contents of the TSS whose selector is in the TR register.

Table 3-11. Task State Segment

Name	Abbreviation
Local Descriptor Table Register	LDTR
Interrupt Descriptor Table Register	IDTR
Global Descriptor Table Register	GDTR
Task Register	TR
Link to Nested Task	LINK
Level 0 Stack Segment	SS0
Level 1 Stack Segment	SS1
Level 2 Stack Segment	SS2
Level 0 Stack Pointer	ESP0
Level 1 Stack Pointer	ESP1
Level 2 Stack Pointer	ESP2

`(expression)`
A selector for a TSS.

If you do not include any further parameters, SDM displays the contents of the indicated TSS.

`.reg` The name of the TSS component that you wish to display. Include a period (.) before the component name so SDM can recognize the name.

If you do not include any further parameters, SDM displays the contents of the register in the TSS followed by a dash (-) prompt.

At this point, to change the TSS component register value, enter the new value followed by a <CR>. If you do not want to change the TSS contents, enter a <CR>.

Additional Information

If you use the **x** command with both a register name and an expression, the modification you specify takes place immediately and SDM does not display the new value. If you redisplay the NPX state (after modification), SDM displays the results of the change you made.

You can use the **x** command to set the registers or the TSS contents to any value. If you use any invalid values, SDM reports them and does not execute the change.

Examples

1. This example shows the results of invoking the **x** command with no parameters on an Intel386-based board:

   ```
   ..x
   ```

   SDM responds with this display:

```
EAX=0000FFFF  CS=1000  EIP=00000000  EFL=00000000  LDTR=2A0
EBX=0000FFFF  SS=0000  ESP=00000428  BP=0000FFFF   TR=278
ECX=0000FFFF  DS=0000  ESI=0000FFFF  FS=0000       MSW=FFF0
EDX=0000FFFF  ES=0000  EDI=0000FFFF  GS=0000
GDTR .BASE=00000000          .LIMIT=0000
IDTR .BASE=00000000          .LIMIT=0000
```

2. This example illustrates the results of invoking the **x** command with only the **n** option as a parameter:

```
..xn
```

SDM responds:

```
CW: X  X  X IC  R  C  P  C IM  X PM UM OM ZM DM IM
    0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1   IP = 00000:0000
SW: B C3  T  O  P C2 C1 C0 IR  X PE UE OE ZE DE IE
    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TW: T  7  T  6  T  5  T  4  T  3  T  2  T  1  T  0
    0  0  0  0  1  1  1  0  1  0  0  0  0  0  0  1   DP = 00000:0000
ST(0) ZERO     00000000000000000000R   0
ST(1) VALID    3FFF999999999999999AR   1.2
ST(2) VALID    BFFF999999999999999AR  -1.2
ST(3) SPECIAL  FFFF0000000000000000R  -Infinity
ST(4) SPECIAL  7FFFFF00000000000000R  +NAN
ST(5) EMPTY    4000C90FDA9E46A7843ER   3.14159265
ST(6) VALID    4CF5F08B8D41AF800AC8R   1.23456E+999
ST(7) VALID    3FFF1800000000000000R   .1875 UNNORM 3 BITS
```

See also: CW, SW, and TW output fields, in the hardware reference manual for your microprocessor

3. This example instructs SDM to display the contents of the task state segment whose selector is 068H:

```
..x tss(68)
```

SDM responds:

```
EAX=00001234 CS=0020 EIP=00000000 EFL=00000000 LDTR=02D0 LINK=0058
EBX=00001234 SS=0000 ESP=00000428 EBP=0000FFFF SS0=0020  ESP0=0000FFFE
ECX=00001234 DS=0000 ESI=0000FFFF  FS=0000     SS1=0000  ESP1=00000000
EDX=00001234 ES=0000 EDI=0000FFFF  GS=0000     SS2=0000  ESP2=00000000
```

❑❑❑

Console I/O Calls A

The console I/O calls are calls for reading from and writing to SDM's console device. A console device is a standalone terminal attached to a target system or a development system. You can configure what I/O device SDM uses as a console in the ICU, on the SDM screen under the SUB-(systems) screen.

Table A-1 lists the console I/O calls.

Table A-1. Console I/O Calls

Call	Description
ci	Reads a character from the console input device. Waits for an available character.
co	Writes a character to the console output device.
csts	Reads a character from the console input device. Does not wait for an available character.

Using the Console I/O Calls

iRMX software contains interface libraries for the console I/O calls. For PL/M, the library file is *cico.lb3* and *cico.lb2*. This file contains the external declarations for the **ci**, **co**, and **csts** calls. Include the library file in applications that use console I/O calls.

The console I/O library supports only the LARGE segmentation model. To use this library, include this statement in your program and make a far call:

```
$large(cico exports ci, co, csts)
```

See also: Segments and subsystems, *Programming Techniques*, *iC-386 Compiler User's Guide*, and *PL/M-386 Programmer's Guide*

Note

When you use C, subsystem declarations must be put in a separate file and then included using the **subsys** directive.

ci

Reads an ASCII character from the console input device and places it in the AL register. It waits for character input.

Syntax

```
a$char = ci;

a_char = ci();
```

Parameter	PL/M Data Type	C Data Type
a_char	BYTE	UINT_8

Return Value

`a_char` A character entered at the console.

Additional Information

The parity (high) bit is stripped from any character entered at the console.

The **ci** call affects the AX and EAX registers and the CPU flags. After a **ci** call, the CPU flag contents are undefined.

A **ci** call does not disable interrupts.

Examples

These PL/M program segments show how to use the **ci** call to retrieve a character from the console:

```
CI:
      PROCEDURES BYTE EXTERNAL;
      END CI;
            ..._
            ..._
      DECLARE CHAR BYTE;
            ..._
            ..._
      DISABLE;
      CHAR = CI;
      ENABLE;
```

co

Transfers a character from the low-order byte of the word on the top of the stack to the console output device.

Syntax

```
call co(a$char);
```

```
co(a_char);
```

Parameter	PL/M Data Type	C Data Type
a_char	BYTE	UINT_8

Parameter

`a_char` An ASCII character to be output to the console.

Additional Information

If you have entered a <Ctrl-S> at the console, the call waits for you to enter <Ctrl-Q> before transmitting the character.

The **co** call affects the AX and EAX registers, as well as the CPU flags. After using the **co** routine, the flag contents are undefined.

The **co** call does not function properly if the parity (high) bit is set. The characters must be in the range of 00H to 7FH.

See also: Editing characters, in this manual

Examples

These PL/M program segments show how to use the **co** call to transmit a character to the console:

```
CO:
      PROCEDURE (CHARACTER) EXTERNAL;
            DECLARE CHARACTER BYTE;
      END CO;
            ..._
            ..._
      DECLARE CHAR BYTE:
            ..._
            ..._
      CALL CO(CHAR);
```

csts

Reads an ASCII character from the console input device and places it in the AL register. It does not wait for character input.

Syntax

```
a$char = csts;
```

```
a_char = csts();
```

Parameter	PL/M Data Type	C Data Type
a_char	BYTE	UINT_8

Return Value

`a_char` Receives a character from the console.

Description

The **csts** call does not wait for console input if a character is not available immediately. Instead, it sends an ASCII null character (0) to the console or the application, whichever makes the call. It then returns to the application and executes the next instruction.

The parity (high) bit is stripped from any character entered at the console.

The **csts** call affects the AX and EAX registers and the CPU flags. After using the **csts** call, the flag contents are undefined.

Examples

These PL/M program segments show how to use the **csts** call to retrieve a character from the console:

```
CSTS:
      PROCEDURE BYTE EXTERNAL;
      END CSTS;
            ..._
            ..._
      DECLARE CHAR BYTE;
            ..._
            ..._
      CHAR = CSTS;
```

Related Publications B

You may need to refer to one or more of the following manuals:

- *80387 Programmer's Reference Manual*
- *i386™ DX Hardware Reference Manual*
- *i386 Programmer's Reference Manual*
- *i386 System Software Writer's Guide*
- *i387™ DX Programmer's Reference Manual*
- *i486 Microprocessor Hardware Reference*
- *i486™ Programmer's Reference Manual*
- *Pentium Processor User's Manual*
- *iC-386 Compiler User's Guide*
- *Microprocessor Handbook*
- *PL/M-386 Programmer's Reference*

Index

A

address parameter, definition, 86
AL register, 138
ASCII characters
    reading, 138, 141
    transferring, 139
AX register, 139, 141

B

backspace key, 81
bc SDM command, 94
Bootstrap Loader debug option, 3
breakpoint clear SDM command, 94
breakpoint set SDM command, 96
breakpoints
    clearing, 94
    displaying, 100
    execution, 99
    redundant, 94
    setting, 4, 96
bs SDM command, 96
byte parameters, 84

C

c SDM command, 102
ci call, 138
clearing breakpoints, 94
CLI-restart, 7
co call, 139
combining SDM commands, 82
compare SDM command, 102
comparing memory blocks, 102
configuration, 2
console I/O calls
    ci, 138
    co, 139
    csts, 141
    summary, 137
contents of the stack, 39
continuation comma, 82
continuing SDM commands, 82
control keys, for editing, 81
conventions, 10
copying memory blocks, 112
CR3 register, 117
CS, definition, 108
CS:EIP
    definition, 3
CS:EIP, definition, 10
csts call, 141
Ctrl-C keys, 81
Ctrl-Q keys, 81
Ctrl-S keys, 81
Ctrl-X keys, 81

D

d SDM command, 103
data types, 88
debug command (HI), 3
decimal values, 91
descriptor types, 104, 127
displaying memory descriptor tables, 103
displaying/executing instructions, 113
DUIB information, displaying, 11

E

editing SDM commands, 81
EIP, definition, 108
entering SDM commands, 81
error messages
    SDM, 93
examining/modifying registers, 130
expression parameters, 86

F

f SDM command, 107
fail-safe timeout, 26
find SDM command, 107
flat model, 43, 96, 120, 121
    displaying pointers, 120
front panel interrupt button, 3

G

g SDM command, 108
GDT slots, displaying free amount, 20
getting help, 21
go SDM command, 108

H

halfword parameters, 84
hardware/software requirements, 2
help, 21

I

i SDM command, 110
I/O Result Segment (IORS), 35
input SDM command, 110
instructions, displaying/executing, 113
Int3 instruction, 3
invocation, 3
IORS, displaying, 35
iSDM see SDM, 1

J

job tokens, displaying, 22

L

long integer data type, 88
long read data type, 88

M

m SDM command, 112
manuals, related, 143
memory blocks
    comparing, 102
    copying, 112
    displaying, 103
memory descriptor tables, displaying, 103
Message Passing Coprocessor see MPC, 26
models of segmentation, 137
monitor
    definition, 1
move SDM command, 112
MPC, 27, 30
    input message queue, 27
MPC (Message Passing Coprocessor), 26
Multibus II, 26, 27, 30
multiple SDM commands on a single line, 82

N

n SDM command, 113
nonnumeric values, 91
NPX
    data type, 88
    integers, 88
    number format, 90
    real numbers, 89
NPX (numeric processor extension), 87
numeric parameters, 87
numeric processor extension, See NPX

O

o SDM command, 115
object directory, displaying, 18
objects, displaying, 33
offset, definition, 86
output SDM command, 115

P

packed binary coded decimal (BCD) numbers, 89
packed decimal data type, 88
page directory
    base address of, 117
    changing, 123, 124
    displaying, 118
page table
    displaying, 121

pdbr SDM command, 117
pdd SDM command, 118
pdp SDM command, 120
pdt SDM command, 121
port input SDM command, 110
port output SDM command, 115
product overview, 1
prompts, description, 10
psd SDM command, 123
pst SDM command, 124

Q

quitting the debugger, 7

R

reading ASCII characters, 138, 141
reason codes
  breakpoint, 98, 100
register, definition, 85
related publications, 143
repeating SDM commands, 82
requirements, hardware and software, 2
returning to your application, 7
rubout key, 81

S

s SDM command, 125
SDB commands, 4
  overview, 4
  summary, 9
  syntax, 4
  token validity in, 6
  vb, 11
  vc, 15
  vd, 18
  vf, 20
  vh, 21
  vj, 22
  vk, 25
  vmf, 26
  vmi, 27
  vmo, 30
  vo, 33
  vr, 35
  vs, 39
  vt, 45
  vu, 74
SDB prompt, description, 10
SDM commands
  bc, 94
  breakpoint clear, 94
  breakpoint set, 96
  bs, 96
  c, 102
  combining, 82
  compare, 102
  continuing, 82
  d, 103
  display memory descriptor table, 103
  editing, 81
  entering, 81
  f, 107
  find, 107
  g, 108
  go, 108
  i, 110
  input, 110
  line conventions, 81
  m, 112
  move, 112
  n, 113
  o, 115
  output, 115
  parameters, 84
  pdbr, 117
  pdd, 118
  pdp, 120
  pdt, 121
  psd, 123
  pst, 124
  repeating, 82
  s, 125
  structure, 80
  x, 130
SDM editing keys, 81
SDM error messages, 93
SDM parameters
  address, 86
  byte, 84
  expression, 86
  halfword, 84

numeric, 87
term, 85
types, 84
word, 84
segment selector, definition, 86
setting breakpoints, 4, 96
short integer data type, 88
short real data type, 88
single step, definition, 80
Soft-Scope, 2
prompt, description, 10
special-case numeric values, 92
SS:ESP
definition, 10
stack contents, 39
starting the debugger, 3
structure of SDM commands, 80
substituting memory/descriptor table entries, 125
susbsys directive, 137
syntax for debugger commands, 4
system call information, displaying, 15
system call parameters on the stack
displaying, 39
identifying, 39
interpreting, 39
system requirements, 2

T

task state segment, 134
task system calls, displaying, 74
task tokens, displaying, 25
temporary real data type, 88
term parameters, 85
tokens
displaying, 45
validity checking, 6
transferring ASCII characters, 139

V

vb SDB command, 11
vc SDB command, 15
vd SDB command, 18
vf SDB command, 20
vh SDB command, 21
vj SDB command, 22
vk SDB command, 25
vmf SDB command, 26
vmi SDB command, 27
vmo SDB command, 30
vo SDB command, 33
vr SDB command, 35
vs SDB command, 39
vt SDB command, 45
composite object display, 56
extension object display, 55
job display, 46
mailbox display, 51
region display, 54
segment display, 55
semaphore display, 53
task display, 48
vu SDB command, 74

W

warm-start, 7
word integer data type, 88
word parameters, 84

X

x SDM command, 130