intel

iRMX®
Disk Verification Utility
Reference Manual

Order Number: 462922-001

intel®

iRMX®
Disk Verification Utility
Reference Manual

Order Number: 462922-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

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

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

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

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

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9 (a) (9).

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

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

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

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

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

PREFACE

INTRODUCTION

The iRMX® Disk Verification Utility is a software tool for the iRMX I and iRMX II Operating Systems. The utility runs as a Human Interface command verifying and modifying the data structures of iRMX named and physical volumes.

This manual includes invocation instructions and detailed descriptions of all utility commands. It also documents the iRMX capability of backing up and restoring volume file descriptor nodes (fnodes).

In addition, the manual describes the structure of iRMX named volumes as users must be familiar with volume structure to use the full capabilities of the Disk Verification Utility.

READER LEVEL

This manual is intended for programmers who have an understanding of the operating system, and particularly the Basic I/O System and Human Interface layers. To use this manual effectively, programmers should be familiar with iRMX volume structure. Appendix A provides a description of iRMX named volume structure. However, it is intended as a reference and not as a tutorial.

MANUAL OVERVIEW

This manual is organized as follows:

Chapter 1	This chapter describes two ways of invoking the utility: single-command mode or interactive mode. It explains single-command mode and how to interpret output and error messages from the single-command verification. It also describes the invocation in interactive mode and the interactive mode error messages. Commands for the interactive mode are explained in Chapter 2.
Chapter 2	This chapter contains detailed descriptions of the Disk Verification Utility commands. The commands are discussed in alphabetical order. When verifying and modifying volumes, you should refer to this chapter for specific information about the format and parameters of the commands.
Chapter 3	This chapter explains the fnode backup and restore feature in detail. This feature provides a limited mechanism for attempting to recover data when the volume label or the fnode file has been damaged.

Appendix A	This appendix provides information on the format of iRMX named volumes. It includes details of the volume label and fnode file, differences between long and short files, and format information specific to diskettes. Programmers should be familiar with this information before attempting to modify a volume.

CONVENTIONS

This manual uses the following conventions:

- Information appearing as UPPERCASE characters when shown in keyboard examples must be entered or coded exactly as shown. You may, however, mix lower and uppercase characters when entering the text.
- Fields appearing as lowercase characters within angle brackets (< >) when shown in keyboard examples indicate variable information. You must enter an appropriate value or symbol for variable fields.
- User input appears in as **blue** text.
- **Blue** text is used to indicate the first occurrence of each command described in Chapter 2; subsequent occurrences are printed in black.
- All numbers unless otherwise stated are assumed to be decimal. Hexadecimal numbers include the "H" radix character (for example, 0FFH).

CONTENTS

Chapter 1. Invoking Diskverify

Chapter 2. Diskverify Commands

Chapter 2. Diskverify Commands (continued)

Chapter 3. Backing Up and Restoring Fnodes

Appendix A. Structure of a Named Volume

Appendix A. Structure of a Named Volume (continued)

Tables

Figures

1 INVOKING DISKVERIFY

1.1 INTRODUCTION

When using an iRMX® application system, you will need to store data on secondary storage devices. Unfortunately, occasional power irregularities or accidental reset may destroy the index to the data on these devices, making the information inaccessible to the system. In some cases, losing even a small amount of data can render an entire volume useless.

You need a tool to examine and fix the damaged volume. The tool should enable you to determine how much of the data was damaged and help you recreate file structures on the damaged volume. The iRMX Disk Verification Utility (DISKVERIFY) is a tool that enables you to verify the consistency and recover damaged data on iRMX volumes.

The Disk Verification Utility inspects, verifies, and corrects the data structures of iRMX named volumes. It can also verify an iRMX physical volume. The Disk Verification Utility can reconstruct the fnode file, the volume label, the file descriptor nodes (fnodes) map, the volume free space map, and the bad blocks map of the volume. In addition, with DISKVERIFY you can manipulate fnodes, bad track information, and the actual data on the volumes. The Disk Verification Utility also supports auto-volume recognition which means you can verify any iRMX named volume without detaching and attaching the device with the correct DUIB.

You can use DISKVERIFY in one of two ways:

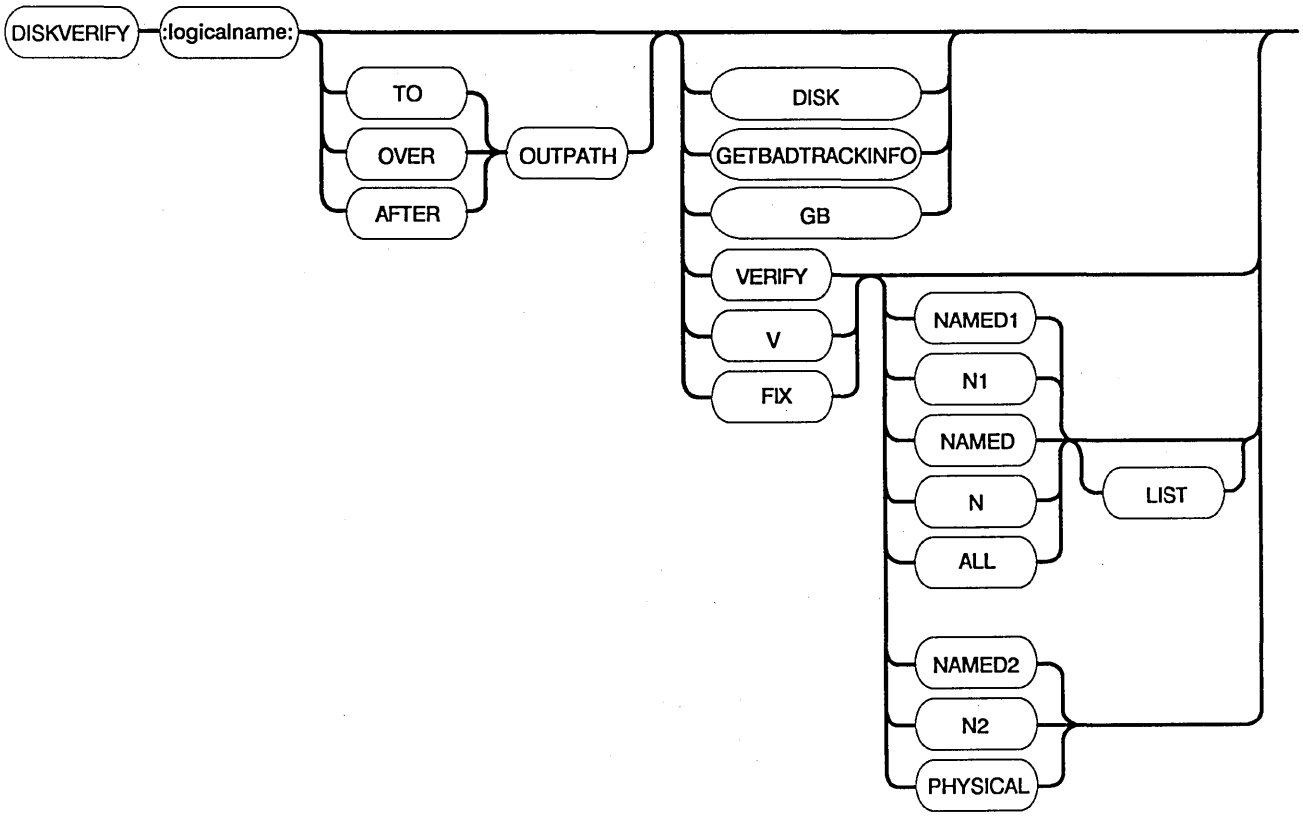
- As a single command that verifies the structures of a volume and returns control to the Human Interface
- As an interactive program that enables you to check and modify data on the volume by entering disk verification commands

To take full advantage of this utility, you must be familiar with the structure of iRMX named volumes (either iRMX I or iRMX II; the volume structure is almost the same for both operating systems). Appendix A contains detailed information about volume structure. If you are unfamiliar with the iRMX volume structure, you should avoid using the DISKVERIFY commands. Some commands, if not used correctly, can render your volumes unusable.

However, even if you know nothing about iRMX volume structures, you can still use the Disk Verification Utility as a single command to verify that the data structures on an iRMX volume are valid.

1.2 INVOCATION

To invoke DISKVERIFY, enter the following command:



where:

:logical name:	Logical name of the secondary storage device containing the volume to be verified.
TO	Copies the output from the Disk Verification Utility to the file specified in OUTPATH. If no "TO" is specified, output is directed to the console screen (:CO:).
OVER	Copies the output from the Disk Verification Utility over the specified file.

AFTER — Appends the output from the Disk Verification Utility to the end of the specified file. If the file does not exist, it is created.

OUTPATH — Pathname of the file to receive the output from the Disk Verification Utility. If you omit this parameter and no preposition is specified, output is directed to the console screen (:CO:) by default. You cannot direct the output to a file on the volume being verified. If you attempt this, the utility returns an E$ALREADY_ATTACHED error message.

Following is a list of the DISKVERIFY options. If you invoke DISKVERIFY without specifying one of these options, you enter the interactive mode. In this case, the utility displays a header message and the utility prompt (*). You can then enter any of the DISKVERIFY commands listed in Chapter 2.

DISK — Displays the attributes of the volume being verified. If you specify this option, the utility performs the function and returns control to you at the Human Interface level. You can then enter any Human Interface command, provided that the device verified is not the system device. Any parameter after this one is ignored. Refer to the description of the DISK command in Chapter 2 for more information.

NOTE

Although you can use DISKVERIFY to verify the system device (:SD:), note that all connections to this device are deleted by the operating system. After exiting, you must reboot the system or use the warm start feature (see the *iRMX® System Debugger Reference Manual*).

GETBADTRACK-INFO or GB — Reads the bad track information from the volume and displays it. Bad track information that is redirected to a file can be used as input to the FORMAT command by removing the header information. Chapter 2 provides a complete explanation of this command.

VERIFY or V — Verifies the volume. This function and the associated options are described in detail under "VERIFY" in Chapter 2. If you specify only this option, the utility performs the NAMED verification function and returns control to you at the Human Interface level. You can then enter any Human Interface command, provided the device verified is not the system device.

FIX

Performs the same functions as VERIFY. In addition, it tries to fix several types of problems on the volume after performing the verification. You should be careful when using FIX as it changes the data on the disk (which may prove dangerous). For example, during NAMED1 verification, FIX corrects the checksums on fnodes with bad checksums. However, an fnode with a bad checksum may indicate another problem with the fnode which should not be ignored. As a result, it is recommended that you use FIX only after performing the following steps.

1. Use DISKVERIFY with the VERIFY option.
2. Examine the output and the problems on the volume to determine the type of "fix" needed.
3. If the problems can be fixed by DISKVERIFY, run DISKVERIFY with the FIX option to correct the problems.

NAMED1 or N1

VERIFY or FIX option that applies to named volumes only. This option checks the fnodes of the volume to ensure that they match the directories in terms of file type and file hierarchy. This option also checks the information in each fnode to ensure consistency.

When used with FIX, the NAMED1 option connects bad orphan fnodes to their parents. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

NAMED2 or N2

VERIFY or FIX option that applies to named volumes only. This option checks the allocation of fnodes and space on the volume, constructs the space and fnode bit maps to reflect the current contents of the volume, and verifies that the fnodes point to the correct locations on the volume. When used with the FIX option, NAMED2 saves the correct bit maps, that were constructed during the verification phase, on the volume. It also removes fnodes with multiple references from their illegal parents. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

NAMED or N

VERIFY or FIX option that performs both the NAMED1 and NAMED2 verification functions on a named volume. If you specify VERIFY or FIX with no option, the system assumes NAMED (default).

ALL

VERIFY or FIX option that applies to both named and physical volumes. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs only the PHYSICAL verification function.

PHYSICAL	VERIFY or FIX option that applies to both named and physical volumes. This option reads all blocks on the volume and checks for I/O errors. When used with FIX, it adds the bad blocks that it encounters to the volume's bad block map.
LIST	A control that you can use with any option that activates NAMED1 verification (NAMED, NAMED1, or ALL). When you use this option, the file information generated by VERIFY or FIX is displayed for every file on the volume, even if the file contains no errors. Refer to the description of the VERIFY and FIX commands in Chapter 2 for more information.

1.3 OUTPUT

When you enter the DISKVERIFY command, the utility responds with

```
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
```

where <version> is "I" or "II", depending upon the version of the operating system, and Vx.x is the version number of the utility. If you specify the VERIFY (or V) parameter in the DISKVERIFY command, the utility verifies the volume and displays the verification information on the screen (or copies it to the file specified by the outpath parameter). The verification information is the same as that from the VERIFY utility command. After generating the verification output, the utility returns control to the Human Interface, which prompts you for more Human Interface commands. The following is an example of such a DISKVERIFY command:

```
-DISKVERIFY :F1: VERIFY NAMED2 <CR>
iRMX <version> Disk Verify Utility,  Vx.x
Copyright <year> Intel Corporation
All Rights Reserved

DEVICE NAME = wfd0  : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED2'  VERIFICATION
    BIT MAPS O.K.
-
```

If you omit the DISK or VERIFY parameter from the DISKVERIFY command, the utility does not return control to the Human Interface. Instead, it issues an asterisk (*) prompt and waits for you to enter DISKVERIFY commands. The following is an example:

```
-DISKVERIFY :F1: <CR>
iRMX <version> Disk Verify Utility,  Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
*
```

At the asterisk prompt, you can enter any of the DISKVERIFY commands listed in the DISKVERIFY COMMANDS chapter of this manual. If you enter anything else, the utility will display an error message.

1.4 INVOCATION ERROR MESSAGES

The following is a list of error messages you might encounter when invoking the Disk Verification Utility.

argument error	The option specified is not valid.
<logical name>, invalid logical name.	The logical name does not exist; was longer than 12 characters; contained invalid characters; or was missing a matching colon.
0045 : E$LOG_NAME_NEXIST or <logical name>, logical name does not exist	A nonexistent <logical name> was specified in either the :logical name: or outpath parameter.
<outpath>, 0038 : E$ALREADY_ATTACHED	The output was directed to a file on the volume being verified.
command syntax error	A syntax error was made when entering the command.
<logical name>, outstanding connections to the device have been deleted.	This warning is not fatal and will occur every time you try to verify the system device or any other volume on which files have been attached.
<logical name> or <outpath>, invalid wildcard specification	The logical name or output pathname contained a wildcard character.

<logical name>, can't attach device	The device cannot be attached and read.
device size inconsistent size in volume label = <value1> : computed size = <value2>	When the Disk Verification Utility computed the size of the volume, the size it computed did not match the information recorded in the iRMX volume label. The volume label may contain invalid or corrupted information. This is not a fatal error, but it is an indication that further error conditions may result during the verification session. You may have to reformat the volume or use the Disk Verification Utility to restore the volume label.
not a named disk	A NAMED, NAMED1, or NAMED2 verification was requested for a physical volume.
<partial logical name>, 0081: E$STRING_BUFFER	The logical name was longer than 14 characters in length, not including colons.
<logical name>, device does not belong to you	An attempt was made to verify a device that was attached by another user. For example, the system device is :SD: and USER is not the super user.
<logical name>, device size is zero	The logical name entered does not define a mass storage device. For example, you cannot perform DISKVERIFY on a line printer.

DISKVERIFY COMMANDS 2

2.1 INTRODUCTION

When the Disk Verification Utility issues the asterisk (*) prompt, you can enter DISKVERIFY commands to examine or change file structure information on the volume. This process usually involves reading a portion of the volume into a buffer, modifying that buffer, and writing the information back to the volume. This chapter describes the commands that enable you to perform these operations.

The commands in this chapter are presented in alphabetical order regardless of their function. The only exception is when two commands are similar, such as DISPLAYBYTE and DISPLAYWORD. In this case, the first command is explained in its alphabetical order, and the second command follows it with only the differences described.

The first occurrence of each command name is printed in blue ink and appears on the outside upper corner of the page; subsequent occurrences are printed in black ink. Blue or bolded text is also used to indicate an entry you make from your terminal.

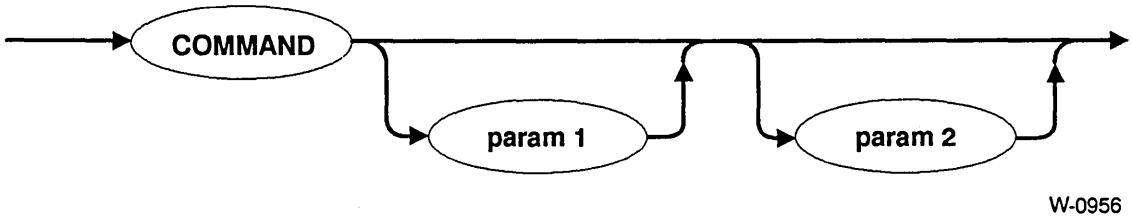
Before describing the individual commands, this chapter discusses command syntax, command names, parameters, input radices, and error messages. It also provides a command dictionary that gives a brief description of each command and the page number on which the command is found.

2.2 COMMAND SYNTAX

The syntax for each command described in this chapter is presented in a "railroad track" schematic, with syntactic elements scattered along the track. Your entrance to any given schematic is always from left to right, beginning with the command name entry.

Elements shown in uppercase characters must be typed in a command line exactly as shown in the schematic, however, you may enter them in either uppercase or lowercase. Syntactic elements shown in lowercase are generic terms, which means you must supply the specific item, such as the pathname of a file.

"Railroad sidings" go through optional parameter elements. In some cases, you have a choice of going through one of several sidings before returning to the main track. In still other cases, the main track itself diverges into two separate tracks, which means you must select one track or the other but not both. For example, a command that consists of a command name and two optional parameters would look like this:



You can enter this command in any one of these forms:

```
COMMAND
COMMAND param1
COMMAND param2
COMMAND param1 param2
```

The arrows are used here to illustrate the possible flow through the tracks. They do not appear in the schematics in the rest of this chapter.

2.3 COMMAND NAMES

When you enter a DISKVERIFY command, you can enter the command name or its abbreviation (listed in this chapter), or you can enter any unique portion of the command name. For example, when specifying the DISPLAYFNODE command, you can enter any of the following:

```
DISPLAYFNODE <fnodenumber>
DF <fnodenumber>
DISPLAYF <fnodenumber>
```

You can also enter any other partial form of the word DISPLAYFNODE that contains at least the characters DISPLAYF.

2.4 PARAMETERS

Several DISKVERIFY commands have parameters described as being in this form:

```
keyword = value
```

You can also enter these parameters in this form:

```
keyword (value)
```

For example, both of these specify a FREE command:

```
FREE FNODE = 10

FREE FNODE (10)
```

2.5 INPUT RADICES

DISKVERIFY always produces numerical output in hexadecimal format. You can provide input to DISKVERIFY in any one of the following three radices by including a radix character immediately after the number. The valid radix characters are

radix	character	example
hexadecimal	h or H	16h, 7CH
decimal	t or T	23t, 100T
octal	o, O, q, or Q	27o, 33Q

If you omit the radix character, DISKVERIFY assumes the number is hexadecimal.

2.6 ABORTING DISKVERIFY COMMANDS

You can abort the following DISKVERIFY commands by entering a CONTROL-C, which terminates the command and returns control to the Disk Verification Utility (not the Human Interface command level).

DISK
DISPLAYBYTE
DISPLAYDIRECTORY
DISPLAYFNODE
DISPLAYNEXTBLOCK
DISPLAYPREVIOUSBLOCK
DISPLAYWORD
EDITFNODE
EDITSAVEFNODE
FIX
GETBADTRACKINFO
LISTBADBLOCKS
SUBSTITUTEBYTE
SUBSTITUTEWORD
VERIFY

2.7 DISKVERIFY ERROR MESSAGES

Each DISKVERIFY command can generate a number of error messages, which indicate errors in the way the command was specified or problems with the volume itself. The following messages can be generated by many of the commands (each command description lists the error messages generated by the particular command):

block I/O error	The utility attempted to read or write a block on the volume and found that the block was physically damaged and therefore, could not complete the requested command. Or, an attempt was made to write a block to a disk volume that is write protected. The error message states whether read or write was performed and the number of the block causing the error.
command syntax error	A syntax error was made in a command.
illegal command	The command specified is not a valid DISKVERIFY command.

fnode file/space map file inconsistent	One of the files, R?SAVE or R?FNODEMAP, is damaged and DISKVERIFY cannot perform further verification.
argument error	The command was missing an argument, or the argument was illegally specified.
not a named disk	The device is not a named volume (a tape, for example) or the iRMX volume label, obtained when DISKVERIFY begins processing, contains invalid information. If the label contains invalid information, the utility (in some cases) can assume that a named volume is a physical volume. In this case, the commands that apply to named volumes only (such as DISPLAYFNODE, DISPLAYDIRECTORY, and VERIFY NAMED) issue this message. If you are sure the volume is a named volume, this message may indicate that the iRMX volume label is corrupted. (If the file was formatted with the RESERVE option of the FORMAT command, DISKVERIFY issues this message only if both volume labels are corrupted. When only the volume label is invalid, the duplicate in the save area is used.)
seek error	The utility unsuccessfully attempted to seek to a location on the volume. This error normally results from invalid information in the iRMX volume label or the fnodes, inserting a new volume after DISKVERIFY is invoked, or a defective disk.

2.8 COMMAND DICTIONARY

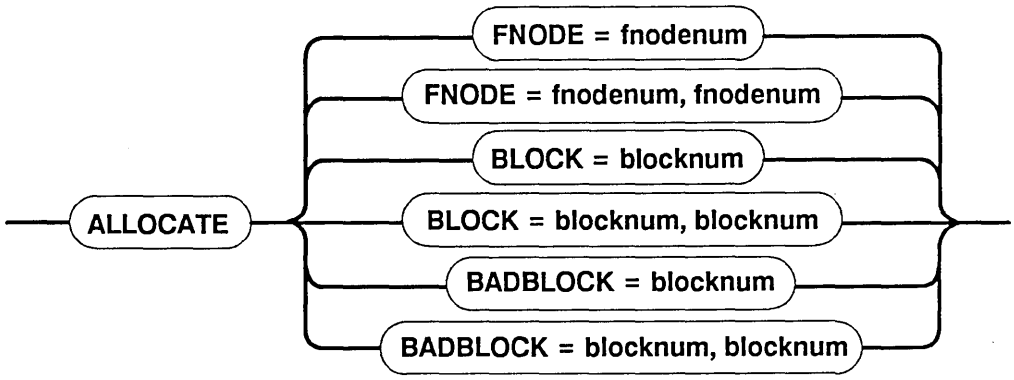
The command dictionary below lists the name and a brief description of each DISKVERIFY command, as well as the page numbers of more complete descriptions. Following each command name is its unique abbreviation, if any. For quick reference, you can also locate the command using the page headers remaining in this chapter.

Command	Synopsis	Page
ALLOCATE	Marks a particular fnode or volume block as allocated.	2-8
BACKUPFNODES BF	Copies current fnode file into a backup file named R?SAVE.	2-11
DISK	Displays the attributes of the volume being verified.	2-14
DISPLAYBYTE DB or D	Displays the working buffer in byte format.	2-17
DISPLAYWORD DW	Displays the working buffer in word format.	2-19
DISPLAYDIRECTORY DD	Displays directory contents.	2-21
DISPLAYFNODE DF	Displays the specified fnode information.	2-24
DISPLAYSAVEFNODE DSF	Displays the fields of a single fnode in the R?SAVE file.	2-29
DISPLAYNEXTBLOCK DNB or > or <CR>	Displays the "next" volume block.	2-30
DISPLAYPREVIOUSBLOCK DPB or <	Displays the "previous" volume block.	2-31
EDITFNODE EF	Edits the specified fnode.	2-32
EDITSAVEFNODE ESF	Edits the specified saved fnode.	2-35
EXIT E	Exits the Disk Verification Utility.	2-36

Command	Synopsis	Page
FIX	Verifies the disk and fixes inconsistencies.	2-37
FREE	Marks a particular fnode or volume block as free.	2-40
GETBADTRACKINFO GB	Displays the bad track information.	2-43
HELP H	Lists the DISKVERIFY commands.	2-45
LISTBADBLOCKS LBB	Displays all the bad blocks on the volume.	2-46
Miscellaneous Commands	Perform useful arithmetic and conversion functions; the commands include ADD, SUB, MUL, DIV, MOD, HEX, DEC, ADDRESS,and BLOCK.	2-48
QUIT Q	Exits the Disk Verification Utility.	2-54
READ R	Reads a volume block into the working buffer.	2-55
RESTOREFNODE RF	Copies one fnode (or range of fnodes) from the R?SAVE file to the fnode file.	2-57
RESTOREVOLUMELABEL RVL	Copies the duplicate volume label to the volume label offset on track 0.	2-60
SAVE	Writes the updated fnode map, free space map, and bad block map to the volume.	2-62
SUBSTITUTEBYTE SB or S	Modifies the contents of the working buffer in byte format.	2-65
SUBSTITUTEWORD SW	Modifies the contents of the working buffer in word format.	2-68
VERIFY V	Verifies the volume.	2-69
WRITE W	Writes the working buffer to the volume.	2-79

ALLOCATE

This command designates file descriptor nodes (fnodes) and volume blocks as allocated. You can also use this command to designate one or a range of volume blocks as "bad." The format of the ALLOCATE command is as follows:



INPUT PARAMETERS

fnodenum	Number of the fnode to allocate. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. Two fnode values separated by a comma signifies a range of fnodes.
blocknum	Number of the volume block to allocate. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. Two block numbers separated by a comma signifies a range of block numbers.

OUTPUT

If you are using ALLOCATE to allocate fnodes, ALLOCATE displays the following message:

```
<fnodenum>, fnode marked allocated
```

where <fnodenum> is the number of the fnode that the utility designated as allocated.

If you are using ALLOCATE to allocate volume blocks, ALLOCATE displays the following message:

```
<blocknum>, block marked allocated
```

where <blocknum> is the number of the volume block that the utility designated as allocated.

If you are using ALLOCATE to designate one or more volume blocks as "bad," ALLOCATE displays the following message:

```
<blocknum>, block marked bad
```

where <blocknum> is the number of the volume block that the utility designated as "bad." If this block was not allocated before you attempt to designate it as "bad," ALLOCATE also displays

```
<blocknum>, block marked allocated
```

ALLOCATE checks the allocation status of fnodes or blocks before allocating them. Therefore, if you specify ALLOCATE for a block or fnode already allocated, ALLOCATE returns one of the following messages:

```
<fnodenum>, fnode already marked allocated

<blocknum>, block already marked allocated

<blocknum>, block already marked bad
```

DESCRIPTION

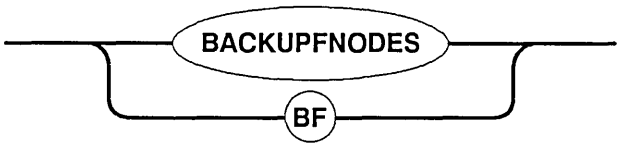
Fnodes are data structures on the volume that describe the files on the volume. They are created when the volume is formatted. An allocated fnode is one that represents an actual file. ALLOCATE designates fnodes as allocated by updating the FLAGS field of the fnode and free-fnodes-map file with this information.

An allocated volume block is a block of data storage that is part of a file; it is not available to be assigned to a new file. ALLOCATE designates volume blocks as allocated by updating the volume free-space-map with this information. When you use ALLOCATE to designate bad blocks, it not only updates the volume free-space-map but also marks an associated bit as "bad" in the bad blocks file.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
<fnodenum>, fnode out of range	The fnode number specified was larger than the largest fnode number in the volume.
no badblocks file	The volume does not have a bad blocks file. This message could appear if an earlier version of the Human Interface FORMAT command was used when the disk was formatted.

This command copies the current fnode file into a designated fnode backup file named R?SAVE. R?SAVE must have been reserved when the volume was formatted. (That is, the RESERVE option of the FORMAT command must have been specified.) The format of the BACKUPFNODES command is as follows:



W-0958

INPUT PARAMETERS

None.

OUTPUT

BACKUPFNODES displays the following message:

```
fnode file backed up to save area
```

DESCRIPTION

The BACKUPFNODES command ensures against data loss that occurs when the fnode file is damaged or destroyed. To use this command, you must have formatted the volume using the FORMAT command (V1.1 or later) to create a special reserve area (R?SAVE). A switch in the FORMAT command (the RESERVE switch) controls the creation of R?SAVE. If you did not specify the RESERVE parameter when the volume was formatted, the BACKUPFNODES command will be unable to copy the fnode file to R?SAVE. An error message will be returned indicating that no save area has been reserved. In this case, the volume must be reformatted if you wish to use the BACKUPFNODES command.

The FORMAT command writes the initialized copy of the fnode file into R?SAVE. Therefore, you do not have to use BACKUPFNODES to back up a newly formatted volume. Subsequently, you can routinely (for example, once a day) backup fnodes to assure that the data in R?SAVE matches the data in the fnode file. You can do this by using either the BACKUPFNODES command or the Human Interface SHUTDOWN command with the BACKUP option. (For more information on SHUTDOWN, see the *Operator's Guide to the iRMX® Human Interface*.)

NOTE

Be sure that the current fnode file is valid before executing the BACKUPFNODE command (using NAMED verification).

ERROR MESSAGES

argument error	When the command was entered, an argument was supplied. BACKUPFNODES does not accept an argument.
no save area was reserved when volume was formatted	The volume has not been formatted to support fnode backup. To allow future use of backupfnodes on this volume, you should invoke the Human Interface BACKUP command to save the data on the volume, reformat the volume with a save area (using the RESERVE option of the FORMAT command), and finally, restore the volume data.
not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
*verify NAMED <CR>
            .
            .
            .
   BIT MAPS O.K.
*backupfnodes <CR>   or bf <CR>
fnode file backed up to save area
*
```

DISK

This command displays the attributes of the volume being verified. You can abort this command by typing a CONTROL-C. The format of the DISK command is as follows:

W-0959

INPUT

None.

OUTPUT

The output of the DISK command depends on whether the volume is formatted as a physical or named volume. For a physical volume, the DISK command displays the following information:

```
        device name  = <devname>
    physical disk
device granularity  = <devgran>
        block size  = <devgran>
  number of blocks  = <numblocks>
       volume size  = <size>
```

where:

<devname>	Name of the device containing the volume. This is the physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devgran>	Granularity of the device, as defined in the Device Unit Information Block (DUIB) for the device. Refer to the *iRMX® Device Drivers User's Guide* for more information about DUIBs. For physical devices, this is also the volume block size.
<numblocks>	Number of volume blocks in the volume.
<size>	Size of the volume, in bytes.

For a named volume, the DISK command displays the following information:

```
             device name = <devname>
named disk, volume name = <volname>
     device granularity = <devgran>
            block size = <volgran>
       number of blocks = <numblocks>
  number of free blocks = <numfreeblocks>
           volume size = <size>
            interleave = <inleave>
        extension size = <xsize>
       number of fnodes = <numfnodes>
  number of free fnodes = <numfreefnodes>
           root fnode = <rootfnode>
      save area reserved = (yes/no)
MSA second stage included = (yes/no)          <-- Appears in iRMX II
                                                  only
```

The <devname>, <devgran>, <numblocks>, and <size> fields are the same as for physical files. The remaining fields are as follows:

<volname>	Name of the volume, as specified when the volume was formatted.
<volgran>	Volume granularity, as specified when the volume was formatted.
<numfreeblocks>	Number of available volume blocks in the volume.
<inleave>	The interleave factor for a named volume.
<xsize>	Size, in bytes, of the extension data portion of each file descriptor node (fnode).
<numfnodes>	Number of fnodes in the volume. The fnodes were created when the volume was formatted.
<numfreefnodes>	Number of available fnodes in the named volume.
<rootfnode>	The number of the fnode that contains the volume's root directory.
save area reserved	Indicates whether the R?SAVE file is reserved for volume label and fnode file backups.
MSA second stage included	Indicates whether the MULTIBUS® II System Architecture second stage bootstrap loader (R?SECONDSTAGE) is present on the disk. This field only appears in the iRMX II version of the Disk Verification Utility.

Refer to Appendix A of this manual or to the description of the FORMAT command in the *Operator's Guide to the iRMX® Human Interface* for more information about the named disk fields.

DESCRIPTION

The DISK command displays the attributes of the volume. The format of the output from DISK depends on whether the volume is formatted as a named or physical volume.

ERROR MESSAGES

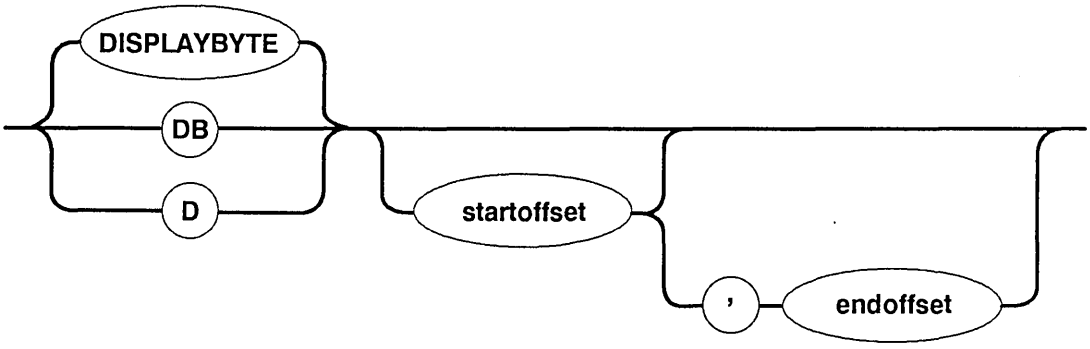
None.

EXAMPLE

The following example shows the output of the DISK command for an 5.25-inch diskette.

```
super- diskverify :f0: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved

*disk  <CR>

                  device name = wmf0
        named disk, volume name = rmx286
             device granularity = 0200
                      block size = 0200
               number of blocks = 0000027C
         number of free blocks = 000001E9
                   volume size = 0004F800
                    interleave = 0005
                extension size = 03
              number of fnodes = 00CF
         number of free fnodes = 00BE
                    root fnode = 0006
           save area reserved = no
     MSA second stage included = no          <-- Appears in iRMX II
                                                 only
```

This command displays the specified portion of the working buffer in BYTE format. It displays the buffer in 16-byte rows. You can abort this command by typing a CONTROL-C. The format of the DISPLAYBYTE command is as follows:



INPUT PARAMETERS

startoffset	Number of the byte, relative to the start of the buffer, that begins the display. DISPLAYBYTE starts the display with the row containing the specified offset. If you omit this parameter and the endoffset parameter, DISPLAYBYTE displays the entire working buffer.
endoffset	Number of the byte, relative to the start of the buffer, that ends the display. If you omit this parameter, DISPLAYBYTE displays only the row indicated by startoffset. However, if you omit both startoffset and endoffset, DISPLAYBYTE displays the entire working buffer.

OUTPUT

In response to the command, DISPLAYBYTE displays the specified portion of the working buffer in rows, with 16 bytes displayed in each row. Figure 2-1 illustrates the format of the display.

As Figure 2-1 shows, DISPLAYBYTE begins by listing the block number where data resides in the working buffer. It then lists the specified portion of the buffer, providing the column numbers as a header and beginning each row with the relative address of the first byte in the row. It also includes, at the right of the listing, the ASCII equivalents of the bytes, if the ASCII equivalents are printable characters. (If a byte is not a printable character, DISPLAYBYTE displays a period in the corresponding position.)

```
*displaybyte 7,13 <CR>

BLOCK NUMBER = blocknum

offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII STRING
 0000  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   ............
 0010  61 6E 20 65 78 61 6D 70 6C 65 20 20 20 20 20 20   an example
```

Figure 2-1. DISPLAYBYTE Format

DESCRIPTION

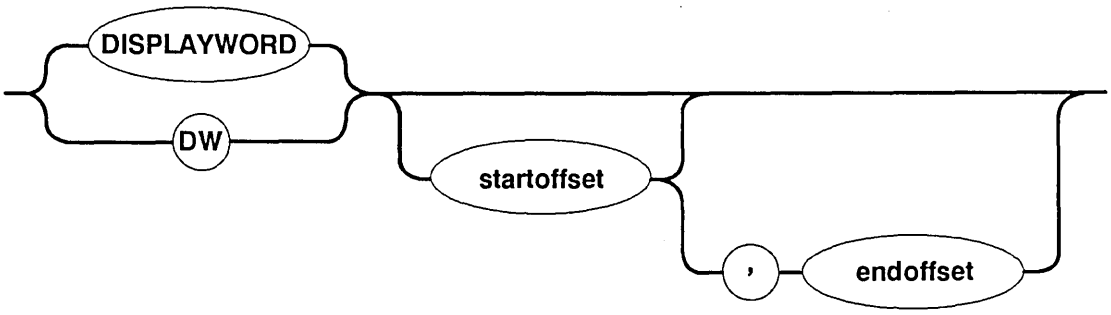
DISKVERIFY maintains a working buffer for READ and WRITE commands. The size of the buffer is equal to the volume's granularity value. After you read a volume block of memory into the working buffer with the READ command, you can display part or all of that buffer, in BYTE format, by entering the DISPLAYBYTE command. DISPLAYBYTE displays the hexadecimal value for each byte in the specified portion of the buffer.

If you omit all parameters, DISPLAYBYTE displays the entire block stored in the working buffer.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in one of the offset parameters.
<offset>, invalid offset	Either a larger value was specified for startoffset than for endoffset or an offset value larger than the number of bytes in the block was specified.

This command is the same as the DISPLAYBYTE command, except that it displays the working buffer in WORD format, 8-words per row. The format of the DISPLAYWORD command is as follows:



W-0961

EXAMPLES

Assuming that the volume granularity is 128 bytes and that you have read block 20H into the working buffer with the READ command, the following command displays that block in WORD format.

```
*DISPLAYWORD   <CR>

BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0000    0000  0000  0000  0000  0000  0000  0000  0000
 0010    0000  0080  0000  0000  0000  0001  FF0F  00FF
 0020    0000  0000  0500  0000  0000  0025  0108  FFFF
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
 0050    0000  0000  0000  0000  0000  0000  0000  0000
 0060    0000  0000  0000  0000  0000  0000  0080  0000
 0070    0000  0000  0001  FF0F  00FF  0000  0000  0500
*
```

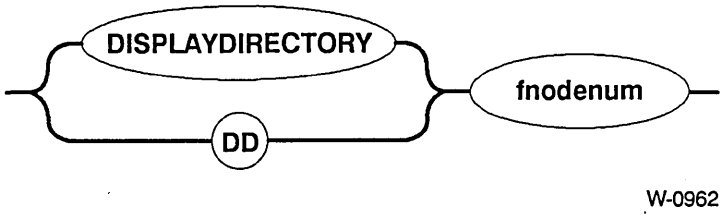
The following command displays the portion of the block that contains the offsets 31H through 45H (words beginning at odd addresses).

```
*DW 31, 45  <CR>
BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0031    001F  2E00  0000  2500  001F  2B00  0000  0100
 0041    0000  0100  8000  0000  0000  0000  0000  0000
*
```

The following command displays the portion of the block that contains the offsets 30H through 45H (words beginning at even addresses).

```
* DISPLAYWORD 30, 45 <CR>
BLOCK NUMBER = 20

offset     0     2     4     6     8     A     C     E
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
*
```

This command lists all the files contained in a directory. You can abort this command by typing a CONTROL-C. The format of the DISPLAYDIRECTORY command is as follows:



INPUT PARAMETER

fnodenum	Number of the fnode that corresponds to a directory file. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. DISPLAYDIRECTORY lists all files or directories contained in this directory.

OUTPUT

In response to the command, DISPLAYDIRECTORY lists information about all files contained in the specified directory. The format of this display is as follows:

```
FILE NAME  FNODE   TYPE    FILE NAME FNODE   TYPE    FILE NAME FNODE   TYPE

<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
          .                         .                         .
          .                         .                         .
          .                         .                         .
```

where:

<filenam>	Name of the file or directory contained in the directory.
<fnode>	Number of the fnode that describes the file.

<type> Type of the file. The <type> can be

Type of file	Description
DATA	data files
DIR	directory files
SMAP	volume free space map
FMAP	free fnodes map
BMAP	bad blocks map
VLAB	volume label file
****	indicates an illegal fnode type

DESCRIPTION

DISPLAYDIRECTORY displays a list of files contained in the specified directory, along with their fnode numbers and types. You can then use other DISKVERIFY commands to examine the individual files.

ERROR MESSAGES

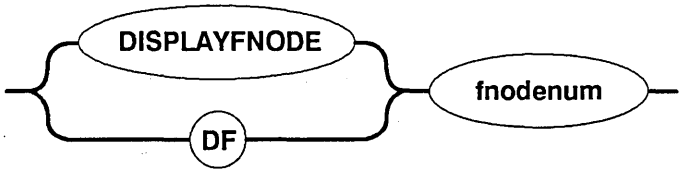
argument error	A nonnumeric character was specified in the fnodenum parameter.
<fnodenum>, fnode not allocated	The number specified for the fnodenum parameter does not correspond to an allocated fnode. This fnode does not represent an actual file.
<fnodenum>, not a directory fnode	The number specified for the fnodenum parameter is not an fnode for a directory file.
<fnodenum>, fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.

EXAMPLE

The following command lists the files contained in the directory whose fnode is fnode 6.

```
*DISPLAYDIRECTORY 6  <CR>

     FILE NAME  FNODE TYPE  FILE NAME  FNODE TYPE     FILE NAME  FNODE TYPE
    R?SPACEMAP   0001 SMAP R?FNODEMAP   0002 FMAP R?BADBLOCKMAP   0004 BMAP
 R?VOLUMELABEL   0005 VLAB     R?SAVE   0007 DATA        RMX286   0008 DIR
        MYFILE   0009 DATA   YOURFILE   000A DATA       ONEFILE   000B DATA
*
```

This command displays the fields associated with an fnode. You can abort this command by typing a CONTROL-C. The format of the DISPLAYFNODE command is as follows:



INPUT PARAMETER

fnodenum	Number of the fnode to be displayed. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.

OUTPUT

In response to this command, DISPLAYFNODE displays the fields of the specified fnode. The format of the display is as follows:

```
Fnode number = <fnodenum>
path name: <pathname>
                    flags : <flgs>
                     type : <typ>
      file gran/vol gran : <gran>
                    owner : <own>
 create,access,mod times : <crtime>, <acctime>, <modtime>
   total size,total blks : <totsize>, <totblks>
       block pointer (1) : <blks>, <blkptr>
       block pointer (2) : <blks>, <blkptr>
       block pointer (3) : <blks>, <blkptr>
       block pointer (4) : <blks>, <blkptr>
       block pointer (5) : <blks>, <blkptr>
       block pointer (6) : <blks>, <blkptr>
       block pointer (7) : <blks>, <blkptr>
       block pointer (8) : <blks>, <blkptr>
              this size  : <thissize>
               id count  : <count>
            accessor (1) : <access>, <id>
            accessor (2) : <access>, <id>
            accessor (3) : <access>, <id>
        parent, checksum : <prnt>, <checksum>
                 aux(*)  : <auxbytes>
```

where:

<fnodenum> Number of the fnode being displayed. If the fnode does not describe an actual file (that is, if it is not allocated), the following message appears next to this field:

```
*** ALLOCATION STATUS BIT IN THIS FNODE NOT SET ***
```

In this case, the fnode fields are normally set to zero.

<pathname> Full pathname of the file described by the fnode. This field is not displayed if the fnode does not describe a file.

<flgs> A word defining the attributes of the file. Significant bits of this word are as follows:

Bit	Meaning
0	Allocation status. This bit is set to 1 for allocated fnodes and 0 for free fnodes.
1	Long or short file attribute. This bit is set to 1 for long files and 0 for short files.
5	Modification attribute. This bit is set to 1 whenever a file is modified.
6	Deletion attribute. This bit is set to 1 to indicate a temporary file or a file to be deleted.

The DISPLAYFNODE command displays a message next to this field to indicate whether the file is a long or short file.

<typ> Type of file. This field contains a value and a description which is displayed next to the value. The possible values and descriptions are as follows:

Value	Descriptions
00	fnode file
01	volume map file
02	fnode map file
03	account file
04	bad block file
06	directory file
08	data file
09	volume label file
any other value	illegal value

<gran>	File granularity, specified as a multiple of the volume granularity.
<own>	User ID of the owner of the file.
<crtime> <acctime> <modtime>	Time and date of file creation, last access, and last modification. These values are expressed as the time, in seconds, since midnight (00:00) on January 1, 1978.
<totsize>	Total size, in bytes, of the actual data in the file.
<totblks>	Total number of volume blocks used by the file, including indirect block overhead.
<blks>, <blkptr>	Values that identify the data blocks of the file. For short files, each <blks> parameter indicates the number of volume blocks in the data block, and each <blkptr> is the number of the first such volume block. For long files, each <blks> parameter indicates the number of volume blocks pointed to by an indirect block, and each <blkptr> is the block number of the indirect block.
<thissize>	Size in bytes of the total data space allocated to the file, minus any space used for indirect blocks.
<count>	Number of user IDs associated with the file.
<access>, <id>	Each pair of fields indicates the access rights for the file and the ID of the user who has that access ID. Bits in the <access> field are set to indicate the following access rights:

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	list
2	append	add entry
3	update	change entry

The first ID listed is the owner's ID.

<prnt>	Fnode number of the directory file that contains the file.
<checksum>	Checksum of the fnode.
<auxbytes>	Auxiliary bytes associated with the file.

Appendix A contains a more detailed description of the fnode fields.

DESCRIPTION

Fnodes are system data structures on the volume that describe the files on the volume. The fnode structures are created when the volume is formatted. Each time a file is created on the volume, the Basic I/O System allocates an fnode for the file and fills in the fnode fields to describe the file. The DISPLAYFNODE command enables you to examine these fnodes and determine where the data for each file resides.

ERROR MESSAGES

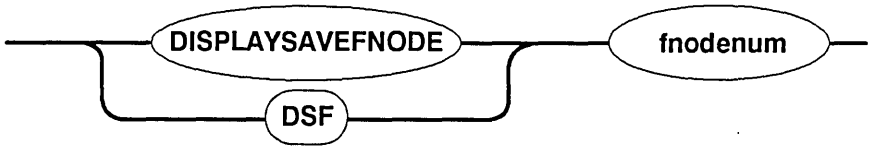
argument error	The value entered for the fnodenum parameter was not a legitimate fnode number.
<fnodenum>, fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.
Unable to get pathname - <reason>	The pathname specified could not be retrieved. Possible causes of this error are seek error, I/O error, or invalid parent, or insufficient memory.

EXAMPLE

The following example displays fnode 10 of a volume. This fnode represents a directory.

```
* DISPLAYFNODE 10   <CR>

Fnode number = 10
path name : /MYDIR
                   flags : 0025 =>short file
                    type : 06 =>directory file
      file gran/vol gran : 01
                   owner : FFFF => world
 create,access,mod times : 10219017, 10219E58, 10219E58
 total size,total blocks : 00000360, 00000001
       block pointer (1) : 0001, 000050
       block pointer (2) : 0000, 000000
       block pointer (3) : 0000, 000000
       block pointer (4) : 0000, 000000
       block pointer (5) : 0000, 000000
       block pointer (6) : 0000, 000000
       block pointer (7) : 0000, 000000
       block pointer (8) : 0000, 000000
              this size : 00000400
               id count : 0001
           accessor (1) : 0F, FFFF
           accessor (2) : 00, 0000
           accessor (3) : 00, 0000
        parent, checksum : 0006, 796D
                 aux(*) : 000000
*
```

This command is identical to DISPLAYFNODE, except the DISPLAYSAVEFNODE takes the fnode information from the R?SAVE file, and displays the fnode as saved. R?SAVE must have been reserved when the volume was formatted. (That is, the RESERVE option in the FORMAT command must have been specified.) The format of the DISPLAYSAVEFNODE command is as follows:



OUTPUT

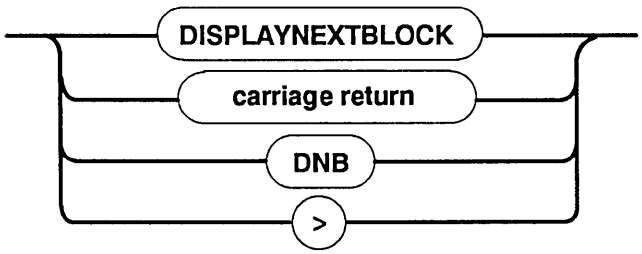
The output is identical to DISPLAYFNODE except for the first line, which indicates that the fnode is saved. The format of the first line is as follows:

```
Fnode number = <fnodenum> (saved)
               .
               .
               .
```

ERROR MESSAGES

argument error	When the command was entered, no argument was supplied. DISPLAYSAVEFNODE requires a designation of the fnode number.
<fnodenum>, fnode out of range	The number specified for the fnodenum parameter is larger than the largest fnode number on the volume.
no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.
Unable to get pathname - <reason>	The pathname specified could not be retrieved. Possible causes of this error are seek error, I/O error, invalid parent, or insufficient memory.

This command displays the "next" volume block. (The "next" volume block is the block immediately following the block currently in the working buffer.) The display format can be either WORD or BYTE. The utility remembers the mode in which you displayed the volume block currently in the working buffer, and it displays the next block in that format. So, if you used DISPLAYBYTE to display the current volume block, the next volume block appears in BYTE format; if you used DISPLAYWORD, the next volume block appears in WORD format. DISPLAYNEXTBLOCK uses the BYTE format as a default if you have not yet displayed a volume block. You can abort this command by typing a CONTROL-C. The format of the DISPLAYNEXTBLOCK command is as follows:



W-0965

OUTPUT

In response to the command, DISPLAYNEXTBLOCK reads the "next" volume block into the working buffer and displays it on the screen.

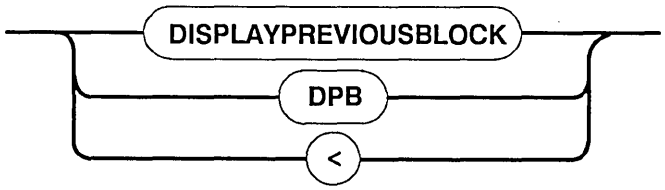
DESCRIPTION

The DISPLAYNEXTBLOCK command copies the "next" volume block from the volume to the working buffer and displays it at your terminal. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back out to the volume.

NOTE

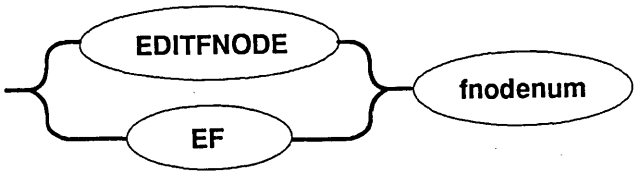
If you specify the DISPLAYNEXTBLOCK command at the end of the volume, the utility "wraps around" and displays the first block in the volume.

This command is identical to DISPLAYNEXTBLOCK, except that it displays the volume block preceding the current block in the working buffer. The format of the DISPLAYPREVIOUSBLOCK command is as follows:



W-0966

This command allows you to edit values within a specified fnode. It can be aborted by entering CONTROL-C. The format of the EDITFNODE command is



INPUT PARAMETER

fnodenum — Number of the fnode to edit. This number can be in the range of 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.

OUTPUT

When EDITFNODE is invoked it displays the following message:

```
Fnode number = nnnn
```

where nnnn is the number of the fnode you want to edit. The first field of the fnode is displayed with its current value, as follows:

```
flags(xxxx):
```

where xxxx is the current value of the flags field. From this point on, you can edit the fnode fields, one at a time. After the last fnode field has been edited or a "Q" has been entered while in edit mode, the following query appears on the screen and the modified fnode is displayed.

```
Write back?
```

A response of "Yes" causes the fnode with the modified values to be written on the volume and the following message to be displayed:

```
Fnode has been updated
```

Any other response causes the fnode to remain unchanged and the following message to be displayed:

```
Fnode not changed
```

DESCRIPTION

EDITFNODE is used to change values within a specified fnode. When it is invoked, it displays the message shown above. Once you receive the invocation message, you can edit the fnode, one field at a time. The first field, flags, is displayed upon invocation (as shown above). The current value of each field is displayed followed by a colon. EDITFNODE then waits for one of the following responses from the terminal.

Response	Meaning
<CR>	No modification to the field.
numerical value <CR>	The new value to be assigned. This value is always interpreted as hexadecimal.
QUIT or Q or q <CR>	Skip the remaining fields and display the query.

Any response, other than those listed above, causes the field to remain unchanged, and the next field to be displayed.

Once the fnode has been updated, you can use DISPLAYFNODE to examine the contents of the fnode and the changes you made. Changing the contents of an fnode causes it to have a bad checksum. Use FIX with the NAMED1 option to correct it. For more details, see the explanation of FIX later in this chapter.

ERROR MESSAGES

argument error	The option specified is not valid.
<fnode num>, fnode out of range	The fnode number specified was larger than the largest fnode number on the volume.
Error in Input	Invalid input was entered while editing an entry.

EXAMPLE

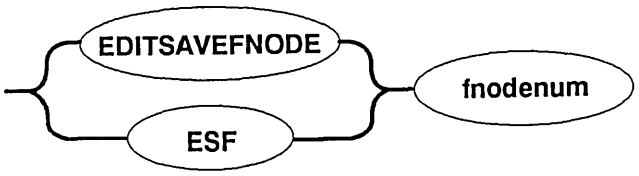
The following example illustrates using EDITFNODE to edit fnode 10.

```
* editfnode 10 <CR>
fnode number = 10
flags(0025): <CR>
type(0006): <CR>
file gran/vol gran(01):  <CR>
owner(0FFFF):   0 <CR>
create time(10219CB2):  q <CR>
```

Entering "q" causes the modified fnode to be displayed.

```
                    flags : 0025 =>short file
                     type : 06 =>directory file
          file gran/vol gran : 01
                    owner : 0000
 create,access,mod times : 10219CB2, 10219CC8, 10219CC8
 total size,total blocks : 00000360, 00000001
       block pointer (1) : 0001, 000050
       block pointer (2) : 0000, 000000
       block pointer (3) : 0000, 000000
       block pointer (4) : 0000, 000000
       block pointer (5) : 0000, 000000
       block pointer (6) : 0000, 000000
       block pointer (7) : 0000, 000000
       block pointer (8) : 0000, 000000
              this size : 00000400
               id count : 0001
            accessor (1) : 0F, FFFF
            accessor (2) : 00, 0000
            accessor (3) : 00, 0000
        parent, checksum : 0006, 0000
                aux(*) : 000000
Write back?  yes  <CR>
Fnode has been updated
*
```

EDITSAVEFNODE is identical to EDITFNODE, except that it allows you to edit an fnode from the R?SAVE file. (R?SAVE must have been reserved when the volume was formatted.) In addition, it designates the fnode as saved when displaying the fnode number. You can abort this command by entering CONTROL-C. The format of the EDITSAVEFNODE command is

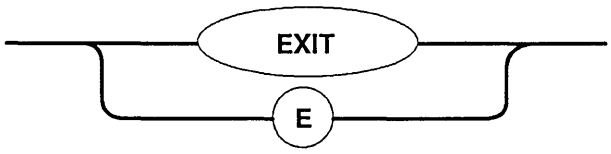


ERROR MESSAGES

The error messages are the same as in EDITFNODE with the addition of the following message.

no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.

This command exits the Disk Verification Utility and returns control to the Human Interface command level. The format of the EXIT command is as follows:



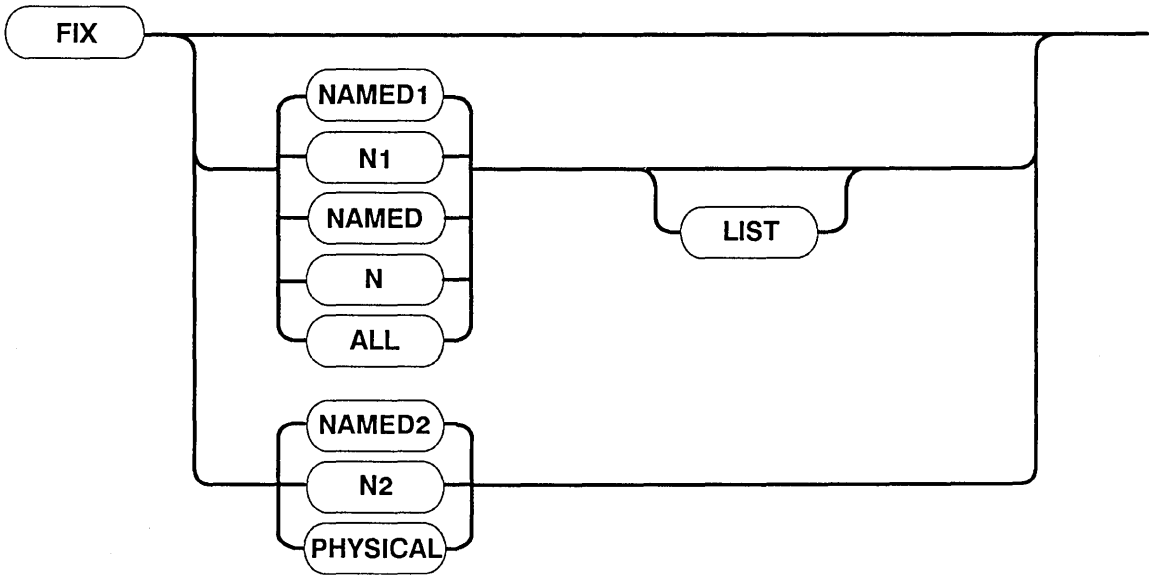
This command is identical to the QUIT command.

NOTE

Although you can use DISKVERIFY to verify the system device (:SD:), note that all connections to this device are deleted by the operating system. After exiting, you must reboot the system or use the warm start feature (see the *iRMX® System Debugger Reference Manual*).

This command verifies the volume in the same way as the VERIFY command to determine if the data on the volume is consistent. In addition, this command "fixes" various kinds of inconsistencies discovered during verification. You can abort this command by entering CONTROL-C. (CONTROL-C is ignored when FIX is writing to the volume in order to prevent inconsistencies on the volume.)

Because FIX and VERIFY perform the same verification functions and generate the same error messages, the command description given below describes only the additional functions of FIX. For a complete explanation of the verify functions, see the VERIFY command described later in this chapter. The format of the FIX command is:



INPUT PARAMETERS

ALL — Performs all operations appropriate to the volume. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs only the PHYSICAL verification function. For both NAMED and PHYSICAL volumes, ALL performs the fixes for the relevant verifications.

LIST	Lists the file information displayed in Figure 2-3 (see the VERIFY command description later in this chapter) for any verification that includes NAMED1.
NAMED1 or N1	Performs NAMED1 verification and fixes the following inconsistencies: • Fixes bad checksums • Attaches orphan fnodes to their parents. An orphan fnode is an fnode contained within a directory and whose parent field does not point back to this directory. If the parent field of the specified fnode points to a second valid directory, and the second directory also points to the fnode, no fix is performed since the specified fnode belongs to an existing directory. This is a case of multiple references (discussed in NAMED2). If the parent field does not point to a valid parent, the parent field is fixed to point to the directory that contains this fnode in its file list.
NAMED2 or N2	Performs NAMED2 verification and fixes the following inconsistencies: • Removes fnodes from their illegal parents. If there is a multiple reference to an fnode, the fnode is removed from the directories that it does not point to (if FIX was performed with NAMED1, the fnode should now point to one valid parent). • Saves fnode and block bit maps on completion of NAMED2.
NAMED or N	Performs both the NAMED1 and NAMED2 verification functions on a named volume and fixes the inconsistencies defined for these options.
PHYSICAL	Performs PHYSICAL verification and saves the bad block bit map.

OUTPUT

FIX produces the same output as the VERIFY command (see Figures 2-3, 2-4, and 2-5) with additional messages displayed when an inconsistency is fixed. NAMED1 output includes these messages.

```
Checksum Fixed
fnode nnnn was attached to parent nnnn
```

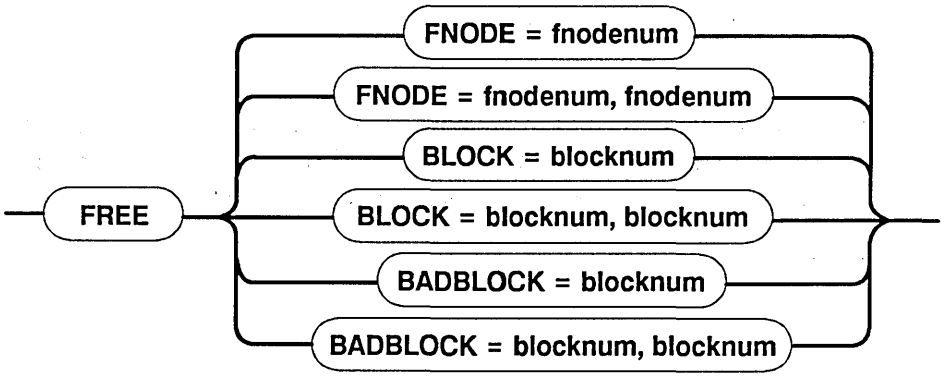
The first message appears after a bad checksum is fixed. The second message is displayed when the parent field of an fnode is modified to point to a valid parent.

NAMED2 displays this message when an fnode with multiple references is removed from the directory.

```
fnode removed from this directory
```

If an fnode exists on a disk and is marked allocated, but has not been referenced, FIX issues a warning message and asks if you want to save the bit maps. This prevents SAVE from freeing this fnode and its blocks and possibly causing a file to be lost.

This command designates fnodes and volume blocks as free (unallocated). It also removes volume blocks from the bad blocks file. The format of the FREE command is as follows:



INPUT PARAMETERS

fnodenum	Number of the fnode to free. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. Two fnode values separated by a comma signify a range of fnodes.
blocknum	Number of the volume block to free. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. Two block numbers separated by a comma signify a range of block numbers.

OUTPUT

If you are using FREE to deallocate fnodes, FREE displays the following message:

```
<fnodenum>, fnode marked free
```

where <fnodenum> is the number of the fnode that the utility designated as free.

If you are using FREE to deallocate volume blocks, FREE displays the following message:

```
<blocknum>, block marked free
```

where <blocknum> is the number of the volume block that the utility designated as free.

If you are using FREE to designate one or more "bad" blocks as "good," FREE displays the following message:

```
<blocknum>, block marked good
```

where <blocknum> is the number of the volume block that the utility designated as "good."

FREE checks the allocation status of fnodes or blocks before freeing them. Therefore, if you specify FREE for a block or fnode that is already unallocated, FREE returns one of the following messages:

```
<fnodenum>, fnode already marked free

<blocknum>, block already marked free

<blocknum>, block already marked good
```

DESCRIPTION

Free fnodes are fnodes for which no actual files exist. FREE designates fnodes as free by updating both the FLAGS field of the fnode and the free fnodes map file.

Free volume blocks are blocks that are not part of any file; they are available to be assigned to any new or current file. FREE designates volume blocks as free by updating the volume free space map.

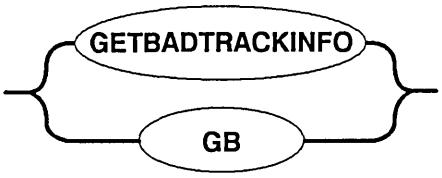
When you use the FREE command to designate one or more bad blocks as "good," it removes the block number from the bad blocks file. However, FREE BADBLOCK does not designate the blocks as free. To update the volume free space map and designate these blocks as free, use the FREE BLOCK command.

ERROR MESSAGES

argument error	A syntax error was made in the command or a nonnumeric character was specified in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.

<fnodenum>, fnode out of range	The fnode number specified was larger than the largest fnode number in the volume.
no badblocks file	The volume does not have a bad blocks file. This message could appear because an earlier version of the Human Interface FORMAT command was used when the disk was formatted.
not a named disk	FREE was performed on a physical volume.

This command displays the volume's bad track information. It can be aborted by entering CONTROL-C. The format of GETBADTRACKINFO is



W-0972

INPUT PARAMETERS

None.

OUTPUT

The GETBADTRACKINFO command displays the volume's bad track information as written by the manufacturer or the Human Interface FORMAT command. The output displayed by the GETBADTRACKINFO command is compatible with the format required by the Human Interface FORMAT command when writing bad track information on the disk. To use the output as input to FORMAT, remove the first two lines, leaving only the actual bad track information data. The display is as follows:

```
Bad track information:
cyl   head  sector
cccc  hh    ss
cccc  hh    ss
.     .     .
.     .     .
```

where cccc is cylinder number, hh is the head number and ss is the sector number (always zero for all devices supported in this release of the operating system).

As mentioned above, the output of the GETBADTRACKINFO command can be used as input to the FORMAT command when creating the bad track information file. The example below shows how to use GETBADTRACKINFO this way.

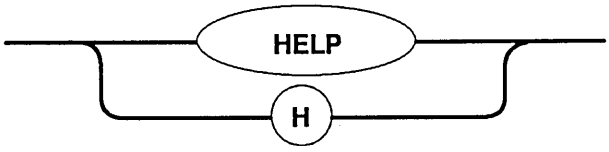
```
- attachdevice wmf0 as :w: <CR>
- diskverify :sd: to :w:bad.list <CR>
* getbadtrackinfo <CR>
* exit <CR>
```

After exiting DISKVERIFY and rebooting the system, edit :w:bad.lst and remove the header lines. The file can then be used as input to the bad track information file created by the FORMAT command.

ERROR MESSAGES

I/O error while trying to read bad track information	An I/O error occurred while reading the bad track information.
No valid bad track info found	Bad track information is not valid and cannot be displayed.
No bad track info found	The area designated for bad track information is empty.

This command lists all available Disk Verification Utility commands and provides a short description of each command. The format of the HELP command is



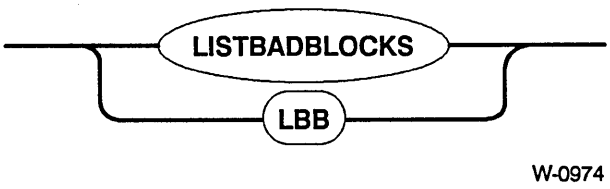
W-0973

OUTPUT

In response to this command, HELP displays the following information:

```
* help
                  allocate/free : allocate/free fnodes, space blocks, bad blocks
 backup/restore fnodes (bf/rf) : backup/restore fnode file to/from save area
                      Control-C : abort the command in progress
                           disk : display disk attributes
   display byte/word (d,db/dw) : display the buffer in (byte/word format)
        display directory (dd) : display the directory contents
            display fnode (df) : display fnode information
    display next block (>,dnb) : read and display 'next' volume block
display previous block (<,dpb) : read and display 'previous' volume block
      display save fnode (dsf) : display saved fnode information
                     exit,quit : quit disk verify
          list bad blocks (lbb) : list bad blocks on the volume
                      read (r) : read a disk block into the buffer
    restore volume label (rvl) : copy volume label from save area
                          save : save free fnodes, free space & bad block maps
substitute byte/word (s,sb/sw) : modify the buffer (byte/word format)
                        verify : verify the disk
                     write (w) : write to the disk block from the buffer
               edit fnode (ef) : edit an fnode
         edit save fnode (esf) : edit a saved fnode
                           fix : perform various fixes on the volume
       get bad track info (gb) : get the bad track info on the volume
misc commands-
                       address : convert block number to absolute address
                         block : convert absolute address to block number
                       hex/dec : display number as hexadecimal/decimal number
   add,+,sub,-,mul,*,div,/,mod : arithmetic operations on unsigned numbers
```

This command displays all the bad blocks on a named volume. You can abort this command by typing a CONTROL-C. The format of the LISTBADBLOCKS command is as follows:



OUTPUT

In response to this command, LISTBADBLOCKS displays up to eight columns of block numbers that you specified as "bad." Figure 2-2 illustrates the format of the display.

```
Badblocks on Volume:  volumenum

<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>

<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>
    .           .           .           .           .           .
    .           .           .           .           .           .
    .           .           .           .           .           .
<blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>  <blocknum>
```

Figure 2-2. LISTBADBLOCKS Format

If none of the blocks have been marked as "bad", LISTBADBLOCKS displays the following message:

```
no badblocks
```

NOTE

Bad tracks and bad blocks are different. Bad tracks are handled by the device drivers in conjunction with the hardware, whereas, bad blocks are handled by the Basic I/O System.

ERROR MESSAGES

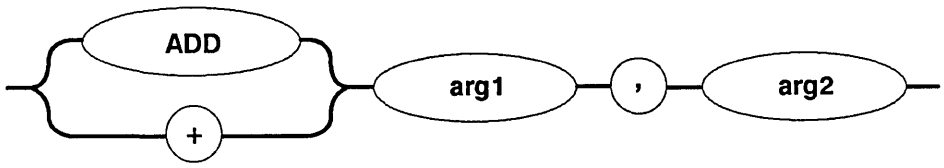
no badblocks file	The volume does not have a bad blocks file. This message could appear because an earlier version of the Human Interface FORMAT command was used when the disk was formatted or because the disk is a physical volume.

The following commands provide you with the ability to perform arithmetic and conversion operations within the Disk Verification Utility. The commands perform the operations on unsigned numbers only and do not report any overflow conditions. When the number is displayed in both hexadecimal and decimal format, it appears in hexadecimal format first, followed by the decimal number in parentheses. For example:

```
13 (  19T)
```

ADD

This command adds two numbers together. Its format is



W-0975

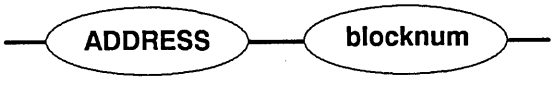
where:

arg1 and arg2 — Numbers the command adds together. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned sum of the two numbers in both hexadecimal and decimal format.

ADDRESS

All memory in a volume is divided into volume blocks, which are areas of memory the same size as the volume granularity. Volume blocks are numbered sequentially in the volume, starting with the block containing the smallest addresses (block 0). The ADDRESS command converts a block number into an absolute address (in hexadecimal) on the volume, so that you don't have to perform this conversion by hand. The format of this command is



W-0976

where:

blocknum	Volume block number that ADDRESS converts into an absolute address in hexadecimal. This parameter can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

In response, ADDRESS displays the following information:

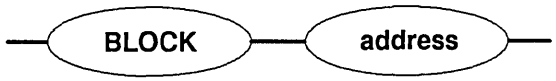
```
absolute address = <addr>
```

where:

<addr>	Absolute address in hexadecimal that corresponds to the specified block number. This address represents the number of the byte that begins the block and can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

BLOCK

The BLOCK command is the inverse of the address command. It converts a 32-bit absolute address (in hexadecimal) into a volume block number, so that you don't have to perform this conversion by hand. The format of this command is



W-0977

where:

address	Absolute address in hexadecimal that BLOCK converts into a block number. This parameter can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

In response, BLOCK displays the following information:

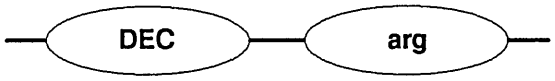
```
block number = <blocknum>
```

where:

<blocknum>	Number of the volume block that contains the specified absolute address in hexadecimal. The BLOCK command determines this value by dividing the absolute address by the volume block size and truncating the result.

DEC

This command finds the decimal equivalent of a number. Its format is



W-0978

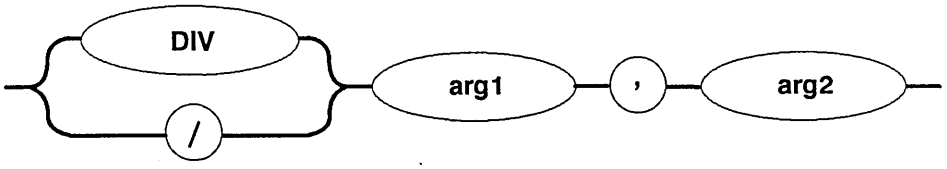
where:

arg	Number for which the command finds the decimal equivalent. The value of the argument cannot be greater than 2^{32}-1. The default base is in hexadecimal.

In response, the command displays the decimal equivalent of the specified number.

DIV

This command divides one number by another. Its format is



W-0979

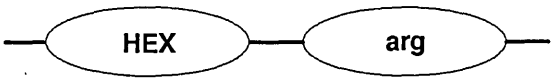
where:

arg1 and arg2	Numbers on which the command operates. It divides arg1 by arg2. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned integer quotient in both hexadecimal and decimal format.

HEX

This command finds the hexadecimal equivalent of a number. Its format is



W-0980

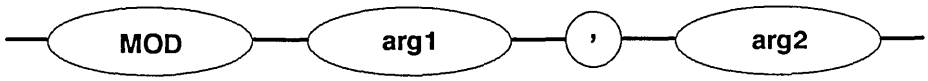
where:

arg — Number for which the command finds the hexadecimal equivalent. If you are specifying a decimal number, you must specify a "T". The value of the argument cannot be greater than 2^{32}-1.

In response, the command displays the hexadecimal equivalent of the specified number.

MOD

This command finds the remainder of one number divided by another. Its format is



W-0981

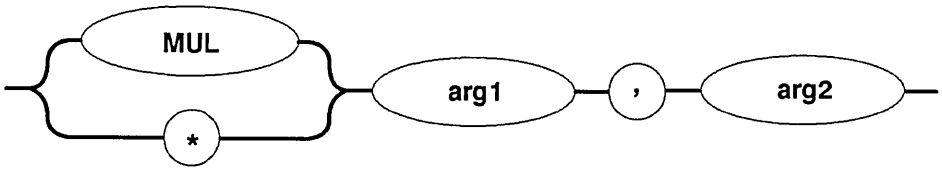
where:

arg1 and arg2 — Numbers on which the command operates. It performs the operation arg1 modulo arg2. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the value arg1 modulo arg2 in both hexadecimal and decimal format.

MUL

This command multiplies two numbers together. Its format is



W-0982

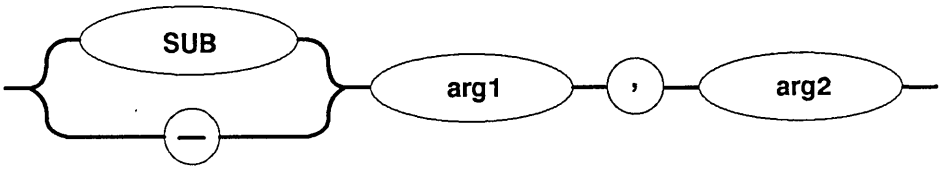
where:

arg1 and arg2 — Numbers the command multiplies together. The value of each argument cannot be greater than 2^{32}-1.

In response, the command displays the unsigned product of the two numbers in both hexadecimal and decimal format.

SUB

This command subtracts one number from another. Its format is



W-0983

where:

arg1 and arg2 — Numbers on which the command operates. The command subtracts arg2 from arg1. The value of each argument cannot be greater than 232-1.

In response, the command displays the unsigned difference in both hexadecimal and decimal format.

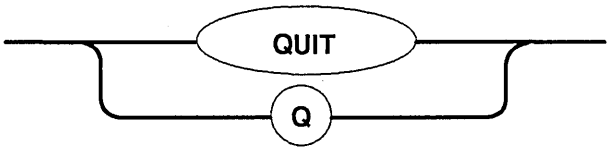
ERROR MESSAGES

argument error	A syntax error was made in the command, a nonnumeric value for one of the arguments was specified, or a value for a block number parameter that was not a valid block number was specified.
<blocknum>, block out of range	If the command was an ADDRESS command, the block number entered was greater than the number of blocks in the volume.
<address>, address not on the disk	If the command was a BLOCK command, BLOCK converted the address to a volume block number, but the block number was greater than the number of blocks in the volume.

EXAMPLES

```
*MUL 134T, 13T <CR>
 6CE ( 1742T)
*+ 8, 4  <CR>
 0C  (   12T)
*SUB 8884, 256 <CR>
 862E (34350T)
*MOD 1225, 256T <CR>
 25 (   37T)
*HEX 155T <CR>
 9B
*ADDRESS 15 <CR>
absolute address = 0A80
*BLOCK 2236 <CR>
block number = 44
```

This command exits the Disk Verification Utility and returns control to the Human Interface command level. The format of the QUIT command is as follows:



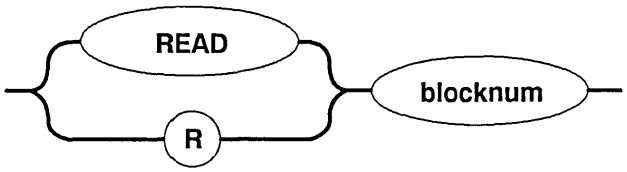
W-0984

This command is identical to the EXIT command.

NOTE

Although you can use DISKVERIFY to verify the system device (:SD:), note that all connections to this device are deleted by the operating system. After exiting, you must reboot the system or use the warm start feature (see the *iRMX® System Debugger Reference Manual*).

This command reads a volume block from the disk into the working buffer. The format of the READ command is



W-0985

INPUT PARAMETER

blocknum	Number of the volume block to read. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. If you omit this parameter, the READ command reads the most recently accessed block.

OUTPUT

In response to the command, READ reads the block into the working buffer and displays the following message:

```
read block number:  <blocknum>
```

where <blocknum> is the number of the block.

DESCRIPTION

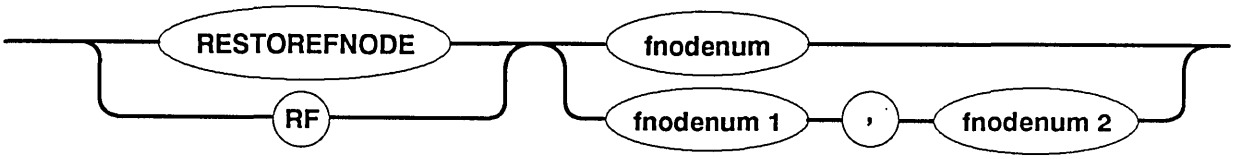
The READ command copies a specified volume block from the volume to the working buffer. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use DISPLAYBYTE and DISPLAYWORD to display the block, and you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back to the volume and repair damaged volume data.

ERROR MESSAGES

argument error	A nonnumeric character was specified in the blocknum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.

FFFFFFFF, block out of range	No block number was specified and no previous read request was executed on this volume.

This command copies an fnode or a range of fnodes from the R?SAVE file to the fnode file. Before changing the fnode file, RESTOREFNODE displays the fnode number to be changed and prompts you to confirm (by entering a "Y" or "y") that the fnode is to be restored. R?SAVE must have been reserved (the RESERVE option of the FORMAT command must have been specified) when the volume was formatted. The format of the RESTOREFNODE command is as follows:



W-0986

INPUT PARAMETER

fnodenum	The hexadecimal number of the fnode to be restored. This number must be greater than or equal to zero and less than the maximum number of fnodes defined when the volume was formatted.
fnodenum1	The initial hexadecimal fnode number in a range of fnodes to be restored. This number must be greater than or equal to zero and less than or equal to the final fnode number in the range (fnodenum2).
fnodenum2	The final hexadecimal fnode number in a range of fnodes to be restored. This number must be greater than or equal to the initial fnode number in the range (fnodenum1) and less than the maximum number of fnodes defined when the volume was formatted.

OUTPUT

When the fnode is restored (the response to the confirmation query is "Y" or "y"):

```
restore fnode    (fnodenum)?  Y <CR>
restored fnode number:    (fnodenum)
*
```

When the fnode is not restored (the response to the confirmation query is not "Y"):

```
restore fnode     (fnodenum)?   <CR>
*
```

DESCRIPTION

The RESTOREFNODE command enables you to rebuild a damaged fnode file, thereby re-establishing links to data that would otherwise be lost. RESTOREFNODE copies an fnode or a range of fnodes from the R?SAVE file (the fnode backup file) to the fnode file. Before each of the specified fnodes is copied, RESTOREFNODE displays a query prompting you to confirm that the indicated fnode is to be restored. You must reply to this query with the letter "Y" (either "Y" or "y") to restore the fnode. If you enter any other response, RESTOREFNODE will not restore the fnode and will pass on to the next fnode in the range.

Since RESTOREFNODE operates on the R?SAVE file, you must have reserved this file when the volume was formatted. (You reserve R?SAVE by specifying the RESERVE parameter when you invoke the FORMAT command to format the volume.) If the R?SAVE file was not reserved when the volume was formatted, RESTOREFNODE will return an error message.

CAUTION

When using this command, be sure that any fnode you restore represents a file that has not been modified since the last fnode backup. RESTOREFNODE overwrites the specified fnode in the fnode file with the corresponding fnode in the R?SAVE file. If that fnode has not been backed up since the last file modification, a valid fnode may be overwritten with invalid data. Thus, all links to the associated file will be destroyed, and YOU WILL LOSE ALL OF THE DATA IN THE FILE.

ERROR MESSAGES

argument error	When the command was entered, no argument was supplied. This command requires an argument.
no save area was reserved when volume was formatted	The volume was not formatted to support backup fnodes. This means the RESERVE option was not specified when the volume was formatted.

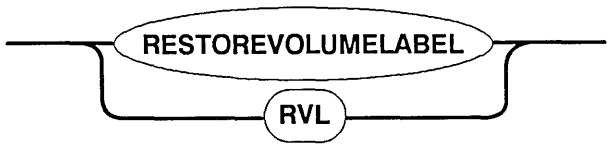
not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.
<fnode num>, fnode out of range	The fnode number specified is not in the range of 0 to (maximum fnodes - 1).
allocation bit not set for saved fnode restore fnode <fnode num>?	The fnode you specified has not been backed up in the R?SAVE file. If you respond to the query with a "Y" or "y", THE DATA IN THE FILE ASSOCIATED WITH THE ORIGINAL FNODE WILL BE LOST.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
*restorefnode 9,0B <CR>   or   rf 9,0B <CR>
restore fnode     9?  Y <CR>
restored fnode number:    9
restore fnode    0A?  Y <CR>
restored fnode number:   0A
restore fnode    0B?  Y <CR>
restored fnode number:   0B
*
```

RESTOREVOLUMELABEL

This command copies the duplicate volume label to the volume label on track 0. The duplicate volume label must have been constructed when the volume was formatted. (That is, the RESERVE option of the FORMAT command must have been specified when the volume was formatted.) The format of the RESTOREVOLUMELABEL command is as follows:



W-0987

INPUT PARAMETERS

None.

OUTPUT

Volume label restored.

DESCRIPTION

The RESTOREVOLUMELABEL command enables you to rebuild a damaged volume label, thereby re-establishing links to data that would otherwise be lost. RESTOREVOLUMELABEL copies the duplicate volume label to the volume label offset on track 0. When you use the Human Interface FORMAT command to create the duplicate volume label (by specifying the RESERVE parameter), the volume label is automatically copied to the end of the R?SAVE file. Because the contents of the volume label do not change, no other volume label backup is required.

If a duplicate volume label has been reserved on a volume, the Disk Verification Utility can access that volume as a Named volume even if the volume label is damaged. When the original volume label is corrupted, the Disk Verification Utility attempts to use the duplicate volume label. If the backup label is used, a "DUPLICATE VOLUME LABEL USED" message appears when the utility is invoked.

If the duplicate volume label was not reserved when the volume was formatted, RESTOREVOLUMELABEL will return an error message.

ERROR MESSAGES

argument error	When the command was entered, an argument was supplied. This command does not accept an argument.
no save area was reserved when volume was formatted	The volume has not been formatted to support volume label backup.
not a named disk	The volume specified when the Disk Verification Utility was invoked is a physical volume, not a named volume.

EXAMPLE

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
DUPLICATE VOLUME LABEL USED
*restorevolumelabel <CR>  or  rvl <CR>
volume label restored
*
```

This command writes the reconstructed free fnodes bit map, volume free space bit map, and the bad blocks bit map to the volume being verified. (The NAMED2 and PHYSICAL options of the VERIFY command originally created the maps.) The format of the SAVE command is

W-0988

OUTPUT

In response to this command, SAVE displays the following message:

```
save fnode map?
```

If you want to write the reconstructed free fnodes map to the volume, enter Y, y, or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the free fnodes map to the volume and displays the following message:

```
free fnode map saved
```

In any case, SAVE next displays the following message:

```
save space map?
```

If you want to write the reconstructed free space map to the volume, enter Y or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the volume free space map to the volume and displays the following message:

```
free space map saved
```

SAVE displays the following message if the bad blocks map is reconstructed:

```
save bad block map?
```

If you want to write the reconstructed bad blocks map to the volume, enter Y, y, or YES. Otherwise, enter any other character or a carriage return. If you enter YES, SAVE writes the volume bad blocks map to the volume and displays the following message:

```
bad block map saved
```

DESCRIPTION

Whenever you perform a VERIFY function with the NAMED2 option (refer to the description of the VERIFY command for more information), VERIFY creates its own free fnodes map and volume free space map. It does this by examining all directories and fnodes on the volume, not by copying the maps that exist on the volume. To create the free fnodes map, it examines every directory on the volume to determine which fnodes represent actual files. To create the volume free space map, it examines the POINTER(n) fields of the fnodes to determine which volume blocks the files use.

If the volume has a bad blocks file and you perform a VERIFY function with the PHYSICAL option (refer to the description of the VERIFY command for more information), VERIFY creates its own bad blocks map. It does this by examining every block on the volume, not by copying the maps that exist on the volume.

VERIFY then compares the newly created maps with the maps that exist on the volume. If a discrepancy exists, VERIFY displays a message indicating this.

The SAVE command takes the free fnodes map, the volume free space map, and the bad block map created during the VERIFY operation and writes them to the volume, replacing the maps that currently exist.

ERROR MESSAGE

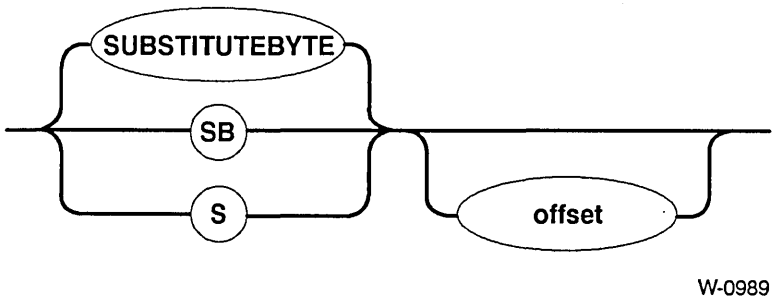
nothing to save	No bit map was constructed prior to invoking SAVE. (Bit maps are constructed by NAMED2 or PHYSICAL verifications.)

EXAMPLE

The following example illustrates the format of the SAVE command after you use VERIFY and the NAMED or NAMED2 option.

```
* VERIFY NAMED2 <CR>
'NAMED2' VERIFICATION
      .
      .
      .
   BIT MAPS O.K.
* SAVE <CR>
save fnode map?  y <CR>
   free fnode map saved
save space map?  y <CR>
   free space map saved
*
```

This command enables you to interactively change the contents of the working buffer (in byte format). You can abort this command by typing a CONTROL-C. The format of the SUBSTITUTEBYTE command is



INPUT PARAMETER

offset — Number of the first byte, relative to the start of the working buffer, that you want to change. This number can range from 0 to (block size - 1), where block size is the size of a volume block (and thus the size of the working buffer). If you omit this parameter, the command assumes a value of 0.

OUTPUT

In response to the command, SUBSTITUTEBYTE displays the specified byte and waits for you to enter a new value. This display appears as

```
<offset>: val -
```

where <offset> is the number of the byte, relative to the start of the buffer, and val is the current value of the byte. At this point, you can enter one of the following:

- A value followed by a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. If the value you enter requires more than one byte of storage, SUBSTITUTEBYTE uses only the low-order byte of the value. It then displays the next byte in the buffer and waits for further input.
- A carriage return alone. This causes SUBSTITUTEBYTE to leave the current value as is and display the next byte in the buffer. It then waits for further input.

- A value followed by a period (.) and a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. It then exits from the SUBSTITUTEBYTE command and gives the asterisk (*) prompt, enabling you to enter any DISKVERIFY command.
- A period (.) followed by a carriage return. This exits the SUBSTITUTEBYTE command and gives the asterisk (*) prompt, enabling you to enter any DISKVERIFY command.

DESCRIPTION

With the SUBSTITUTEBYTE command you can interactively change bytes in the working buffer. Once you enter the command, SUBSTITUTEBYTE displays the offset and the value of the first byte. You can change the byte by entering a new byte value, or you can leave the byte as is by entering a carriage return. The command then displays the next byte in the buffer. In this manner, you can consecutively step through the buffer, changing whatever bytes are appropriate. When you finish changing the buffer, you can enter a period followed by a carriage return to exit the command.

The SUBSTITUTEBYTE command considers the working buffer to be a circular buffer. That is, entering a carriage return when you are positioned at the last byte of the buffer causes SUBSTITUTEBYTE to display the first byte of the buffer.

The SUBSTITUTEBYTE command changes only the values in the working buffer. To make the changes in the volume, you must enter the WRITE command to write the working buffer back to the volume.

ERROR MESSAGES

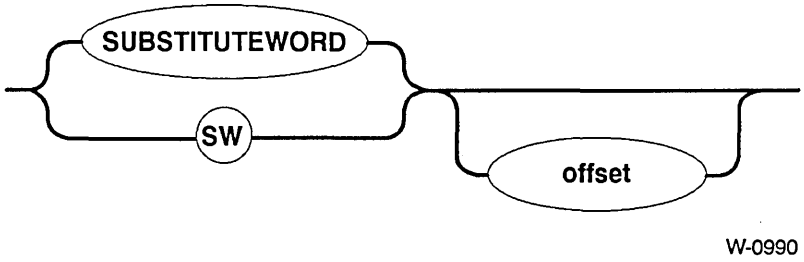
argument error	A nonnumeric character was specified in the offset parameter.
<offsetnum>, invalid offset	An offset value larger than the number of bytes in the block was specified.

EXAMPLE

This example changes several bytes in two portions of the working buffer. Two SUBSTITUTEBYTE commands are used.

```
* SUBSTITUTEBYTE<CR>

0000: A0 - 00<CR>
0001: 80 - <CR>
0002: E5 - <CR>
0003: FF - 31<CR>
0004: FF - .<CR>

* SUBSTITUTEBYTE 40<CR>

0040: 00 - E6<CR>
0041: 00 - E6.<CR>

*
```

This command is identical to SUBSTITUTEBYTE, except that it displays the buffer in WORD format, and substitutes word values in the buffer. The format of the SUBSTITUTEWORD command is

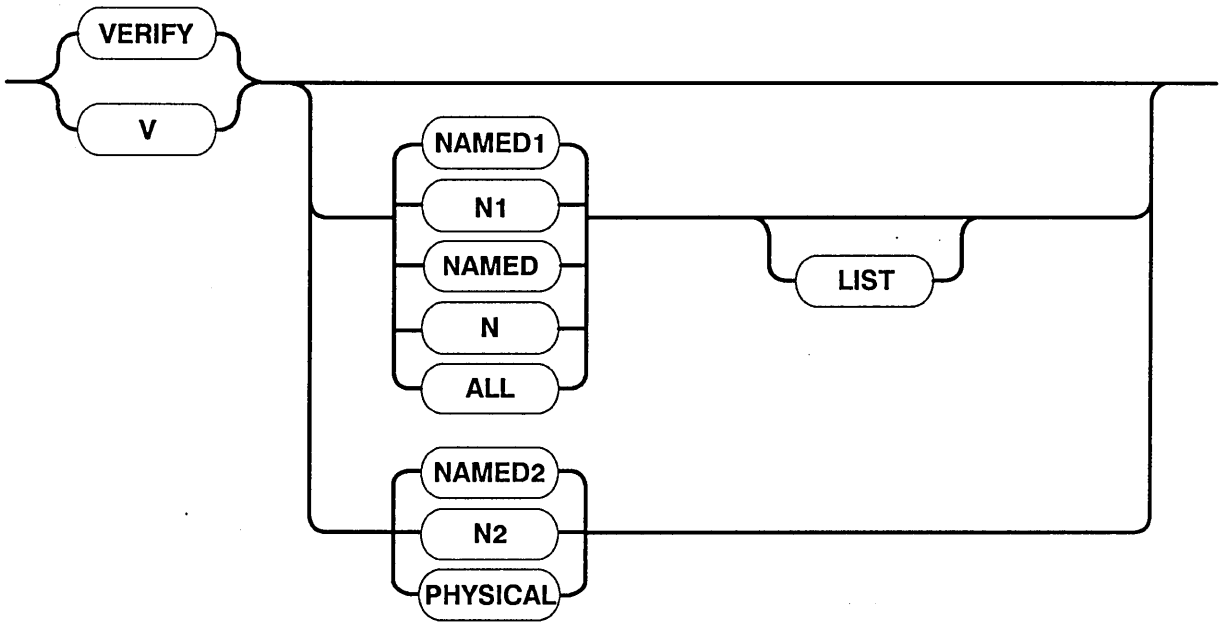


EXAMPLE

This example changes several bytes in two areas of the working buffer. Two SUBSTITUTEWORD commands are used. In the first command the words begin on even addresses, and in the second command, they begin on odd addresses.

```
*SUBSTITUTEWORD<CR>

0000: A0B0 - 0000<CR>
0002: 8070 - <CR>
0004: E511 - <CR>
0006: FFFF - 3111<CR>
0008: FFFF - .<CR>

*SUBSTITUTEWORD 35<CR>

0035: 0000 - E6FF<CR>
0037: 0000 - E6AB.<CR>

*
```

This command checks the structures on the volume to determine whether the volume is properly formatted. You can abort this command by typing a CONTROL-C. The format of the VERIFY command is



INPUT PARAMETERS

NAMED1 or N1 — Checks named volumes to ensure that the information recorded in the fnodes is consistent and matches the information obtained from the directories themselves. VERIFY performs the following operations during a NAMED1 verification:

- Checks fnode numbers in the directories to see if they correspond to allocated fnodes.
- Checks the parent fnode numbers recorded in the fnodes to see if they match the information recorded in the directories.
- Checks the fnodes against the files to determine if the fnodes specify the proper file type.
- Checks the POINTER(n) structures of long files to see if the indirect blocks accurately reflect the number of blocks used by the file.
- Checks each fnode to see if the TOTAL SIZE, TOTAL BLKS, and THIS SIZE fields are consistent.

- Checks the bad blocks file to see if the blocks in the file correspond to the blocks marked as "bad" on the volume.
- Checks the checksum of each fnode.

NAMED2 or N2

Checks named volumes to ensure that the information recorded in the free fnodes map and the volume free space map matches the actual files and fnodes. VERIFY performs the following operations during a NAMED2 verification:

- Creates a free fnodes map by examining every directory in the volume. It then compares that free fnodes map with the one already on the volume.
- Creates a free space map by examining the information in the fnodes. It then compares that free space map with the one already on the volume.
- Checks to see if the block numbers recorded in the fnodes and the indirect blocks actually exist.
- Checks to see if two or more files use the same volume block. If so, it lists the files referring to each block.
- Checks the volume free space map for any bad blocks that are marked as "free."
- Checks to see if two or more directories reference the same fnode. If so, it lists the directories referring to each fnode.

NAMED or N

Performs both the NAMED1 and NAMED2 operations on a named volume. If you specify the VERIFY command with no option, NAMED is the default.

PHYSICAL

Reads all blocks on the volume and checks for I/O errors. This parameter applies to both named and physical volumes. VERIFY also creates a bad blocks map by examining every block on the volume.

ALL

Performs all operations appropriate to the volume. For named volumes, this option performs both the NAMED and PHYSICAL operations. For physical volumes, this option performs only the PHYSICAL operations.

LIST

When you specify this option, the file information in Figure 2-3 is displayed for every file on the volume, even if the file contains no errors. You can use this option with all parameters that, either explicitly or implicitly, specify the NAMED1 parameter.

OUTPUT

VERIFY produces a different kind of output for each of the NAMED1, NAMED2, and PHYSICAL options. The NAMED and ALL options produce combinations of these three kinds of output.

Figure 2-3 illustrates the format of the NAMED1 output (without the LIST option).

```
DEVICE NAME = <devname>  : DEVICE SIZE = <devsize>  : BLOCK SIZE = <blksize>

'NAMED1' VERIFICATION

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>
   .          .          .          .          .          .
   .          .          .          .          .          .
   .          .          .          .          .          .
FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
    <error messages>
```

Figure 2-3. NAMED1 Verification Output

The following paragraphs identify the fields listed in Figure 2-3.

<devname>	Physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devsize>	Hexadecimal size of the volume, in bytes.
<blksize>	Hexadecimal volume granularity. This number is the size of a volume block.
<filename>	Name of the file (1 to 14 characters).
<fnodenum>	Hexadecimal number of the file's fnode.
<lev>	Hexadecimal level of the file in the file hierarchy. The root directory of the volume is the only level 0 file. Files contained in the root directory are level 1 files. Files contained in level 1 directories are level 2 files. This numbering continues for all levels of files in the volume.

<parnt>	Fnode number of the directory that contains this file, in hexadecimal.
<typ>	File type, either DATA (data files), DIR (directory files), SMAP (volume free space map), FMAP (free fnodes map), BMAP (bad blocks map), or VLAB (volume label file). If VERIFY cannot ascertain that the file is a directory or data file, it displays the characters "*****" in this field.
<error messages>	Messages that indicate the errors associated with the previously-listed file. The possible error messages are listed later in this section.

As Figure 2-3 shows, the NAMED1 option (without the LIST option) displays information about each file that is in error. If you used the LIST option with the NAMED1 option, the file information in Figure 2-3 is displayed for every file, even if the file contains no errors. The NAMED1 display also contains error messages that immediately follow the list of the affected files.

Figure 2-4 illustrates the format of the NAMED2 output. If VERIFY detects an error during NAMED2 verification, it displays one or more error messages in place of the "BIT MAPS O.K." message.

```
DEVICE NAME = <devname> : DEVICE SIZE = <devsize> : BLK SIZE = <blksze>

'NAMED2' VERIFICATION

   BIT MAPS O.K.
```

Figure 2-4. NAMED2 Verification Output

The fields in Figure 2-4 are exactly the same as the corresponding fields in Figure 2-3.

Figure 2-5 illustrates the format of the PHYSICAL output.

```
DEVICE NAME = <devname>    : DEVICE SIZE = <devsize> : BLOCK SIZE = <blksize>

'PHYSICAL' VERIFICATION

        NO ERRORS
```

Figure 2-5. PHYSICAL Verification Output

The fields in Figure 2-5 are exactly the same as the corresponding fields in Figure 2-3.

If VERIFY detects an error during PHYSICAL verification, it displays the message:

```
<blocknum>, error
```

in place of the "NO ERRORS" message.

If you specify NAMED verification, VERIFY displays both the NAMED1 and NAMED2 output. If you specify the ALL verification for a named volume, VERIFY displays the NAMED1, NAMED2, and PHYSICAL output. If you specify the ALL verification for a physical volume, VERIFY displays the PHYSICAL output.

DESCRIPTION

The VERIFY command checks physical and named volumes to ensure that the volumes contain valid file structures and data areas. VERIFY can perform three kinds of verification: NAMED1, NAMED2, and PHYSICAL. NAMED1 and NAMED2 verifications check the file structures of named volumes. They do not apply to physical volumes. A PHYSICAL verification checks each data block of the volume for I/O errors. PHYSICAL verification applies to both named and physical volumes.

As part of the NAMED2 verification, VERIFY creates a free fnodes map and a volume free space map, which it compares with the corresponding maps on the volume. You can use the SAVE command to write the maps produced during NAMED2 verification to the volume, overwriting the maps on the volume.

When you perform a PHYSICAL verification on a named volume, VERIFY also creates a bad blocks map. You can use the SAVE command to write the bad blocks map produced during PHYSICAL verification to the volume; this destroys the bad blocks map already on the volume.

ERROR MESSAGES

Four kinds of error messages can occur as a result of entering the VERIFY command: VERIFY command errors, NAMED1 errors, NAMED2 errors, and PHYSICAL errors.

VERIFY Command Error

argument error — The parameter specified is not a valid VERIFY parameter.

NAMED1 Messages

The following messages can appear in a NAMED1 display, immediately after the file to which they refer.

- <blocknum 1 - blocknum n>, block bad

  The block numbers displayed in this message are marked as "bad."

- <blocknum 1 - blocknum n >, invalid block number recorded in the fnode/indirect block

  One of the POINTER(n) fields in the fnode specifies block numbers larger than the largest block number in the volume.

- directory stack overflow

  This message indicates that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area. In this case, performing NAMED2 verification may indicate the cause of this problem.

- file size inconsistent
  total$size = <totsize> :this$size = <thsize> :data blocks = <blks>

  The TOTAL SIZE, THIS SIZE, and TOTAL BLKS fields of the fnode are inconsistent.

- <filetype>, illegal file type

  The file type of a user file, as recorded in the TYPE field of the fnode, is not valid. The valid file types and their descriptions are as follows:

File type	Number	Description
SMAP	1	volume free space map
FMAP	2	free fnodes map
BMAP	4	bad blocks map
DIR	6	directory
DATA	8	data
VLAB	9	volume label file

- <fnodenum>, allocation status bit in this fnode not set

  The file is listed in a directory but the flags field of its fnode indicates that fnode is free. The free fnodes map may or may not list the fnode as allocated.

- <fnodenum>, fnode out of range

  The fnode number is larger than the largest fnode number in the fnode file.

- <fnodenum>, parent fnode number does not match

  The file represented by fnodenum is contained within a directory whose fnode number does not match the parent field of the file.

- invalid blocknum recorded in the fnode/indirect block

  One of the pointers within the fnode or within the indirect block specifies a block number that is larger than the largest block number in the volume.

- insufficient memory to create directory stack

  There is not enough dynamic memory available in the system for the utility to perform the verification.

- sum of the blks in the indirect block does not match block in the fnode

  The file is a long file, and the number of blocks listed in a POINTER(n) field of the fnode does not agree with the number of blocks listed in the indirect block.

- total-blocks does not reflect the data-blocks correctly

  The TOTAL BLKS field of the fnode and the number of blocks recorded in the POINTER(n) fields are inconsistent.

- Bad Checksum, checksum is : <number>
  Checksum should be : <number>

  The checksum recorded in the fnode does not match the checksum calculated by DISKVERIFY.

NAMED2 Messages

The following messages can appear in a NAMED2 display.

- <blocknum1 - blocknum2>, bad block not allocated

  The volume free space map indicates that the blocks are free, but they are marked as "bad" in the bad blocks file.

- <blocknum>, block allocated but not referenced

  The volume free space map lists the specified volume block as allocated, but no fnode specifies the block as part of a file.

- <blocknum>, block referenced but not allocated

  An fnode indicates that the specified volume block is part of a file, but the volume free space map lists the block as free.

- directory stack overflow

  This message can indicate that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area. The "Multiple Reference" message (explained below) may help you find the cause of this problem.

- Fnodes map indicates fnodes > max$fnode

  The free fnodes map indicates that there are a greater number of unallocated fnodes than the maximum number of fnodes in the volume.

- <fnodenum>, fnode-map bit marked allocated but not referenced

  The free fnodes map lists the specified fnode as allocated, but no directory contains a file with the fnode number.

- <fnodenum>, fnode referenced but fnode-map bit marked free

  The specified fnode number is listed in a directory, but the free fnodes map lists the fnode as free.

- Free space map indicates Volume block > max$volume$block

  The free space map indicates that there are a greater number of unallocated blocks than the maximum number of blocks in the volume.

- insufficient memory to create directory stack

  Not enough dynamic memory is available in the system for the utility to perform the verification.

- insufficient memory to create fnode and space maps

  During a NAMED2 verification, the utility tried to create a free fnodes map and a volume free space map. However, not enough dynamic memory is available in the system to create these maps.

- insufficient memory to create bad blocks map

  During a PHYSICAL verification, the utility tried to create a bad blocks map. However, not enough dynamic memory is available in the system to create the map.

- Multiple reference to fnode <fnodenum>
  Path name : <full path name>
  referring fnodes:
  <fnodenum> Path name: <full path name>
  <fnodenum> Path name: <full path name>

  The directories on the volume list more than one file associated with this fnode number.

- Multiple reference to block <blocknum>
  referring fnodes:
  <fnodenum> Path name: <full path name>
  <fnodenum> Path name: <full path name>

  More than one fnode specifies this block as part of a file.

PHYSICAL Messages

- <blocknum>, error

  An I/O error occurred when VERIFY tried to access the specified volume block. The volume probably has a physical defect.

Miscellaneous Messages

The following messages indicate internal errors in the Disk Verification Utility. Under normal conditions these messages should never appear. However, if these messages (or other undocumented messages) do appear during a NAMED1 or NAMED2 verification, you should exit the Disk Verification Utility and re-enter the DISKVERIFY command.

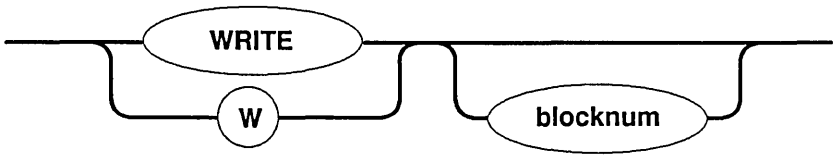
directory stack empty
directory stack error
directory stack underflow

EXAMPLE

The following command performs both named and physical verification on a named volume.

```
*VERIFY ALL <CR>

DEVICE NAME = F1            : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED1' VERIFICATION

'NAMED2' VERIFICATION
   BIT MAPS O.K.
'PHYSICAL' VERIFICATION
   NO ERRORS
*
```

This command writes the contents of the working buffer to the volume. The format of this command is



W-0992

INPUT PARAMETER

blocknum	Number of the volume block to which the command writes the working buffer. This number can range from 0 through maxblocks-1, where maxblocks is the maximum number of blocks in the volume. If you omit this parameter, WRITE writes the buffer back to the block most recently accessed.

OUTPUT

In response to the command, WRITE displays the following message:

```
write to block <blocknum>?
```

where <blocknum> is the number of the volume block to which WRITE intends to write the working buffer. If you respond by entering Y or any character string beginning with Y or y, WRITE copies the working buffer to the specified block on the volume and displays the following message:

```
written to block number:<blocknum>
```

Any other response aborts the write process.

DESCRIPTION

The WRITE command is used in conjunction with the READ, DISPLAYBYTE, DISPLAYWORD, SUBSTITUTEBYTE, and SUBSTITUTEWORD commands to modify information on the volume. Initially you use READ to copy a volume block from the volume to a working buffer. Then you can use DISPLAYBYTE and DISPLAYWORD to view the buffer and SUBSTITUTEBYTE and SUBSTITUTEWORD to change the buffer. Finally, you can use WRITE to write the modified buffer back to the volume. By default, WRITE copies the buffer to the block most recently accessed by a READ or WRITE command.

A WRITE command does not destroy the data in the working buffer. The data remains the same until the next SUBSTITUTEBYTE, SUBSTITUTEWORD, or READ command modifies the buffer.

ERROR MESSAGES

argument error	A syntax error was made or nonnumeric characters were specified in the blocknum parameter.
<blocknum>, block out of range	The block number specified was larger than the largest block number in the volume.
FFFFFFFF, block out of range	No blocknum was specified and no previous read request was executed on this volume.

EXAMPLE

The following command copies the working buffer to the block from which it was read.

```
*WRITE  <CR>
write 4B?  y <CR>
write to block 4B? y
written to block number: 4B
*
```

BACKING UP AND RESTORING FNODES 3

3.1 INTRODUCTION

To access data on a named volume (such as a disk), the iRMX I and iRMX II Operating Systems use a mechanism common to virtually all operating systems: it maintains an index to every file on the disk. This index is created when the disk is formatted and remains as a permanent structure at a dedicated location on the disk. The index consists of a system of pointers that indicate the location of the data files on the disk. Thus, when data must be stored on or retrieved from the disk, the operating system can find the exact location of the appropriate file by looking up the file name in the index.

In the operating system, the index consists of the iRMX volume label and an fnode file. This volume label resides at the same location in all devices and serves as the initial entry point into the device. The fnode file can reside anywhere on the disk (specified when the disk is formatted) and contains a series of individual structures called file descriptor nodes or "fnodes." There is one fnode for each file on the disk. The fnode contains information essential to accessing and maintaining the respective file.

The iRMX file structure for a named volume is organized as a hierarchical tree. That is, there is a root directory with branches to other directories and ultimately, to files. The organization of the fnode file reflects this hierarchical structure. The iRMX volume label contains a pointer to the fnode of the file structure's root directory. The root directory is always the starting address for any file or directory on the volume. It lists all the first level files and directories on the volume. First level directories point to second level files and directories, and so on, down the hierarchical structure.

As previously mentioned, each file or directory is represented by an fnode. The fnode, along with other data describing the file or directory, contains pointers to blocks on the volume. If the fnode describes a short file, these blocks contain the actual file data. If the fnode describes a long file, these blocks contain pointers to other blocks containing the actual data. (For a description of short and long files, see Appendix A.) If the fnode describes a directory, these blocks contain entries which describe the contents of the directory. Each entry lists the fnode number and name of the associated file or directory.

The operating system creates the iRMX volume label and the fnode file when the disk is formatted.

The number of unallocated fnodes in the fnode file is controlled by the FILES parameter of the FORMAT command. In addition to the unallocated fnodes, seven (with an option of nine) allocated fnodes are established when the fnode file is created. These allocated fnodes represent

- the fnode file
- the volume label file - R?VOLUMELABEL
- the volume free space map file - R?SPACEMAP
- the free fnodes map file - R?FNODEMAP
- the bad blocks file - R?BADBLOCKMAP
- the root directory
- the space accounting file,
- optionally, the duplicate volume label file - R?SAVE
- optionally, the MULTIBUS II second stage bootloader - R?SECONDSTAGE

For a full description of these files, see Appendix A, "Structure of A Named Volume."

Thereafter, when files or directories are created directly subordinate to the root, the operating system must adjust a pointer in the root fnode to indicate the fnode number of the new data file or directory file. Subsequently, directories subordinate to the root must also have their pointers adjusted when they become parents to a new data file or directory.

This method of storing and retrieving data on a disk has one major drawback. All access to files on the disk is through the iRMX volume label and the fnode file. If either the volume label file or the fnode file is damaged or destroyed, there is no practical way to recover data on the disk.

The backup and restore fnodes feature enables some recovery of data lost as a result of damage to the fnode file or the iRMX volume label. With this feature, you create a backup version of the volume label and all the fnodes on the disk. The backup version is stored in one of the innermost tracks of the disk where the chance of accidental loss of data is minimal. (In normal use, the disk heads do not extend to the innermost tracks.)

To implement this feature, the Human Interface FORMAT command has an optional parameter -- RESERVE. The FORMAT command creates a file named R?SAVE in the innermost track of the volume. A copy of the iRMX volume label is placed in the front (that is, the physical end) of the file and an fnode is allocated for R?SAVE in the fnode file. (The fnode for the R?SAVE file is allocated out of the fnodes reserved through the FILES parameter of the FORMAT command. Thus, if you specify "FILES = 3000" when you format, only 2999 of those fnodes will remain available after the R?SAVE fnode has been allocated.) Finally, FORMAT copies the fnode file into R?SAVE.

Notice that the FORMAT command creates a backup of the fnode file in its initialized state. R?SAVE is not subsequently updated as files are written to or deleted from the volume. Therefore, you will have to use the BACKUPFNODES Disk Verification Utility command or the BACKUP option of the Human Interface SHUTDOWN command to back up the fnode file at regular intervals. If the volume label or the fnode file become damaged, you can attempt to recover files on the volume by using the Disk Verification Utility commands (RESTOREFNODE and RESTOREVOLUMELABEL) to rebuild the index. To assist in this process, the DISPLAYSAVEFNODE Disk Verification Utility command enables you to look at individual fnodes stored in the R?SAVE file.

Since the contents of the iRMX volume label do not change, the copy of the volume label in R?SAVE is always valid. Therefore, you can restore the volume label at any time regardless of when the R?SAVE file was last updated. (When the Disk Verification Utility encounters a damaged volume label, it automatically uses the backup volume label if the R?SAVE file is present, however, it does not restore unless explicitly instructed.)

CAUTION

One note of caution: The fnode file is changed each time a volume is modified (that is, each time a file or directory is created, written to, or deleted from the volume). Therefore, valid restoration can be assured only for fnodes whose associated files or directories have not been changed since the last backup.

If the fnodes are not backed up after each modification, the structure of the R?SAVE file will differ from that of the fnode file. Some fnodes in R?SAVE may not be associated with the same files as the corresponding fnodes in the fnode file. Attempting to recover fnodes under these conditions is dangerous because the RESTOREFNODE command will overwrite what may be a valid fnode in the fnode file.

While the backup and restore fnodes feature is a useful aid in attempting to recover data on a volume, this capability is limited in scope. If you are troubleshooting your system, you may want to back up the fnodes on the system disk before taking any action that may risk the disk's integrity. You may also decide to back up the fnodes on a routine basis (before or during each system shutdown, for instance) so that the R?SAVE file is always relatively current. However, under normal circumstances, where a volume is accessed and modified frequently, backing up the fnodes after each modification is not practical. The most practical solution is to back up the fnode file once a day using the BACKUP option of the SHUTDOWN command.

Note that this feature is not intended to provide comprehensive protection from the loss of data associated with damaged iRMX volume labels or fnode files. Rather, it offers a tool that, when properly applied, can be useful in maintaining volume integrity in certain situations. For comprehensive protection against loss of data use the Human Interface BACKUP command.

3.2 USING FNODE BACKUP AND RESTORE

To use the fnode backup and restore feature, you must use the Human Interface FORMAT command and the Disk Verification Utility. Used together, these enable you to

- format a volume to create the backup file (R?SAVE)
- back up the fnodes of any files written to the volume
- examine the contents of the backup file (R?SAVE)
- restore damaged fnodes
- restore the volume label
- edit fnodes or save fnodes

This section describes how to perform each of these operations. A brief overview of the operation is followed by one or more examples of a typical implementation. In the examples, blue or bolded text indicates an entry you make from your terminal. Standard type (this is standard type) indicates system output to your terminal.

3.2.1 Creating the R?SAVE Fnode Backup File

If you intend to backup the volume label and the fnodes on a volume, you must first create the R?SAVE backup file on the innermost tracks of the volume. To do so, you must invoke the Human Interface FORMAT command, specifying the RESERVE option. NOTE THAT THE FORMAT COMMAND OVERWRITES ALL OF THE DATA CURRENTLY ON THE DISK. Therefore, make a backup copy of any files you wish to save using the Human Interface BACKUP command before invoking FORMAT.

Once the volume has been formatted, the R?SAVE file will contain a copy of the fnode file including the allocated fnodes (R?SPACEMAP, R?FNODEMAP, etc.). Therefore, you need not back up the fnode file immediately after formatting the volume.

PROCEDURE

From the Human Interface, invoke the FORMAT command, specifying the RESERVE parameter.

EXAMPLE

Assume that you have booted your system from a flexible diskette to format the system disk. The command listed below formats the disk and creates the R?SAVE backup file. The initialized fnode file is copied into R?SAVE.

```
_attachdevice cmbo as :mydisk: <CR>
_format :mydisk: il = 4 files = 3000 reserve <CR>

volume ( ) will be formatted as a NAMED volume
    granularity        =  1,024 map start   =   7,859
    interleave         =     4
    files              =  3000
    extensionsize      =     3
    save area reserved =    yes
    bad track/sector information written = no
    MSA bootstrap information written = no      <-- Appears in iRMX II
    volume size        = 15,984K                    only

TTTTTTTTTTTTTTTT
volume formatted
```

NOTE

The "map start" value may change if R?SAVE (and possibly R?SECONDSTAGE for iRMX II) is present.

The disk has now been formatted. A file named R?SAVE has been reserved in the innermost tracks of the disk. (If you use the Disk Verification Utility DISPLAYDIRECTORY command on the volume root fnode (fnode 6) or the Human Interface DIR command with the invisible (I) option on the volume root directory, you will find an fnode listed for R?SAVE.) R?SAVE contains a duplicate copy of the fnodes in the fnode file. That is, R?SAVE contains eight allocated fnodes (R?SAVE, R?SPACEMAP, R?FNODEMAP, etc.) and 2,999 unallocated fnodes. (Remember, the R?SAVE fnode is allocated out of the 3,000 fnodes specified through the FILES parameter.)

3.2.2 Backing up Fnodes on a Volume

DESCRIPTION

To back up the fnodes on a volume, you must have previously reserved the back up file R?SAVE when the volume was formatted. Thereafter, any modification to the volume (creating, writing to, or deleting a file) requires that the fnodes be backed up if the R?SAVE file is to contain an accurate copy of the fnode file.

You can backup the fnode on a volume either by:

- Using the Human Interface SHUTDOWN command with the BACKUP option
- Using the BACKUPFNODES option of DISKVERIFY (see Chapter 2)

EXAMPLE 1

This example shows how to backup the fnode file using SHUTDOWN with the BACKUP option. The BACKUP option allows you to copy the volume fnode file to its duplicate file, R?SAVE, on any attached volume.

```
super- SHUTDOWN  B  <CR>
***SYSTEM WILL BE SHUTDOWN IN 10 MINUTE(S)
:SD:, outstanding connections to device have been deleted
***SHUTDOWN COMPLETED  ***
```

R?SAVE now contains a duplicate copy of all fnodes in the fnode file.

EXAMPLE 2

This example shows how to use the BACKUPFNODE command of DISKVERIFY to backup the fnode file. Assume that the system disk is attached as logical device :SD:. The initial contents of the :SD: fnode file were copied to R?SAVE by the FORMAT command. A file has just been written to the volume. An fnode for this file is entered in the fnode file; however, no corresponding entry has been made in R?SAVE. The following sequence of commands will copy all fnodes in the fnode file into the R?SAVE file.

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
*backupfnodes <CR>          bf <CR>
fnode file backed up to save area
*
```

R?SAVE now contains a duplicate copy of all fnodes (allocated and unallocated) in the fnode file.

Note that in both cases you must reboot the system after backing up the fnodes on the volume.

3.2.3 Backing up the Volume Label

The volume label is initially copied to R?SAVE when the volume is formatted. Since the contents of the volume label do not change, no other volume label backup procedures are required.

3.2.4 Restoring Fnodes

DESCRIPTION

To restore fnodes on a volume, you must have previously reserved the backup file R?SAVE when the volume was formatted. If damage has occurred to the fnode file, you can attempt to rebuild the file (or portions of it) by using the Disk Verification Utility RESTOREFNODE command.

RESTOREFNODE enables you to restore a single fnode or a range of fnodes. You designate the fnodes to be restored by entering the fnode numbers. The specified fnodes in R?SAVE are copied into the corresponding fnodes in the fnode file.

Before restoring each fnode, RESTOREFNODE prompts you with the message "restore fnode <fnode number>?". To restore the fnode, you must enter "yes" or the letter "Y" (either Y or y). If you enter any other response, the fnode will not be restored.

When restoring fnodes, you must be very careful to ensure that you are not overwriting a valid fnode in the fnode file with an invalid fnode from R?SAVE. Be sure the volume has not been modified since the fnodes were last backed up.

PROCEDURE

1. Invoke the Disk Verification Utility, using the logical device name of the volume to be backed up.
2. When you receive the Disk Verification Utility prompt (*), enter the appropriate Disk Verification Utility commands (VERIFY, DISPLAYFNODE, etc.) to examine the fnodes file and determine which fnode must be restored.
3. Invoke the Disk Verification Utility RESTOREFNODE command to replace the damaged fnodes. The Disk Verification Utility prompts you to confirm that the proper fnode is being restored. Make sure you have specified the correct hexadecimal number for the fnode, then enter the letter "Y" in response to the prompt.
4. RESTOREFNODE returns the message "restored fnode < fnode number >" after the fnode in the R?SAVE file has been written over the corresponding fnode in the fnode file.

EXAMPLE 1

Assume that a disk drive is attached as logical device :SD:. The volume :SD: contains the R?SAVE fnode backup file. You have not modified the disk since the fnodes were last backed up. You suspect the fnode file has been damaged, so you use the Disk Verification Utility to confirm your suspicions:

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
* verify
    .
    .
    .
```

After using the Disk Verification Utility to examine the structure of the disk, you find that fnodes 09H through 0CH have probably been destroyed. You then use the RESTOREFNODE command to recover these fnodes.

```
* restorefnode 9, 0C <CR>    or    rf 9, 0C <CR>
restore fnode      9?   Y <CR>
restored fnode number:    9
restore fnode     0A?   Y <CR>
restored fnode number:   0A
restore fnode     0B?   Y <CR>
restored fnode number:   0B
restore fnode     0C?   Y <CR>
restored fnode number:   0C
```

Fnodes 09H through 0CH in the R?SAVE file have been copied into fnode 09H through 0CH in the fnode file. Since the disk has not been modified since the last fnode backup, restoring the damaged fnodes should now enable you to recover the data on the disk.

EXAMPLE 2

Assume the same initial conditions as Example 1 with the following exception: two files have been modified since the last time the fnodes were backed up. In the fnode file, the new files are represented by fnodes 0DH and 0EH. Again, you suspect that the fnode file has been damaged, so you use the Disk Verification Utility to check the condition of data on the disk:

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
*verify
    .
    .
    .
```

After using the Disk Verification Utility to examine the structure of the disk, you find that fnodes 9 through 10 have probably been destroyed. You decide to use the RESTOREFNODE command to recover these fnodes. You do not wish to restore fnodes 0DH and 0EH because they were not backed up. Since the data fields of fnodes 0DH and 0EH in R?SAVE contain all zeros, you would be destroying possibly useful data in the corresponding fnodes. You then use RESTOREFNODE to restore a range of fnodes that includes 0DH and 0EH. However, you intend to pass over the restoration of these two fnodes by responding to the confirmation prompt with some character other than "Y."

```
*restorefnode 9,10 <CR>    or    rf 9,10 <CR>
restore fnode     9?  Y <CR>
restored fnode number:    9
restore fnode    0A?  Y <CR>
restored fnode number:   0A
restore fnode    0B?  Y <CR>
restored fnode number:   0B
restore fnode    0C?  Y <CR>
restored fnode number:   0C
allocation bit not set for saved fnode
restore fnode    0D?   <CR>
allocation bit not set for saved fnode
restore fnode    0E?  n <CR>
restore fnode    0F?  Y <CR>
restored fnode number:   0F
restore fnode    10?  Y <CR>
restored fnode number:   10
```

Notice that because fnodes 0DH and 0EH were not allocated when they were backed up, those fnodes in R?SAVE are unallocated. Therefore, the Disk Verification Utility returns the "allocation bit not set for saved fnode" message. Since you do not wish to restore this fnode, you respond to the confirmation prompt with a character other than "Y."

The R?SAVE fnodes 09H through 0CH and fnodes 0FH through 10H have been copied over the corresponding fnodes in the fnode file. Fnodes 0D and 0E were not restored.

3.2.5 Restoring the Volume Label

DESCRIPTION

To restore the volume label, you must have previously reserved the backup file R?SAVE when you formatted the volume. If the volume contains the R?SAVE file, a backup copy of the volume label already exists. The FORMAT command automatically places a copy of the volume label into R?SAVE when the file is created. Thereafter, the contents of the volume label do not change and you can restore the label without fear of destroying data in the existing label.

To restore the volume label, you must invoke the Disk Verification Utility using the logical device name of the appropriate volume. If the volume label is corrupted, the Disk Verification Utility attempts to use the backup copy of the volume label in R?SAVE. When the backup label is used, the Disk Verification Utility issues a message that reads "duplicate volume label used." If this message appears when the Disk Verification Utility is activated, then the volume label is damaged. To restore the volume label, enter the Disk Verification Utility RESTOREVOLUMELABEL command. The current volume label will be overwritten with the volume label copy from R?SAVE.

PROCEDURE

1. Invoke the Disk Verification Utility, using the logical device name of the volume to be backed up.
2. If the "duplicate volume label used" message appears, the volume label must be restored. Enter the Disk Verification Utility RESTOREVOLUMELABEL command.
3. When the volume label has been restored, the Disk Verification Utility returns the message "volume label restored."

EXAMPLE

Assume that a disk drive is attached as logical device :SD:. The volume :SD: contains the R?SAVE fnode backup file. When you attempt to access files on :SD:, the system returns an E$ILLEGAL_VOLUME message. You suspect that the volume label may be damaged. You decide to check your suspicions using the Disk Verification Utility.

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
duplicate volume label used
*
```

The "duplicate volume label used" message confirms that the volume label has been damaged. You restore the volume label using the RESTOREVOLUMELABEL command.

```
*restorevolumelabel <CR>    or    rvl <CR>
volume label restored
*
```

The original volume label has been overwritten with the duplicate copy from the R?SAVE file. Attempts to access files on volume :SD: should now be successful.

3.2.6 Displaying R?SAVE Fnodes

DESCRIPTION

Any fnode (both allocated and unallocated) in the R?SAVE file can be examined by using the Disk Verification Utility DISPLAYSAVEFNODE command. The Disk Verification Utility will display vital information about the fnode (total blocks, total size, block pointers, parent node, etc.). The fnode is displayed in the same format used by the DISPLAYFNODE command.

To display an R?SAVE fnode, enter the DISPLAYSAVEFNODE command and specify the hexadecimal number of the fnode to be displayed.

PROCEDURE

1. Invoke the Disk Verification Utility using the logical device name of the appropriate volume.
2. When you receive the Disk Verification Utility prompt (*), enter the Disk Verification Utility DISPLAYSAVEFNODE command. Specify the hexadecimal number of the fnode to be displayed.
3. The Disk Verification Utility will return with an fnode display.

EXAMPLE

Assume that you cannot access a file on a disk attached as :SD:. You suspect that the fnode file may be damaged. By entering the Disk Verification Utility and displaying the file's directory, you find that the file you were unable to access is represented by fnode 3C8H. You use DISPLAYFNODE to display fnode 3C8H, but you are not confident of the data you see. Since the fnode for the file has been backed up since the file was last modified, you decide to use data in the R?SAVE fnode to examine the fnode file. The following command displays the data for fnode 3C8H in R?SAVE.

```
super- diskverify :sd: <CR>
iRMX <version> Disk Verify Utility, Vx.x
Copyright <year> Intel Corporation
All Rights Reserved
:sd:, outstanding connections to device have been deleted
        .
        .
        .

* displaysavefnode 3C8 <CR>    or    dsf 3C8 <CR>


    Fnode number = 3C8 (saved)
    path name: /USER/MYFILE
                      flags : 0025  =>  short file
                       type : 08  =>  data file
          file gran/vol gran : 01
                      owner : 0001
     create,access,mod times : 00000000, 00000000, 00000000
     total size,total blocks : 00002D01, 0000000C
           block pointer (1) : 000C, 004910
           block pointer (2) : 0000, 000000
           block pointer (3) : 0000, 000000
           block pointer (4) : 0000, 000000
           block pointer (5) : 0000, 000000
           block pointer (6) : 0000, 000000
           block pointer (7) : 0000, 000000
           block pointer (8) : 0000, 000000
                   this size : 00003000
                    id count : 0001
                accessor (1) : 0F, 0001
                accessor (2) : 00, 0000
                accessor (3) : 00, 0000
            parent, checksum : 03C4, 56CA
                     aux (*) : 000000
*
```

You can modify the contents of the both the original fnode file and the saved fnode file by using either the EDITFNODE or EDITSAVEFNODE commands.

STRUCTURE OF A NAMED VOLUME A

A.1 INTRODUCTION

This appendix describes the structure of an iRMX volume that contains named files. It is provided as reference information to help you interpret output from the DISKVERIFY commands or to help you create your own formatting utility programs.

This appendix is for programmers with experience in reading and writing actual volume information. It does not attempt to teach these functions.

A.2 VOLUME STRUCTURE

This appendix discusses the structure of named file volumes in detail. It covers the structure of directory files and the concepts of long and short files. It also includes information on:

- ISO Volume Label
- iRMX Volume Label
- MSA Bootloader Location Table (iRMX II only)
- fnode file
- volume free space map file
- free fnodes map file
- bad blocks map file
- root directory

The blocks reserved for the Bootstrap Loader (Figure A-1) are not discussed. Bootstrap Loader blocks are automatically included on a new volume when you format a volume with the FORMAT command. Refer to the FORMAT command for a description of the bootstrap option.

Figure A-1 illustrates the general structure of a named file volume.

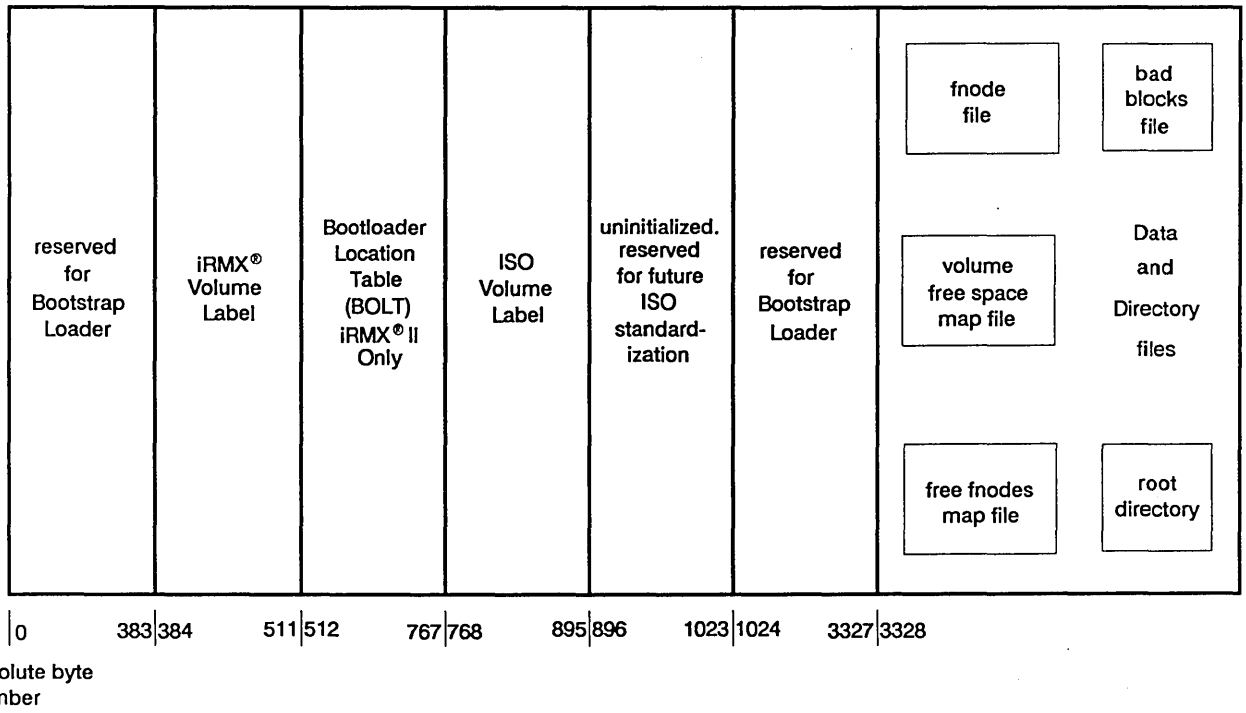


Figure A-1. General Structure of Named Volumes

A.3 VOLUME LABELS

Each iRMX named volume contains ISO (International Standardization Organization) label information as well as iRMX label information and files. This section describes the structure of ISO volume labels and iRMX volume labels, both of which must be present on a named volume.

A.3.1 ISO Volume Label

The ISO volume label is recorded in absolute byte positions 768 through 895 of the volume (for example, sector 07 of a single-density flexible diskette). The structure of this volume label (in PL/M notation) is:

```
DECLARE
    ISO$VOL$LABEL   STRUCTURE(
                        LABEL$ID(3)         BYTE,
                        RESERVED$A          BYTE,
                        VOL$NAME(6)         BYTE,
                        VOL$STRUC           BYTE,
                        RESERVED$B(60)      BYTE,
                        REC$SIDE            BYTE,
                        RESERVED$C(4)       BYTE,
                        ILEAVE(2)           BYTE,
                        RESERVED$D          BYTE,
                        ISO$VERSION         BYTE,
                        RESERVED$E(48)      BYTE);
```

where:

LABEL$ID(3)	Label identifier. For named file volumes, this field contains the ASCII characters "VOL".
RESERVED$A	Reserved field containing the ASCII character "1".
VOL$NAME(6)	Volume name. This field can contain up to six printable ASCII characters, left justified and space filled. A value of all spaces implies that the volume name is recorded in the iRMX Volume Label (absolute byte positions 384-393).
VOL$STRUC	For named file volumes, this field contains the ASCII character "N", indicating that this volume has a non-ISO file structure.
RESERVED$B(60)	Reserved field containing 60 bytes of ASCII spaces.
REC$SIDE	For named file volumes, this field contains the ASCII character "1" to indicate that only one side of the volume is to be recorded.
RESERVED$C(4)	Reserved field containing four bytes of ASCII spaces.

ILEAVE(2)	Two ASCII digits indicating the interleave factor for the volume, in decimal. ASCII digits consist of the numbers 0 through 9. When formatting named volumes, you should set this field to the same interleave factor that you use when physically formatting the volume.
RESERVED$D	Reserved field containing an ASCII space.
ISO$VERSION	For named file volumes, this field contains the ASCII character "1", which indicates ISO version number one.
RESERVED$E(48)	Reserved field containing 48 ASCII spaces.

A.3.2 iRMX® Volume Label

The iRMX Volume Label is recorded in absolute byte positions 384 through 511 of the volume (sector 04 of a single density flexible diskette). The structure of this volume label is as follows:

```
DECLARE
    RMX$VOLUME$INFORMATION  STRUCTURE(
                                VOL$NAME(10)        BYTE,
                                FLAGS               BYTE,
                                FILE$DRIVER         BYTE,
                                VOL$GRAN            WORD,
                                VOL$SIZE            DWORD,
                                MAX$FNODE           WORD,
                                FNODE$START         DWORD,
                                FNODE$SIZE          WORD,
                                ROOT$FNODE          WORD,
                                DEV$GRAN            WORD,
                                INTERLEAVE          WORD,
                                TRACK$SKEW          WORD,
                                SYSTEM$ID           WORD,
                                SYSTEM$NAME(12)     BYTE,
                                DEVICE$SPECIAL(8)   BYTE,
                                VOL$FLAGS           BYTE);
```

iRMX I Note: The iRMX I Volume Label does not contain a VOL$FLAGS field. All other fields in the structure shown above are the same in the iRMX I and iRMX II Operating Systems.

where:

VOL$NAME(10) Volume name in printable ASCII characters, left justified and zero filled.

FLAGS BYTE that lists the device characteristics for automatic device recognition. The individual bits in this BYTE indicate the following characteristics (bit 0 is rightmost bit):

Bit	Meaning
0	VF$AUTO flag. When set to one, this bit indicates that the FLAGS byte contains valid data for automatic device recognition. When set to zero, it indicates that the remaining flags contain meaningless data.
1	VF$DENSITY flag. This bit indicates the recording density of the volume. When set to one, it indicates modified frequency modulation (MFM) or double-density recording. When set to zero, it indicates frequency modulation (FM) or single-density recording.
2	VF$SIDES flag. This bit indicates the number of recording sides on the volume. When set to one, it indicates a double-sided volume. When set to zero, it indicates a single-sided volume.
3	VF$MINI flag. This bit indicates the size of the recording media. When set to one, it indicates a 5 1/4-inch volume. When set to zero, it indicates an 8-inch volume.
4	VF$FORMAT flag. This bit indicates the type of format on track 0. When set to one, it indicates that all tracks, including track 0, have the same format (Uniform format). When set to zero, it indicates track 0 is formatted to be single density with 128-byte sectors (Standard format).
5-7	Reserved

FILE$DRIVER Number of the file driver used with this volume. For named file volumes, this field is set to four.

VOL$GRAN Volume granularity, specified in bytes. This value must be a multiple of the device granularity. It sets the size of a logical device block, also called a volume block.

VOL$SIZE Size of the entire volume, in bytes.

MAX$FNODE — Number of fnodes in the fnode file. (Refer to the next section for a description of fnodes.)

FNODE$START — A 32-bit value that represents the number of the first byte in the fnode file (byte 0 is the first byte of the volume).

FNODE$SIZE — Size of an fnode, in bytes.

ROOT$FNODE — Number of the fnode describing the root directory. (Refer to the next section for further information.)

DEV$GRAN — Device granularity of all tracks except track zero (which contains the volume label). This field is important only when the system requires automatic device recognition.

INTERLEAVE — Block interleave factor for this volume. This value indicates the physical distance, in blocks, between consecutively-numbered blocks on the volume. A value of one indicates that consecutively-numbered blocks are adjacent. A value of zero indicates an unknown or undefined interleave factor.

TRACK$SKEW — Offset, in bytes, between the first block on one track and the first block on the next track. A value of zero indicates that all tracks are identical.

SYSTEM$ID — Numerical code identifying the operating system that formatted the volume. The following codes are reserved for Intel operating systems:

Operating System	Code
iRMX I, II	0 - 0Fh
iRMX 88	10h - 1Fh
iNDX	20h - 2Fh

Currently, the iRMX I and iRMX II Operating Systems place a zero in this field.

SYSTEM$-NAME(12) — Name of the operating system that formatted the volume, in printable ASCII characters, left justified and space filled. Zeros (ASCII nulls) indicate that the operating system is unknown. The iRMX I and iRMX II Operating Systems currently place several pieces of information into this field, as follows:

- The leftmost eight bytes of this field contain the ASCII characters "iRMX I " or "iRMX 286" to identify the operating system. The iRMX 286, Release 1, Operating System and versions of iRMX 86 before Release 7 filled this field with zeros.

- The next byte is an ASCII character that identifies the program that formatted the volume. The following characters apply:

Character	Formatting Program
F	Human Interface FORMAT command
U	iRMX 86 Files Utility (used prior to iRMX 86, Release 7)

  If the formatting program is unable to provide this information, it places an ASCII space in this field.

- The Human Interface FORMAT command that is part of iRMX I.8 and iRMX II.4 places the characters "03 " in the last 3 bytes of this field. The iRMX II.3 FORMAT command places "02 " in the last three bytes.

DEVICE$-SPECIAL(8)

Reserved for special device-specific information. When no device -specific information exists, this field must contain zeros. For example, if the device is a Winchester disk with an iSBC 214/215G controller, the iRMX I and iRMX II Operating Systems imposes a structure on this field and supplies the following information:

```
SPECIAL STRUCTURE(
            CYLINDERS     WORD,
            FIXED         BYTE,
            REMOVABLE     BYTE,
            SECTORS       BYTE,
            SECTOR$SIZE   WORD,
            ALTERNATES    BYTE);
```

where:

CYLINDERS — Total number of cylinders on the disk drive.

FIXED	Number of heads on the fixed disk or Winchester disk.
REMOVABLE	Number of heads on the removable disk cartridge.
SECTORS	Number of sectors in a track.
SECTOR SIZE	Sector size, in bytes.
ALTERNATES	Number of alternate cylinders or spare sectors on a track.

iRMX I Note:	The iRMX I Volume Label does not contain a VOL$FLAGS field.

VOL$FLAGS — Contains flags for general volume information. The following flag is defined:

Flag	Bit	Meaning
VF$INTEGRITY	0	0 = The volume has been properly shut down.
		1 = Indicates possible disk corruption (the volume was attached, but was not subsequently detached).

The remainder of the Volume Label (bytes 441 through 511) is reserved and must be set to zero.

A.3.3 Bootloader Location Table

iRMX I Note:	The iRMX I Operating System does not support the Bootloader Location Table.

The Bootloader Location Table (BOLT) describes the location of the MULTIBUS II System Architecture (MSA) second stage bootstrap loader. For the iRMX II Operating System, the MSA second stage bootstrap loader is in the file R?SECONDSTAGE. The MSA first stage bootstrap loader requires the BOLT to read and load the MSA second stage. The BOLT describes the location of the R?SECONDSTAGE file as a set of data blocks on the disk by listing the number of data blocks and the byte offset and length of each block. The BOLT also contains other information about the MSA second stage needed by the first stage.

The iRMX II Human Interface FORMAT command writes the BOLT structure to bytes 512 through 767 of a named volume. This replaces the area marked "uninitialized, reserved for future ISO standardization" in previous versions of the iRMX II Operating System.

The BOLT structure is as follows:

```
BOLT    STRUCTURE(
        RESERVED(4)                 DWORD,
        MAGIC_WORD1                 DWORD,
        MAGIC_WORD2                 DWORD,
        VERSION                     WORD,
        TYPES                       WORD
        DATA_SIZE                   DWORD,
        NUM_ENTRIES                 DWORD,
        TBL_ENTRY(NUM_ENTRIES)      STRUCTURE(
                                    BYTE_OFFSET     DWORD,
                                    LENGTH          WORD));
```

where:

RESERVED(4) — Reserved for future use. Set to 0.

MAGIC_WORD1 — A value which defines a valid MSA second stage bootloader image. This value is 0B00F10ADH.

MAGIC_WORD2 — Reserved for future use. Set to 0.

VERSION — The version of the BOLT structure. The BOLT structure listed here is version 2.

TYPES — Defines the type of code and data segments used in the second stage file to be bootloaded.

Bit	Meaning
0	Indicates the type of code segment.
	0 = Use16
	1 = Use32
1	Indicates the type of data segment.
	0 = Use16
	1 = Use32

The FORMAT command sets these bits to 0 (Use16).

DATA_SIZE — The size of the data segment for the second stage bootstrap loader.

NUM_ENTRIES — The number of entries in the table describing the second stage location. The iRMX II Operating System uses one entry in this table.

TBL_ENTRY (NUM_ENTRIES)	A table containing byte_offset and length pairs which indicate where the second stage is located on the media.	
	BYTE_OFFSET	The offset, in bytes, from the beginning of the media to this part of the second stage bootstrap loader.
	LENGTH	The length of this part of the second stage bootstrap loader.

A.4 INITIAL FILES

Any mechanism that formats iRMX named volumes must place seven files, with the option of an eighth file (iRMX I and iRMX II) and a ninth file (iRMX II only), on the volume during the format process. These files are

File	File Name
fnode file	
volume label file	R?VOLUMELABEL
volume free space map file	R?SPACEMAP
free fnodes map file	R?FNODEMAP
bad blocks file	R?BADBLOCKMAP
root directory	
space accounting file,	
Optionally, duplicate volume label file	R?SAVE
Optionally, MSA second stage file (iRMX II only)	R?SECONDSTAGE

The first of these files, the fnode file, contains information about all of the files on the volume. The general structure of the fnode file is discussed first. Then all of the files are discussed in terms of their fnode entries and their functions.

A.4.1 Fnode File

A data structure called a file descriptor node (fnode) describes each file in a named file volume. All the fnodes for the entire volume are grouped together in a file called the fnode file. When the I/O System accesses a file on a named volume, it examines the iRMX Volume Label (described in the previous section) to determine the location of the fnode file, and then examines the appropriate fnode to determine the actual location of the file.

When a volume is formatted, the fnode file contains seven allocated fnodes and any number of unallocated fnodes. The original number of unallocated fnodes depends on the FILES parameter of the FORMAT command. These allocated fnodes represent the fnode file, the volume label file, the volume free space map file, the free fnodes map file, the bad blocks file, the root directory, and the space accounting file. (Later sections of this appendix describe these files.) The size of the fnode file is determined by the number of fnodes that it contains. The number of fnodes in the fnode file also determines the number of files that can be created on the volume. The number of files is set when you format the storage medium.

The structure of an individual fnode in a named file volume is as follows:

```
DECLARE
    FNODE STRUCTURE(
            FLAGS           WORD,
            TYPE            BYTE,
            GRAN            BYTE,
            OWNER           WORD,
            CR$TIME         DWORD,
            ACCESS$TIME     DWORD,
            MOD$TIME        DWORD,
            TOTAL$SIZE      DWORD,
            TOTAL$BLKS      DWORD,
            POINTR(40)      BYTE,
            THIS$SIZE       DWORD,
            RESERVED$A      WORD,
            CHK$SUM         WORD,
            ID$COUNT        WORD,
            ACC(9)          BYTE,
            PARENT          WORD,
            AUX(*)          BYTE);
```

where:

FLAGS — A WORD that defines a set of attributes for the file. The individual bits in this word indicate the following attributes (bit 0 is the rightmost bit):

Bit	Meaning
0	Allocation status. If set to one, this fnode describes an actual file. If set to zero, this fnode is available for allocation. When formatting a volume, this bit is set to one in the seven allocated fnodes. In other fnodes, it is set to zero.
1	Long or short file attribute. This bit describes how the PTR fields of the fnode are interpreted. If set to zero, indicating a short file, the PTR fields identify the actual data blocks of the file. If set to one, indicating a long file, the PTR fields identify indirect blocks (described later in this section). When formatting a volume, this bit is always set to zero, since the initial files on the volume are short files.
2	Reserved bit, always set to one.
3-4	Reserved bits, always set to zero.
5	Modification attribute. Whenever a file is modified, this bit is set to one. Initially, when a volume is formatted, this bit is set to zero in each fnode.
6	Deletion attribute. This bit is set to one to indicate that the file is a temporary file or that the file will be deleted (the deletion may be postponed because additional connections exist to the file). Initially, when the volume is formatted, this bit is set to zero in each fnode.
7-15	Reserved bits, always set to zero.

TYPE — Type of file. The following are acceptable types:

Mnemonic	Value	Type
FT$FNODE	0	fnode file
FT$VOLMAP	1	volume free space map
FT$FNODEMAP	2	free fnodes map
FT$ACCOUNT	3	space accounting file
FT$BADBLOCK	4	device bad blocks file
FT$DIR	6	directory file
FT$DATA	8	data file
FT$VLABEL	9	volume label file

During system operation, only the I/O System can access file types other than FT$DATA and FT$DIR. These file types are discussed later in this section.

GRAN — File granularity, specified in multiples of the volume granularity. The default value is 1. This value can be set to any multiple of the volume granularity.

OWNER — User ID of the owner of the file. For the files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the user WORLD (FFFFH). The I/O System does not examine this parameter for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label), so a value of zero can be specified.

CR$TIME — Time and date that the file was created, expressed as a 32-bit value. This value indicates the number of seconds since a fixed, user-determined point in time. By convention, this point in time is midnight (00:00), January 1, 1978. For the files initially present on the volume, this parameter is important only for the root directory. A zero can be specified for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label.)

ACCESS$TIME — Time and date of the last file access (read or write), expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.

MOD$TIME — Time and date of the last file modification, expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.

TOTAL$SIZE — Total size, in bytes, of the actual data in the file.

TOTAL$BLKS — Total number of volume blocks used by this file, including indirect block overhead. A volume block is a block of data whose size is the same as the volume granularity. All memory in the volume is divided into volume blocks, which are numbered sequentially, starting with the block containing the smallest addresses (block 0). Indirect blocks are discussed later in this section.

POINTR(40) — A group of BYTES on which the following structure is imposed:

```
PTR(8) STRUCTURE(
            NUM$BLOCKS  WORD,
            BLK$PTR(3)  BYTE);
```

This structure identifies the data blocks of the file. These data blocks may be scattered throughout the volume, but together they make up a complete file. If the file is a short file (bit 1 of the FLAGS field is set to zero), each PTR structure identifies an actual data block. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS — Number of volume blocks in the data block.

BLK$PTR(3) — A 24-bit value specifying the number of the first volume block in the data block. Volume blocks are numbered sequentially, starting with the block with the smallest address (block 0). The bytes in the BLK$PTR array range from least significant (BLK$PTR(0)) to most significant (BLK$PTR(2)).

If the file is a long file (bit 1 of the FLAGS field is set to one), each PTR structure identifies an indirect block (possibly consisting of more than one contiguous volume block), which in turn identifies the data blocks of the file. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS — Number of volume blocks pointed to by the indirect block.

BLK$PTR(3) — A 24-bit volume block number of the indirect block.

Indirect blocks are discussed later in this section.

THIS$SIZE — Size, in BYTES, of the total data space allocated to the file. This figure does not include space used for indirect blocks, but it does include any data space allocated to the file, regardless of whether the file fills that allocated space.

RESERVED$A — Reserved field, set to zero.

CHK$SUM — Contains a checksum value for the fnode.

ID$COUNT — Number of access-ID pairs declared in the ACC(9) field.

ACC(9) — A group of BYTES on which the following structure is imposed:

```
ACCESSOR(3) STRUCTURE(
                ACCESS   BYTE,
                ID       WORD);
```

This structure contains the access-ID pairs that define the access rights for the users of the file. By convention, when a file is created, the owner's ID is inserted in ACCESSOR(0), along with the code for the access rights. The fields of the ACCESSOR structure contain the following:

ACCESS — Encoded access rights for the file. The settings of the individual bits in this field grant (if set to one) or deny (if set to zero) permission for the corresponding operation. Bit 0 is the rightmost bit.

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	list
2	append	add entry
3	update	change entry
4-7	reserved (must be 0)	

ID — ID of the user who gains the corresponding access permission.

PARENT — Fnode number of directory file that lists this file. For files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the number of the root directory's own fnode. For other files (fnode file, volume free space map file, free fnodes map file, bad blocks file, volume label) the I/O System does not examine this field.

AUX(*) — Auxiliary BYTES associated with the file. The named file driver does not interpret this field, but the user can access it by making GET$EXTENSION$DATA and SET$EXTENSION$DATA system calls. The size of this field is determined by the size of the fnode, specified in the iRMX Volume Label. If you use the Human Interface FORMAT command or create your own utility to format a volume, you can make this field as large as you wish; however, a larger AUX field implies slower file access.

Certain fnodes designate special files that appear on the volume. The following sections discuss these fnodes and the associated files.

A.4.2 Fnode 0 (Fnode File)

The first fnode structure in the fnode file describes the fnode file itself. This file contains all the fnode structures for the entire volume. It must reside in contiguous locations in the volume. The fields of fnode 0 must be set as follows:

- The bits in the FLAGS field are set to the following (bit 0 is the rightmost bit):

Bit	Value	Description
0	1	Allocated file
1	0	Short file
2	1	Primary fnode
3-4	0	Reserved bits
5	0	Initial status is unmodified
6	0	File will not be deleted
7-15	0	Reserved bits

- The TYPE field is set to FT$FNODE.
- The GRAN field is set to 1.
- The OWNER field is set to the ID of the user who formatted it.
- The CR$TIME, ACCESS$TIME, and MOD$TIME fields are set to the time the system was formatted.
- Since the iRMX Volume Label specifies the size of an individual fnode structure and the number of fnodes in the fnode file, the value specified in the TOTAL$SIZE field of fnode 0 must equal the product of the values in the FNODE$SIZE and MAX$FNODE fields of the iRMX Volume Label.
- The TOTAL$BLOCKS field specifies enough volume blocks to account for the memory listed in the TOTAL$SIZE field. The product of the value in the TOTAL$BLOCKS field and the volume granularity equals the value of the THIS$SIZE field, since the fnode file is a short file.
- Since the fnode file must reside in contiguous locations in the volume, only one PTR structure describes the location of the file. The value in the NUM$BLOCKS field of that PTR structure equals the value in the TOTAL$BLOCKS field. The BLK$PTR field indicates the number of the first block of the fnode file.
- The ID$COUNT field is set to one.

A.4.3 Fnode 1 (Volume Free Space Map File)

The second fnode, fnode 1, describes the volume free space map file. The TYPE field for fnode 1 is set to FT$VOLMAP to designate the file as such.

The volume free space map file keeps track of all the space on the volume. It is a bit map of the volume, in which each bit represents one volume block (a block of space whose size is the same as the volume granularity). If a bit in the map is set to one, the corresponding volume block is free to be allocated to any file. If a bit in the map is set to zero, the corresponding volume block is already allocated to a file. The bits of the map correspond to volume blocks such that bit n of byte m represents volume block (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual blocks of memory) must be set to zero.

When the volume is formatted, the volume free space map file indicates that the first 3328 bytes of the volume (the label and bootstrap information) plus any files initially placed on the volume (fnode file, volume free space map file, free fnodes map file, bad blocks file) are allocated. Space is also reserved for the R?SAVE and R?SECONDSTAGE files if they are selected during formatting.

A.4.4 Fnode 2 (Free Fnodes Map File)

The third fnode, fnode 2, describes the free fnodes map file. The TYPE field of fnode 2 is set to FT$FNODEMAP to designate the file as such.

The free fnodes map file keeps track of all the fnodes in the fnodes file. It is a bit map in which each bit represents an fnode. If a bit in the map is set to one, the corresponding fnode is not in use and does not represent an actual file. If a bit in the map is set to zero, the corresponding fnode already describes an existing file. The bits in the map correspond to fnodes such that bit n of byte m represents fnode number (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual fnode structures) must be set to zero.

When the volume is formatted, the free fnodes map file indicates that fnodes 0, 1, 2, 3, 4, 5, and 6 are in use. If either the RESERVE option or the MSABOOT option (iRMX II only) are selected when the volume is formatted, the map file also indicates fnode 7 is in use. If both options are selected, fnode 8 is also used. If other files are initially placed on the volume, the free fnodes map file must be set to indicate this as well.

A.4.5 Fnode 3 (Accounting File)

When a volume is formatted, fnode 3 is set up representing a file of type FT$ACCOUNT. The fnode is set up as allocated, and of the indicated type, but it does not assign any actual space for the file.

A.4.6 Fnode 4 (Bad Blocks Map File)

The fifth fnode, fnode 4, contains a map of all the bad blocks on the volume. The TYPE field of fnode 4 is set to FT$BADBLOCK to indicate this.

The bad block map file keeps track of all the bad blocks on the volume. It is a bit map of the volume, in which each bit represents one volume block (a block of space whose size is the same as the volume granularity). If a bit in the map is set to zero, the corresponding volume block has no bad blocks and may be allocated to any file. If a bit in the map is set to one, the corresponding volume block is bad. If a block is marked "bad," it must also be marked allocated in the volume free space file. The bits of the map correspond to volume blocks such that bit n of byte m represents volume block (8 * m) + n.

A.4.7 Fnode 5 (Volume Label File)

This fnode contains the first 3328 bytes of any volume. The information in this file defines the volume as a whole. The TYPE field of this fnode is set to FT$VLABEL. You cannot write to this fnode.

A.4.8 Fnode 6 (Root Directory)

The root directory is a special directory file. It is the root of the named file hierarchy for the volume. The iRMX Volume Label specifies the fnode number of the root directory. The root directory is its own parent. That is, the PARENT field of its fnode specifies its own fnode number.

The root directory (and all directory files) associates file names with fnode numbers. It consists of a number of entries that have the following structure:

```
DECLARE
    DIR$ENTRY    STRUCTURE(
                     FNODE           WORD,
                     COMPONENT(14)   BYTE);
```

where:

FNODE	Fnode number of a file listed in the directory.
COMPONENT(14)	A string of ASCII characters that is the final component of the path name identifying the file. This string is left justified and null padded to 14 characters.

When a file is deleted, its fnode number in the directory entry is set to zero.

A.4.9 Fnodes 7 and 8 (R?SECONDSTAGE and R?SAVE)

These fnodes may or may not be reserved depending on whether the RESERVE and MSABOOT (iRMX II only) options are used during formatting. If both options are used, the R?SECONDSTAGE file is placed in fnode 7 and the R?SAVE file is placed in fnode 8. If only RESERVE is used, R?SAVE is placed in fnode 7 and fnode 8 remains unallocated. If only MSABOOT (iRMX II only) is used, R?SECONDSTAGE is placed in fnode 7 and fnode 8 remains unallocated. If neither option is used, both fnode 7 and fnode 8 remain unallocated.

A.4.9.1 R?SECONDSTAGE

R?SECONDSTAGE is a file which may be optionally created by the MSABOOT option of the FORMAT command. R?SECONDSTAGE is the second stage bootloader for systems that conform to the MULTIBUS II System Architecture (MSA) specification. R?SECONDSTAGE is created at the end of the volume. However, if the RESERVE option is also specified, R?SECONDSTAGE will be placed in the volume blocks immediately preceeding R?SAVE. (The fnode for the R?SECONDSTAGE file is allocated out of the fnodes reserved through the FILES parameter of the FORMAT command.)

A.4.9.2 R?SAVE

R?SAVE is a file which may be optionally created by the RESERVE option of the FORMAT command. The FORMAT command creates a file named R?SAVE, which contains the duplicate volume label, in the innermost track of the volume. A copy of the iRMX volume label is placed at the physical end of the file and an fnode is allocated for R?SAVE in the fnode file. (The fnode for the R?SAVE file is allocated out of the fnodes reserved through the FILES parameter of the FORMAT command.)

The FORMAT command creates a backup of the fnode file in its initialized state. R?SAVE is not subsequently updated as files are written to or deleted from the volume. Therefore, you will have to use the BACKUPFNODES Disk Verification Utility command or the BACKUP option of the Human Interface SHUTDOWN command to back up the fnode file at regular intervals.

A.4.10 Other Fnodes

When formatting a volume, no other fnodes in the fnode file represent actual files. The remaining fnodes must have bit zero (allocation status) set to zero.

A.5 LONG AND SHORT FILES

A file on a volume is not necessarily one contiguous string of bytes. In many cases, it consists of several blocks of data scattered throughout the volume. The fnode for the file indicates the locations and sizes of these blocks in one of two ways, as short files or as long files.

A.5.1 Short Files

If the file consists of eight or less distinct blocks of data, its fnode can specify it as a short file. The fnode for a short file has bit 1 of the FLAGS field set to zero. This indicates to the I/O System that the PTR structures of the fnode identify the actual data blocks that make up the file. Figure A-2 illustrates an fnode for a short file. Decimal numbers are used in the figure for clarity.

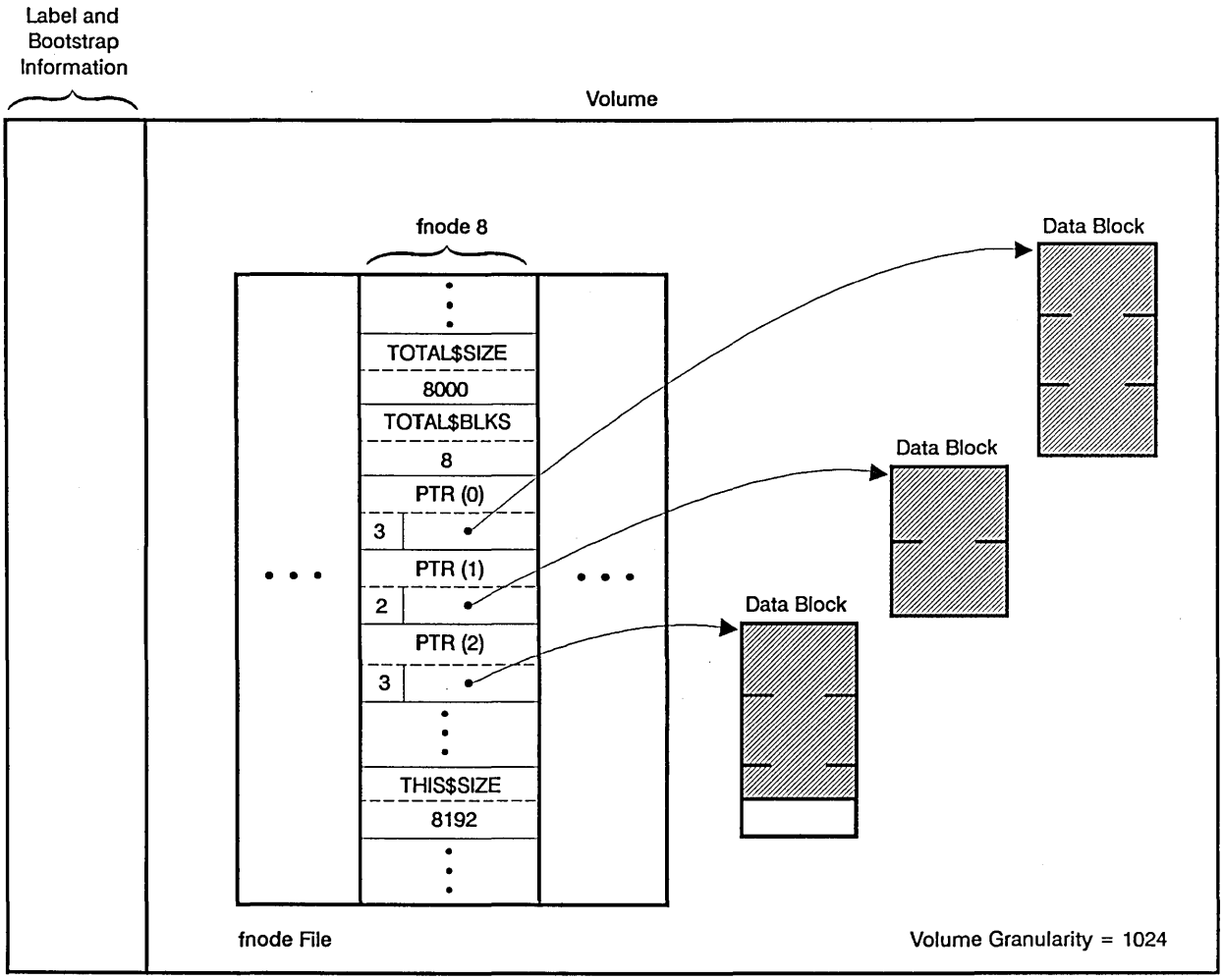


Figure A-2. Short File Fnode

As you can see in Figure A-2, fnode 8 identifies the short file. The file consists of three distinct data blocks. Three PTR structures give the locations of the data blocks. The NUM$BLOCKS field of each PTR structure gives the length of the data block (in volume blocks), and the BLK$PTR field points to the first volume block of the data block.

The other fields shown in Figure A-2 include TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is eight. This equals the sum of NUM$BLOCKS values (3 + 2 + 3), since short files use all allocated space as data space.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file. This is the sum of the NUM$BLOCKS values (3 + 2 + 3) multiplied by the volume granularity (1024) and equals 8192.

The TOTAL$SIZE field specifies the number of bytes of data space that the file occupies (designated in Figure A-2 by the shaded area). As you can see, the file does not occupy all the space allocated for it, so the TOTAL$SIZE value (8000) is not as large as the THIS$SIZE value.

A.5.2 Long Files

If the file consists of more than eight distinct blocks of data, its fnode must specify it as a long file. The fnode for a long file has bit 1 of the FLAGS field set to one. This tells the I/O System that the PTR structures of the fnode identify indirect blocks. The indirect blocks identify the actual data blocks that make up the file.

Each indirect block contains a number of indirect pointers, which are structures similar to the PTR structures. However, an indirect block can contain more than eight structures and thus can point to more than eight data blocks. In fact, an indirect block can consist of more than one volume block; however, all volume blocks of an indirect block must be contiguous. The structure of each indirect pointer is as follows:

```
DECLARE
    IND$PTR STRUCTURE(
                NBLOCKS    BYTE,
                BLK$PTR    BLOCK$NUM);
```

where:

NBLOCKS	Number of volume blocks in the data block.
BLK$PTR	A 24-bit volume block number of the first volume block in the data block. Volume blocks are numbered sequentially throughout the volume, starting with the block with the smallest address (block 0).

The operating system determines how many indirect pointers there are in an indirect block by comparing the NBLOCKS fields of the indirect pointers with the NUM$BLOCKS field of the fnode. It assumes that the indirect block contains as many pointers as necessary for the sum of the NBLOCKS fields to equal the NUM$BLOCKS field.

Because indirect blocks can span several volume blocks, any utility that uses indirect blocks must determine if an indirect block consists of more than one volume block. To do this, the utility should do the following:

1. Read the volume block pointed to by the BLK$PTR field in the fnode's POINTR structure. BLK$PTR points to the beginning of a volume block containing all or the first part of an indirect block.
2. If the sum of all NBLOCKS fields in the volume block is less than NUM$BLOCKS, the indirect block continues into the next contiguous volume block. The utility must read and process the next volume block.
3. Add the NBLOCKS values in the new volume block to the sum of all previous NBLOCKS. When the sum of the NBLOCK values equals NUM$BLOCKS you have reached the end of the indirect block. If necessary, continue reading volume blocks and summing NBLOCKS values until the sum of the NBLOCKS values equals NUM$BLOCKS. The utility may have to read several volume blocks before finding the end of the indirect block.

Figure A-3 illustrates an fnode for a long file. Decimal numbers are used in the figure for clarity.

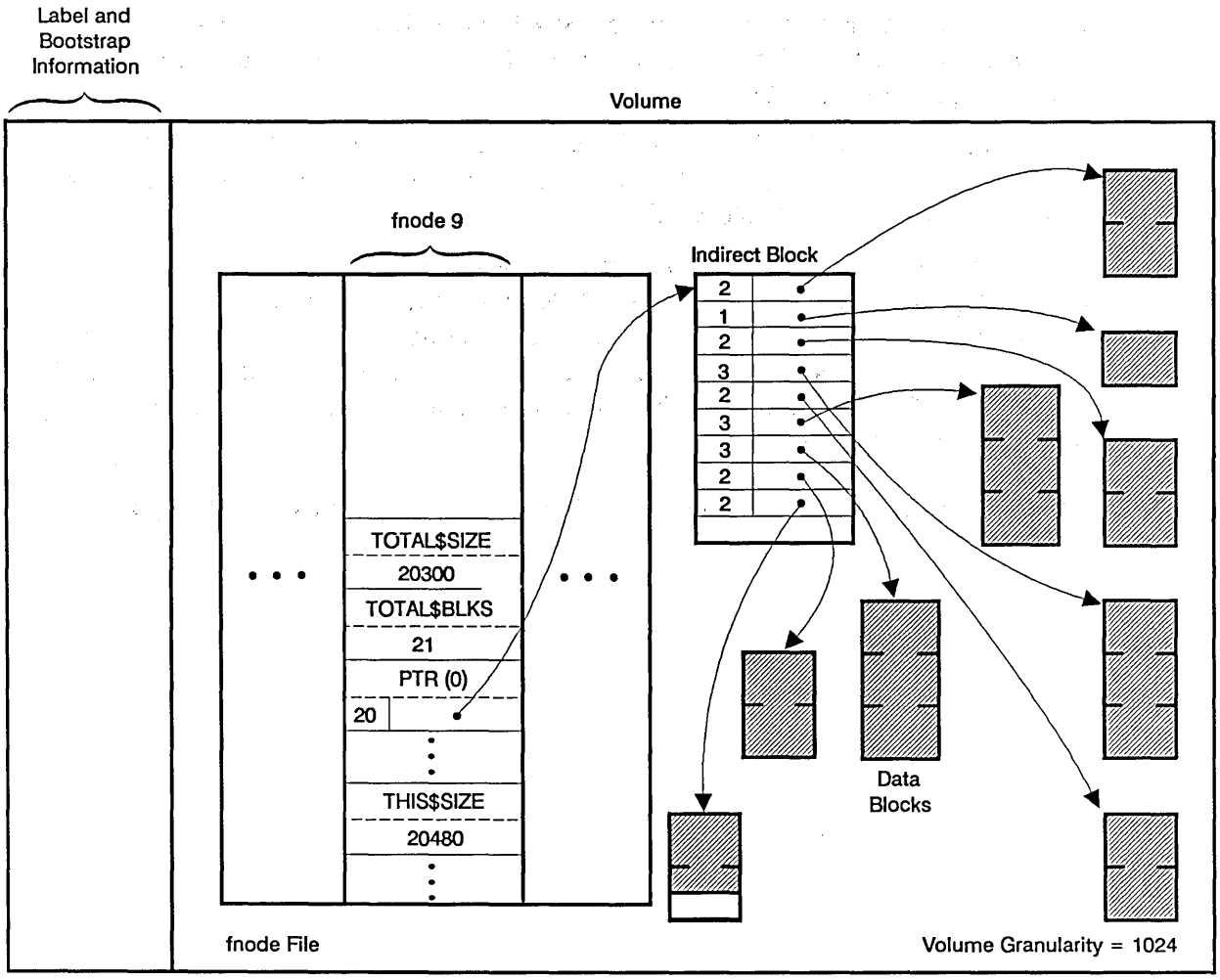


Figure A-3. Long File Fnode

As you can see in Figure A-3, fnode 9 identifies the long file. The actual file consists of nine distinct data blocks. One PTR structure and an indirect block give the locations of the data blocks. The NUM$BLOCKS field of the PTR structure contains the number of volume blocks pointed to by the indirect block. The BLK$PTR field points to the first volume block of the indirect block.

In the indirect block, each NBLOCKS field gives the length of an individual data block, and each BLK$PTR field points to the first volume block of a data block.

Figure A-3 also lists the TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE values, which are more complex than for a short file. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is 21. Of these 21, 20 are used for actual data storage and 1 is used for the indirect block.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file, and does not include the size of the indirect block. This size is equal to the NUM$BLOCKS value (20) or the sum of NBLOCKS values in the indirect block (2 + 1 + 2 + 3 + 2 + 3 + 3 + 2 + 2 = 20) multiplied by the volume granularity (1024) and equals 20480.

The TOTAL$SIZE field specifies the number of bytes of data space that the file currently occupies (designated in Figure A-3 by the shaded areas). As you can see, the file does not occupy all the space allocated for it, so the TOTAL$SIZE value (20300) is not as large as the THIS$SIZE value.

A.6 FLEXIBLE DISKETTE FORMATS

The flexible diskette device drivers supplied with the iRMX I and iRMX II Basic I/O Systems can support several diskette characteristics, listed in Tables A-1 and A-2.

Table A-1. 8-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Format	Device Size (in bytes) One-Sided	Two-Sided
128	Single	26	Standard	256256	512512
256	Single	15	Standard	295168	590848
512	Single	8	Standard	314880	630272
1024	Single	4	Standard	315392	630784
256	Double	26	Standard	509184	1021696
512	Double	15	Standard	587264	1177600
1024	Double	8	Standard	626688	1255424

Table A-2. 5 1/4-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Format	Device Size (in bytes) One-Sided 40 Tracks	One-Sided 80 Tracks	Two-Sided 40 Tracks	Two-Sided 80 Tracks
128	Single	16	Standard	81920	163840	163840	327680
256	Single	9	Standard	91904	184064	184064	368384
512	Single	4	Standard	81920	163840	163840	327680
1024	Single	2	Standard	81920	163840	163840	327680
256	Double	16	Standard	1617921	325632	325632	653312
512	Double	8	Standard	1617921	325632	325632	653312
512	Double	9	Uniform	--	--	368640	--
1024	Double	4	Standard	1617921	325632	325632	653312
512	Quad *	15	Uniform	--	--	--	1228800

* Only supported in the iRMX® II Operating System.

For compatibility with ECMA (European Computer Manufacturers Association) and ISO (International Organization for Standardization), the iRMX device drivers, when called by the Human Interface FORMAT command, can format the beginning tracks of all flexible diskettes in the same way. A configuration option for each driver enables you to specify the following:

- For all 5 1/4-inch and 8-inch flexible diskettes, the device drivers format track 0 of side 0 with single-density, 128-byte sectors, with an interleave factor of 1.

- For 8-inch, double-sided, double-density flexible diskettes, the device drivers format track 0 of side 1 with double-density, 256-byte sectors.

The iRMX device drivers map the sectors on these beginning tracks into blocks of device granularity size so that the Basic I/O System and the Bootstrap Loader can treat flexible diskettes as if they contained a contiguous string of blocks, all of the same size.

However, this mapping is not exact when you use 8-inch, double-sided, double-density diskettes and specify a device granularity of 512 or 1024. A problem arises because there are 26 128-byte sectors in a track, which is not an integral mapping for device granularities of 512 or 1024. Thus, the device driver combines the leftover 128-byte sectors of track 0, side 0 with the first sectors of track 0, side 1 to make a block of device granularity size. This continues throughout track 0, side 1, but the same problem occurs with the last 256-byte sectors of track 0, side 1; not enough sectors are available to make a block of device granularity size.

When the device driver tries to combine these leftover sectors of track 0, side 1 with the first sectors of track 1, side 0, it finds that the sectors of track 1, side 0 are already of device granularity size. Therefore, since the device driver cannot access partial sectors, it is left with one block (the leftover sectors of track 0, side 1) that is less than device granularity size. When the device granularity is 512, this small block is block 19; when the device granularity is 1024, it is block 9.

If nothing is done to exclude this smaller-than-normal block from use, the device driver will treat this block as a normal block, assuming it is of device granularity size. Thus, if you try to write information to that block, the driver will attempt to write an entire device granularity block of information into a block that is much smaller, thereby losing data.

To prevent this situation, the Human Interface FORMAT command automatically declares this smaller-than-normal block as allocated in the volume free space map when it formats the volume. This prevents the Basic I/O System from ever writing information into this block. If you write your own formatting utility, you should also declare this block as allocated.

INDEX

A

B

C

C (continued)

C (continued)

D

D (continued)

E

E (continued)

E (continued)

F

F (continued)

G

H

I

L

M

N

O

P

Q

R

S

T

V

W

INTERNATIONAL SALES OFFICES

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

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

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

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

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

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

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

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

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

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

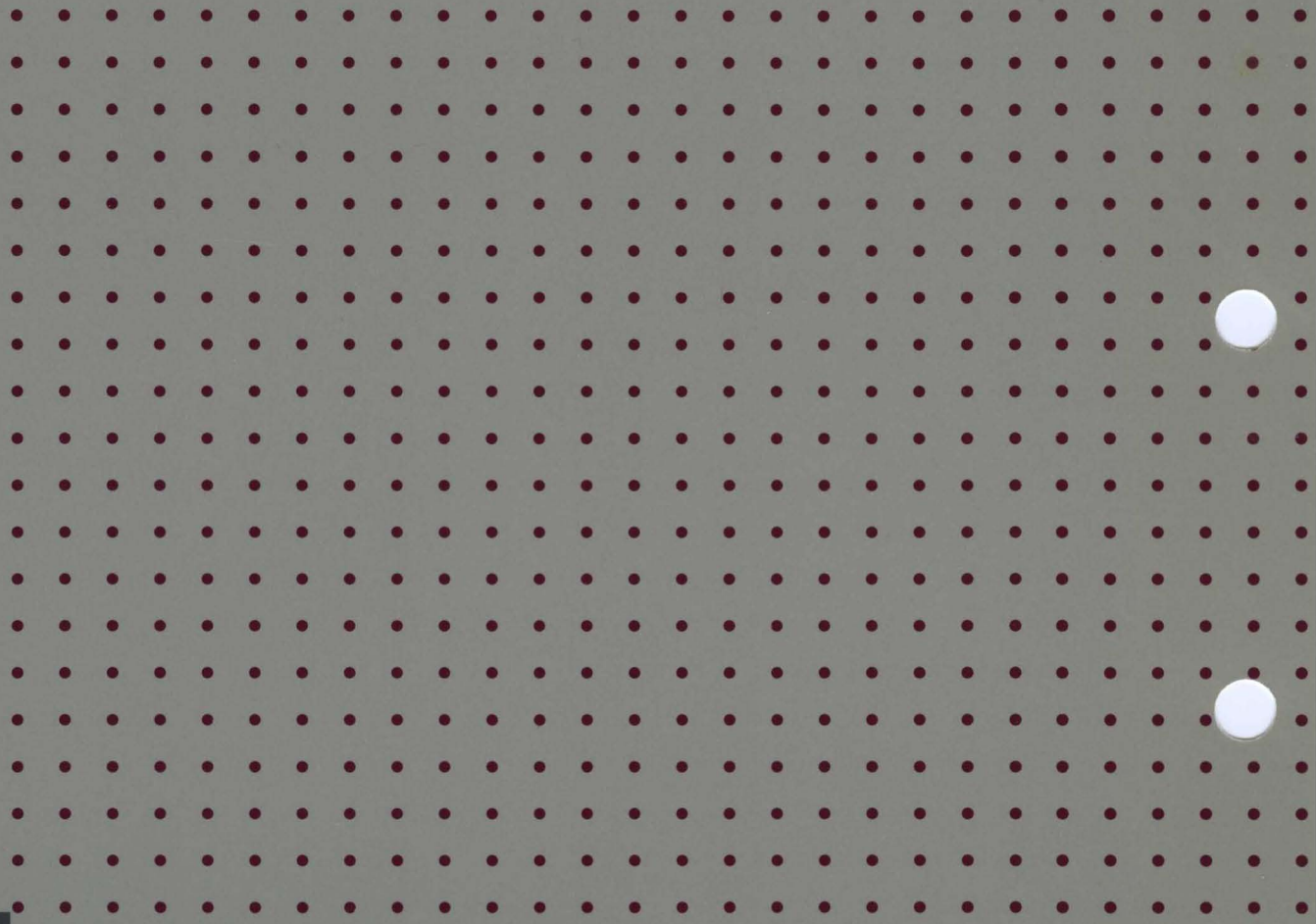
NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

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

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

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



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