RT–11 Volume and File Formats Manual

Order Number AA–PD6PA–TC

August 1991

This manual describes the format of RT–11 disk and magtape volumes and the RT–11 file formats. This information will be most useful to system programmers, but it provides valuable background information for application programmers as well.

Revision/Update Information:	This manual replaces Chapters 8 and 9 of the *RT–11 Software Support Manual*.
Operating System:	RT–11 Version 5.6

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, August 1991

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Any software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. No responsibility is assumed for the use or reliability of software or equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227–7013.

© Digital Equipment Corporation 1991
All rights reserved. Printed in U.S.A.

The Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation: CTS–300, DDCMP, DECnet, DECUS, DECwriter, DIBOL, MASSBUS, MicroPDP–11, Micro/RSX, PDP, Professional, Q-bus, RSTS, RSX, RT–11, RTEM–11, UNIBUS, VMS, VT, and the DIGITAL Logo.

S561

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

Index

Figures

Tables

Preface

Document Structure

This manual is divided into two chapters:

- Chapter 1 describes the file structure for all disk and magtape volumes that RT–11 supports.
- Chapter 2 alphabetically presents information about the structure of supported file formats.

Audience

This manual is written for those users of the RT–11 operating system who want to understand the structure of RT–11 volume and file formats.

Conventions

The following conventions are used in this manual.

Convention	Meaning

Black print	In examples, black print indicates output lines or prompting characters that the system displays. For example:
	`.BACKUP/INITIALIZE DL0:F*.FOR DU1:WRK`
	`Mount output volume in DU1:; continue? Y`

Red print	In examples, red print indicates user input.
Braces ({ })	In command syntax examples, braces enclose options that are mutually exclusive. You can choose only one option from the group of options that appear in braces.
Brackets ([ ])	Square brackets in a format line represent optional parameters, qualifiers, or values, unless otherwise specified.
Lowercase characters	In command syntax examples, lowercase characters represent elements of a command for which you supply a value. For example: `DELETE filespec`

Convention	Meaning
UPPERCASE characters	In command syntax examples, uppercase characters represent elements of a command that should be entered exactly as given.
RET	RET in examples represents the RETURN key. Unless the manual indicates otherwise, terminate all commands or command strings by pressing RET.
CTRL/*x*	CTRL/*x* indicates a control key sequence. While pressing the key labeled Ctrl, press another key. For example: CTRL/C

Associated Documents

Basic Books

- *Introduction to RT–11*
- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*
- *RT–11 Commands Manual*
- *RT–11 Quick Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

- *RT–11 Automatic Installation Guide*
- *RT–11 Installation Guide*
- *RT–11 System Generation Guide*

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 System Internals Manual*
- *RT–11 Device Handlers Manual*

- *DBG–11 Symbolic Debugger User's Guide*

Chapter 1
Disk and Tape Formats

RT–11 stores files under assigned file names on file-structured volumes. RT–11 volumes that are file-structured include all disks, diskettes, and magtapes.

Disks and diskettes are directory-structured volumes; they have a series of directory segments at the beginning of the volume. The directory segments contain entries describing the names, lengths, and creation dates of files on the volume. Because the directory is at the beginning of the volume, you can access any file, no matter where it is located, without reading any other files. For this reason, directory-structured volumes are sometimes called random-access or block-replaceable volumes.

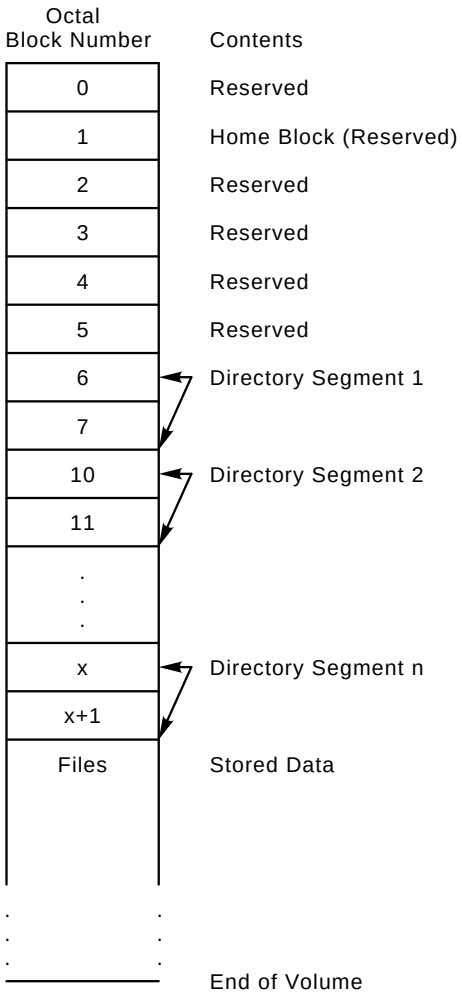
Magtapes are file-structured volumes that do not have a directory at the beginning of the volume. While they do store some directory information at the beginning of each file, you must read the entire volume to obtain all the information about all the files. Because you must read the files in order, one after the other, magtapes are also called sequential-access volumes.

This chapter shows how RT–11 stores files on both random-access and sequential-access volumes. It also describes the contents of a volume directory and shows how to recover information from a random-access volume whose directory is corrupted.

1.1 Random-Access Volumes

A random-access volume consists of a series of 256_{10}-word blocks where blocks 0 through 5 are reserved for system use and cannot be used for data storage. The volume directory begins at block 6. Figure 1–1 shows the format of a random-access volume.

Figure 1–1: Random-Access Volume



1.1.1 Home Block

Block 1 of a random-access volume, called the **home block**, contains information about the volume and its owner. Figure 1–2 and Table 1–1 show the home block format and contents.

Figure 1–2: Home Block Format

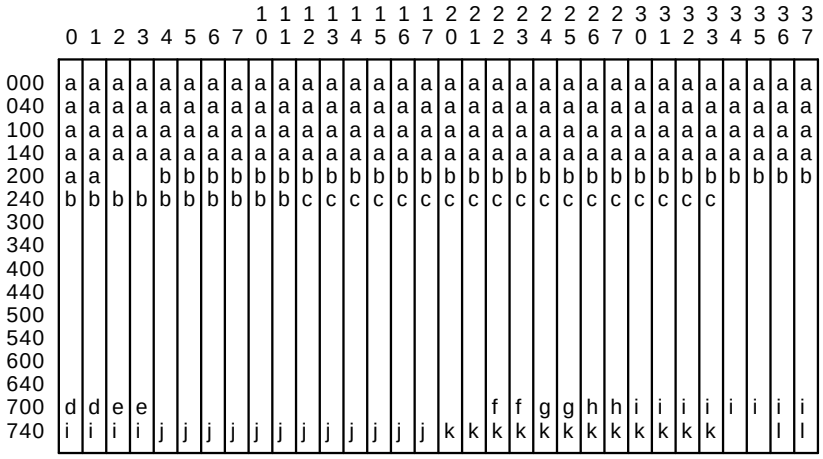


Table 1–1: Home Block Contents

Field	Location	Contents	Default
a	$000–201_8$	Bad block replacement table	
b	$204–251_8$	INITIALIZE/RESTORE data area	
c	$252–273_8$	BUP information area	If BUP volume: "BUQ" and 9 spaces; binary volume number in byte 266_8
d	$700–701_8$	(Reserved for Digital)	000000
e	$702–703_8$	(Reserved for Digital)	000000
f	$722–723_8$	Pack cluster size	000001
g	$724–725_8$	Block number of first directory segment	000006
h	$726–727_8$	System version	Radix–50 "V3A"
i	$730–743_8$	Volume Identification	"RT11A" and seven spaces
j	$744–757_8$	Owner name	12_{10} spaces
k	$760–773_8$	System Identification	"DECRT11A" and four spaces
l	$776–777_8$	Checksum	

The checksum computation conforms to the *FILES-11 On-Disk Structure Specification*. It is a simple additive checksum of all the other words in the home block and is computed according to the following algorithm:

```
        MOV     Header__address,R0
        CLR     R1
        MOV     #255.,R2
10$:    ADD     (R0)+,R1
        SOB     R2,10$
        MOV     R1,@R0
```

The contents of all other areas in the home block are undefined and reserved for future use by Digital.

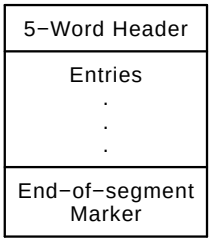
1.1.2 Directory Structure

The directory consists of a series of two-block segments. Each segment is 512_{10} words long and contains information about files such as name, length, and creation date. A directory can have from 1 to 31_{10} segments. You can display the default number of directory segments on an initialized volume by issuing the DIRECTORY/SUMMARY command. You can change the size of the directory area during volume initialization by specifying the number of directory segments. Use the INITIALIZE/SEGMENTS:n command, or DUP with the /Z/N:n options. (See the *RT–11 Commands Manual* for more information on the INITIALIZE command. See the *RT–11 Installation Guide* for a patch that changes the default number of segments.)

In general, you require more segments if you need to store many small files on a large volume. However, the minimal number of segments reduces the size of the directory area and you obtain more space to store large files on a smaller volume. The default number of directory segments is a compromise.

Each directory segment consists of a five-word header plus a number of entries containing file information. Each segment ends with an end-of-segment marker. Figure 1–3 shows the general format of a volume directory segment.

Figure 1–3: Volume Directory Segment Format



1.1.2.1 Directory Segment Header Format

Each directory segment contains a five-word header, which leaves the remaining 507_{10} words of the two-block segment for directory entries. Table 1–2 describes the contents of the header words.

Table 1–2: Directory Segment Header Words

Word	Contents
1	The total number of segments in this directory. The valid range is from 1 through 31_{10}. If you do not specify the number of segments you require when you initialize the volume, DUP uses the default number of segments for that volume.
2	The segment number of the next logical directory segment. The directory is a linked list of segments and this word is the link between the current segment and the next logical segment. If this word is 0, there are no more segments in the list.
3	The number of the highest segment currently in use. RT–11 increments this counter each time it opens a new segment. Note that the system maintains this counter in word 3 of the header only for the first directory segment. It completely ignores the third word of the header of the other segments.
4	The number of extra bytes per directory entry, always an unsigned, even octal number. See Section 1.1.2.2 for more information.
5	The block number on the volume where the actual stored data identified by this segment begins.

1.1.2.2 Directory Entry Format

The remainder of the directory segment consists of one or more directory entries followed by an end-of-segment marker. Figure 1–4 shows the format of a directory entry.

The first word of each directory entry is the status word, which describes the condition of the file identified by that directory entry. The high-order byte of the status word contains information on the current state of file. The low-order byte can identify a file class. Figure 1–5 shows the format of the status word and Table 1–3 describes the contents.

File Type

RT–11 uses three kinds of directory entries to describe the type of file:

- tentative entries
- empty entries
- permanent entries

Figure 1–4: Directory Entry Format

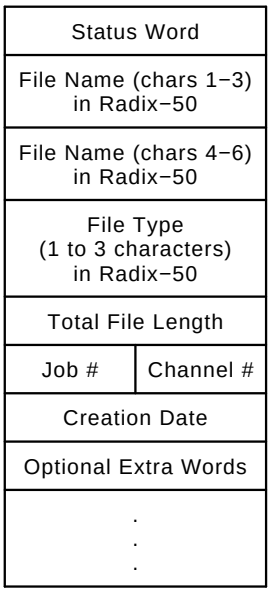
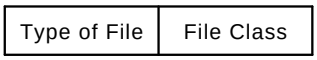


Figure 1–5: Directory Entry Status Word Format



You can think of the three types of entry as describing areas that are categorized as temporary data, available space on the volume, or permanent data. The volume directory contains at all times sufficient entries to describe the entire volume.

A **tentative file** is a file that is in the process of being created. When a program issues the .ENTER programmed request, for example, it creates a tentative file. The program must issue a .CLOSE programmed request to make the tentative file permanent. If you do not eventually close a tentative file, the system deletes it. The DIR utility program lists tentative files that appear in directories as <UNUSED> files.

An **empty entry** defines an area of the volume that is available for use. Thus, when you delete a file, you obtain an empty area. DIR lists an empty area as <UNUSED>, followed by its length.

A **permanent file** is a tentative file that has been closed with the .CLOSE programmed request. Permanent files are unique – that is, only one file can exist with a specific name and file type on a volume. If another file exists with the same name and type when the program closes the current tentative file, the monitor deletes the first file as part of the .CLOSE routine, thus replacing the old file with

the new file. DIR lists permanent files that appear in directories by their file names, file types, sizes, and creation dates.

The file class information in the low byte of the status word either marks a file as having particular characteristics or identifies the file as a member of a particular group of files. At present, only one bit is used; the rest are reserved for future use by Digital.

The **prefix block** bit indicates that the file contains at least one prefix block. Prefix blocks are optional information blocks that precede a file's information blocks and begin at logical block 0 of a file. The utility or application using the prefix blocks must set bit E.PRE in the status word, create the prefix blocks, and maintain them. RT–11 provides no special support for prefix blocks.

Any utility that needs to store information about a file with the file can use prefix blocks. All previous and current utilities and applications can continue to read and write files without regard for prefix block functionality. Prefix block functionality need not concern you unless you write an application that uses it.

The structure of prefix blocks reserved for use by RT–11 and the recommended structure for user applications is as follows:

- The low byte of the first word of the first prefix block contains the number of prefix blocks in the file.

  The high bit of the high byte indicates whether the prefix blocks were created by distributed RT–11 software of a user application. If the high bit is clear (0), distributed RT–11 software created the prefix blocks; if set (1), a user application created them.

- The second and third words of the first block contain a Radix–50 format identifier.
- Subsequent words in the prefix blocks are application-dependent.

Table 1–3 lists the valid values for the bytes of the status word and their meanings. All undefined bits in the directory entry status word remain reserved for Digital.

Table 1–3: Directory Entry Status Word Values

Value	Mnemonic	Meaning
High Byte Values, Expressed as a Word		
000400_8	E.TENT	Tentative file.
001000_8	E.MPTY	Empty area. RT–11 does not use the name, file type, or date fields in the directory entry for an empty area.
002000_8	E.PERM	Permanent file.

Table 1–3 (Cont.): Directory Entry Status Word Values

Value	Mnemonic	Meaning
High Byte Values, Expressed as a Word		
004000_8	E.EOS	End-of-segment marker. RT–11 uses this marker to determine when it has reached the end of the directory segment during a directory search. Note that an end-of-segment marker can appear as the last word of a segment. It does not have to be followed by a name, file type, or other entry information.
040000_8	E.READ	Protected by monitor from write operations from .WRITE request; not protected from any special function operations or deletions.
100000_8	E.PROT	Protected permanent file (see Section 1.1.2.3). Only a permanent file can be protected.
Low Byte Values, Expressed as a Word		
000020	E.PRE	Prefix block indicator. Indicates the presence of at least one prefix block in this file.

The second, third, and fourth words in a directory entry contain the file name and file type in Radix–50. For empty area, RT–11 normally ignores these words. However, the DIR /Q option (or the monitor DIRECTORY command with the /DELETED option) lists the names and file types of deleted files.

The fifth word in a directory entry contains the total file length, which consists of the number of blocks the file occupies on the volume. Attempts to read or write outside the limits of the file result in an end-of-file error.

The sixth word in a directory entry contains the channel number and sometimes the job number as well. RT–11 uses this information only for tentative files. A tentative file is associated with a job in one of two ways, depending on which RT–11 monitor is running.

The low byte of the sixth word of the directory entry contains the channel number. The high byte of the sixth word contains the number of the job that is opening the file. The job number is required to identify the correct tentative file during the .CLOSE operation. It is also necessary because several jobs can have files open using the same channel number.

NOTE

RT–11 uses the sixth word (job number and channel number word) only when the file is tentative. Once the file becomes permanent, RT–11 no longer uses the word. The function of the sixth word while the file is permanent is reserved for future use by Digital.

The seventh word of a directory entry contains the file's creation date. When a program creates a tentative file by issuing the .ENTER programmed request, the system moves the system date word into the creation date slot for the entry. The date word is 0 if you did not enter a date with the DATE monitor command.

User programs can take advantage of the *Age* bits (bits 14 and 15) to extend the directory date by 32_{10} year increments. Support for Age bits is not available in any RT–11 monitor or utility, but is included in the DATE/DATE4Y, IDATE, IDCOMP, and IGTDIR subroutines and functions. The base year for calculation is incremented by the Age bit values (0-3) and is 1972, 2004, 2036, and 2068, respectively. Figure 1–6 shows the format of the date word.

Figure 1–6: Date Word Format

15 14	13 12 11 10	9 8 7 6 5	4 3 2 1 0	
Age 0–3	Month, 1–12	Day, 1–31	Year minus 1972 minus 32 X Age	All Numbers Are Decimal

Normally, directory entries are seven words long, but by using DUP with the /Z:n option, you can allocate extra words for each entry when you initialize the volume. The fourth word of the directory header contains the number of extra bytes you specify. Although DUP lets you allocate extra words, RT–11 provides no means to easily manipulate this extra information. Any program that needs to access these words must perform its own operations on the RT–11 directory. In addition, programs that manipulate the directory should use bit test (BIT) instructions, rather than compare (CMP) instructions.

1.1.2.3 File Protection

RT–11 provides a mechanism to prevent a file from being deleted. A file is protected when the high bit of its status word is set. Only permanent files can be protected. You can protect and unprotect files by using the PIP /R option or the monitor RENAME command. Note that a protected file is **not** protected against modification, only against deletion. For more information, see the *RT–11 Commands Manual*.

1.1.2.4 Sample Directory Segment

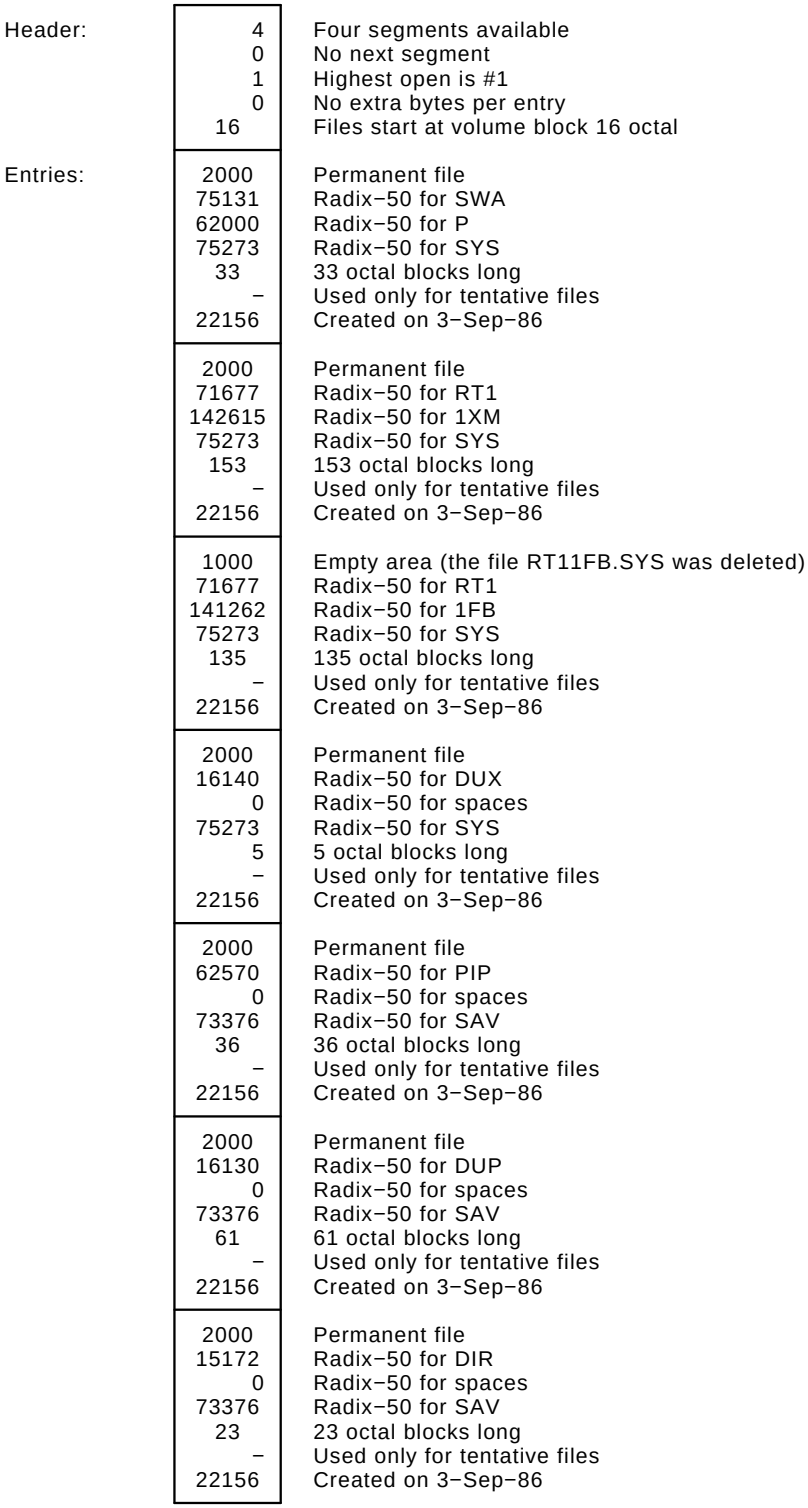
The directory listing shown in Figure 1–7 describes an RX50 diskette with 10 files.

Figure 1–8 shows the contents of segment 1 of the diskette directory, obtained by dumping absolute block number 6 of the volume.

Figure 1–7: Directory Listings

```
DIRECTORY/FULL DU0:

 12-Jul-88
SWAP  .SYS    27  03-Sep-86      RT11XM.SYS   107  03-Sep-86
< UNUSED >    93                 DUX   .SYS     5  03-Sep-86
PIP   .SAV    30  03-Sep-86      DUP   .SAV    49  03-Sep-86
DIR   .SAV    19  03-Sep-86      KED   .SAV    58  03-Sep-86
MACRO .SAV    63  13-Nov-87      LINK  .SAV    49  03-Sep-86
CREF  .SAV     6  13-Nov-87      < UNUSED >   280
 10 Files, 413 Blocks
 373 Free blocks

DIRECTORY/SUMMARY DU0:

 11-Jul-88

    10 Files in segment 1

     4 Available segments, 1 in use

 10 Files, 413 blocks
 373 Free blocks
```

Figure 1–8: RT–11 Directory Segment



Header:	4	Four segments available
	0	No next segment
	1	Highest open is #1
	0	No extra bytes per entry
	16	Files start at volume block 16 octal
Entries:	2000	Permanent file
	75131	Radix−50 for SWA
	62000	Radix−50 for P
	75273	Radix−50 for SYS
	33	33 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	2000	Permanent file
	71677	Radix−50 for RT1
	142615	Radix−50 for 1XM
	75273	Radix−50 for SYS
	153	153 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	1000	Empty area (the file RT11FB.SYS was deleted)
	71677	Radix−50 for RT1
	141262	Radix−50 for 1FB
	75273	Radix−50 for SYS
	135	135 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	2000	Permanent file
	16140	Radix−50 for DUX
	0	Radix−50 for spaces
	75273	Radix−50 for SYS
	5	5 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	2000	Permanent file
	62570	Radix−50 for PIP
	0	Radix−50 for spaces
	73376	Radix−50 for SAV
	36	36 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	2000	Permanent file
	16130	Radix−50 for DUP
	0	Radix−50 for spaces
	73376	Radix−50 for SAV
	61	61 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86
	2000	Permanent file
	15172	Radix−50 for DIR
	0	Radix−50 for spaces
	73376	Radix−50 for SAV
	23	23 octal blocks long
	−	Used only for tentative files
	22156	Created on 3−Sep−86

Figure 1–8 (continued on next page)

Figure 1–8 (Cont.): RT–11 Directory Segment

2000	Permanent file
42614	Radix–50 for KED
0	Radix–50 for spaces
73376	Radix–50 for SAV
72	72 octal blocks long
–	Used only for tentative files
22156	Created on 3–Sep–86
2000	Permanent file
50553	Radix–50 for MAC
71330	Radix–50 for RO
73376	Radix–50 for SAV
77	75 octal blocks long
–	Used only for tentative files
26657	Created on 13–Nov 87
2000	Permanent file
46166	Radix–50 for LIN
42300	Radix–50 for K
73376	Radix–50 for SAV
61	61 octal blocks long
–	Used only for tentative files
22156	Created on 3–Sep–86
2000	Permanent file
12625	Radix–50 for CRE
22600	Radix–50 for F
73376	Radix–50 for SAV
6	6 octal blocks long
–	Used only for tentative files
26657	Created on 13–Nov 87
1000	Empty area (never used since initialization)
000325	Radix–50 for EM (stored at initialization)
063471	Radix–50 for PTY (stored at initialization)
023364	Radix–50 for FIL (stored at initialization)
430	430 octal blocks long
–	Used only for tentative files
–	(the date is not significant)
4000	End–of–segment marker

To find the starting block of a particular file, first find the directory segment containing the entry for that file. Then, take the starting block number in the fifth word of that directory segment and add the length of each permanent, tentative, and empty entry in the directory before your file. For example, in Figure 1–8 the permanent file RT11XM.SYS begins at block number 51_8 on the volume; word 5 shows 16_8 and the previous entry, SWAP.SYS, is 33_8 blocks long.

1.1.3 File Storage on Random-Access Volumes

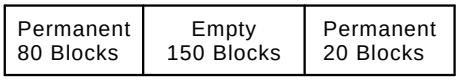
RT–11 uses the three types of directory entry mentioned previously to completely describe the contents of a random-access volume. All files reside on contiguous blocks. There are advantages and disadvantages to this method of storing data.

When data is stored in contiguous blocks, I/O is more efficient. Transfers to large buffers are handled directly by the hardware for certain disks; seeks between blocks and program interrupts between blocks are eliminated. File data is processed simply and efficiently since the data is not encumbered by link words in each block. Routines to maintain the directory are relatively small because the directory structure is simple. File operations, such as open, delete, and close, are performed quickly with few disk accesses, because only the directory must be accessed, and not additional bitmaps or retrieval pointers.

One disadvantage of this method of storing data is that a volume can become fragmented, requiring a squeeze operation to consolidate its free space. Another is that once a file is closed, a running program cannot easily increase its size. Only a small number of output files can be opened simultaneously, even on a large volume, unless the limits of the file sizes are known in advance. Finally, this scheme precludes the use of multiple and hierarchical directories.

Figure 1–9 shows a simplified diagram of a random-access volume that has a total of 250_8 blocks of space available for files after blocks 0 through 5 and the directory are accounted for. The volume in the figure has two permanent files and one empty area stored on it.

Figure 1–9: Random-Access Volume with Two Permanent Files

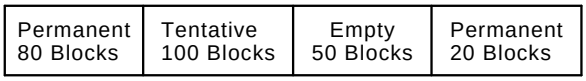


When you create a file, your program must specify a size for the file in the .ENTER programmed request. If you know the exact number of blocks required, you can use that number in the request. RT–11 will use a best-fit algorithm and attempt to put the file in a free space that exactly matches the size requested or, failing that, in the free space that most closely matches the size requested.

If you do not know the actual size, as is often the case, the space you specify should be large enough to accommodate all the data possible. Two special cases for the .ENTER programmed request let you do this easily. In the first case, a length argument of 0 allocates for the file either one-half the largest space available, or the second largest space, whichever is bigger; in the second case, a length argument of –1 allocates the largest space possible on the volume.

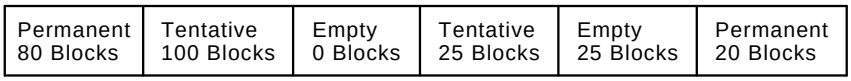
The monitor creates a tentative file on the volume with the length you specified. The tentative file must always be followed by an empty area to enable the system to recover unused space if less data is written to the file than you originally estimated. Figure 1–10 shows an example of a tentative file whose allocated size is 100_8 blocks. Note that the total amount of space on the volume, 250_8 blocks in this case, remains constant.

Figure 1–10: Random-Access Volume with One Tentative File



Suppose, for example, that while the file is being created by one program, another program enters a new file, allocating 25_8 blocks for it. The volume would appear as shown in Figure 1–11. Remember that every tentative file must be followed by an empty area.

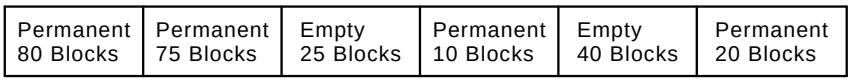
Figure 1–11: Random-Access Volume with Two Tentative Files



When a program finishes writing data to the volume, it closes the tentative file with the .CLOSE programmed request. RT–11 then makes the tentative file permanent. The length of the file is the actual size of the data that was written. The size of the empty area is its original size plus any unused space from the tentative file.

Figure 1–12 shows the same volume after both tentative files are closed. The first file's actual length is 75_8 blocks, and the second file's length is 10_8 blocks.

Figure 1–12: Random-Access Volume with Four Permanent Files



Because of this method of storing files, it is impossible in RT–11 to extend the size of an existing file from within a running program. To make an existing file appear to be bigger, you can read the existing file; allocate a new, larger tentative file; and write both the old and the new data to the new file. You can then delete the old file.

The CREATE/EXTENSION command (DUP /T option) is an easy way to extend the size of an existing file. However, to use this option, you must have an empty file with sufficient space in it immediately following the data file.

1.1.4 Size and Number of Files

The number of files you can store on an RT–11 volume depends on the number of segments in the volume's directory and the number of extra words per entry. If you use no extra words, each segment can contain 72_{10} entries. However, the maximum number of entries includes three that are reserved by the operating system and not available to the user. These three entries in each segment are for end-of-segment, empty (but reserved), and a reserved entry for use when creating a tentative file. Therefore, the maximum number of usable entries per directory segment is 69_{10}.

The maximum number of directory segments on any RT–11 volume is 31_{10}. Use the following formula to calculate the theoretical maximum number of directory entries, and thus, the maximum number of files.

```
     512 - 5
31 * -------  - 3
      7 + N
```

N represents the number of extra information words per directory entry. If N is 0, the maximum number of files you can store on the volume is 2242_{10}.

Note that all divisions are integer and the remainder should be discarded.

If you store files sequentially (that is, one immediately after another) without deleting any files, roughly one-half of the theoretical maximum number of files will fit on the volume before a directory overflow occurs. This situation results from the way RT–11 handles filled directory segments.

When a directory segment becomes full and it is necessary to open a new segment, the monitor moves approximately one-half of the directory entries of the filled segment to the new segment. Thus, when the final segment is full, all previous segments have approximately one-half of their total capacity. See Section 1.1.5 for a detailed explanation of how RT–11 splits a directory segment.

If you add files continually to a volume without issuing the SQUEEZE monitor command, you can use the following formula to compute the maximum number of entries, and thus, the maximum number of files.

```
          S
(M - 1) * - + S
          2
```

M represents the maximum number of segments.

S can be computed from the following formula:

```
    512 - 5
S = ------- - 3
     7 + N
```

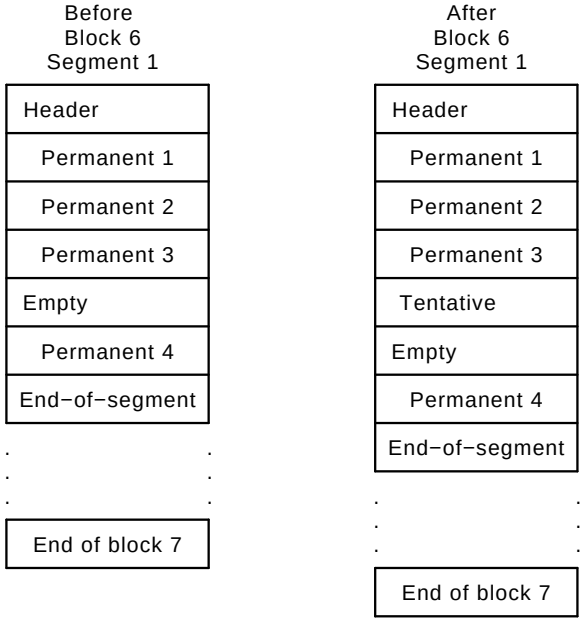
N represents the number of extra information words per entry.

You can realize the theoretical total of directory entries (see the first formula, above) by compressing the volume (using the DUP /S option or the monitor SQUEEZE command) when the directory fills up. DUP packs the directory segments as well as the physical volume.

1.1.5 Splitting a Directory Segment

Whenever RT–11 stores a new file on a volume, it searches through the directory for an empty area that is large enough to accommodate the new tentative file. When it finds a suitable empty area, it creates the new file as a tentative file followed by an empty area, sliding the rest of the directory entries down to make room for the new entry. Figure 1–13 shows how RT–11 stores a new file as a tentative file followed by an empty area.

Figure 1–13: Storing a New File

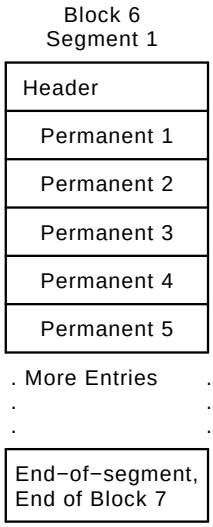


This procedure works properly as long as the empty entry and the entries following it can move downward. However, if the segment is full, the monitor must split the segment, if possible, to store the new entry. Figure 1–14 illustrates a directory segment that is full.

First, the monitor checks the header for the number of segments available. If there are none, a *directory full* error results and the monitor cannot store the new file. You can squeeze the volume at this point to pack the directory segments and try the operation again.

If there is another directory segment available, the monitor divides the current segment by first finding a permanent or tentative entry near the middle of the segment and saving its first word. In place of the first word, the monitor puts an end-of-segment marker. It then saves the current link information, links the current

Figure 1–14: Full Directory Segment



segment to the next available segment, and writes the current segment back to the volume.

Next, the monitor restores the first word of the middle entry to the copy of the segment that is still in memory, and restores the link information. It slides the middle entry and all the entries following it to the top of the segment. Then, the monitor writes this segment to the volume as the next available segment. Finally, the monitor reads segment 1 into memory and updates the information in its header. At this point, control passes to the top of the .ENTER routine, and the monitor begins its search again for a suitable empty entry to accommodate the new file.

Figures 1–15 and 1–16 summarize the process of splitting a directory segment. In this example, segment 1 was the only segment in use. It contained an empty entry but did not have room for a tentative entry in addition to the empty one. After the split, segments 1 and 2 are both about half full.

After a directory segment splits, the monitor can store the new file in either the new segment or the old one, depending on which segment now contains the empty area. In Figure 1–16, the empty area is in segment 2.

Thus far, the link words seem superfluous since the segments are always in numerical order. However, consider a situation in which four segments are available: segment 1 fills and overflows into segment 2; segment 2 fills and overflows into segment 3; segments 1, 2, and 3 are half full, and they are linked in the order in which they are located on the volume (blocks 6, 10, and 12). The picture changes if you delete a large file from segment 2, leaving a large empty entry, and add a lot of

Figure 1–15: Directory Before Splitting

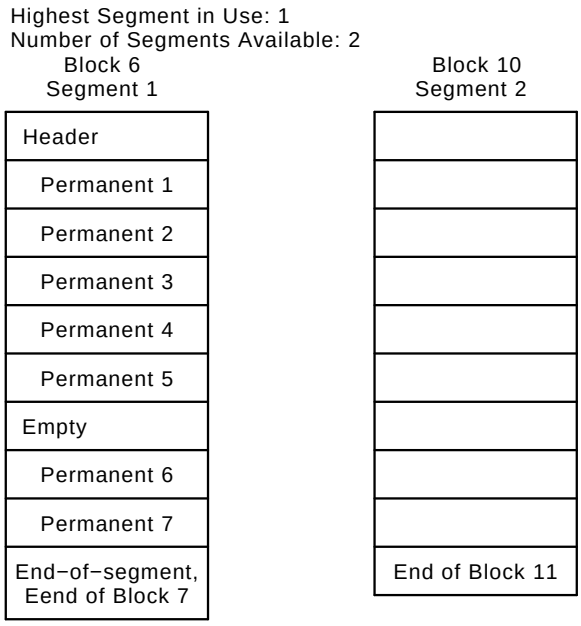
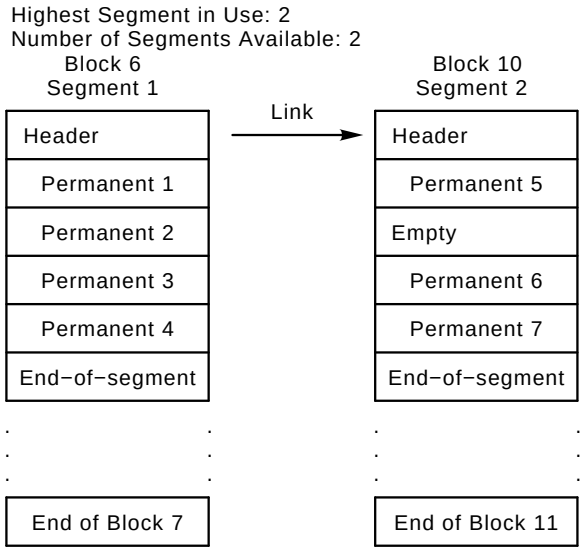
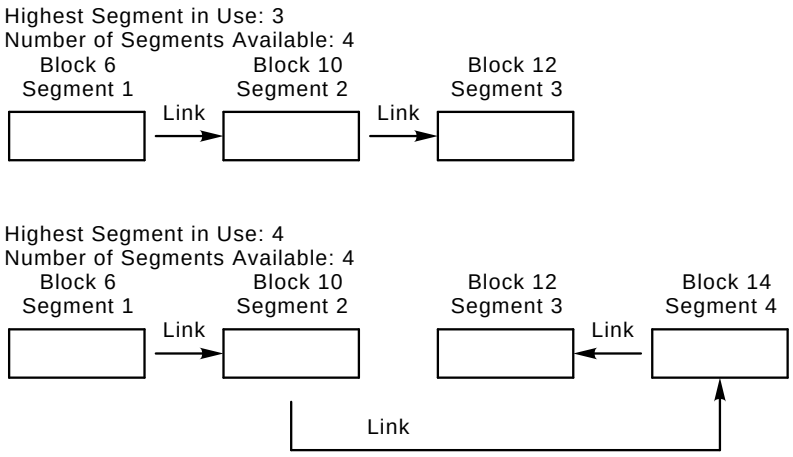


Figure 1–16: Directory After Splitting



small files to the volume. Segment 2 now fills up and overflows into the next free segment, segment 4, so that the links become visibly significant: segment 1 links to 2, segment 2 links to 4, and segment 4 links to 3 because segment 2 previously linked to 3. Figure 1–17 illustrates this example.

Figure 1–17: Directory Links



The process of splitting a directory segment in half when it is full may seem unnecessarily complicated. In fact, if many files are simply copied to a disk, a *directory full* error will occur when the directory is only half full. A squeeze will be required to consolidate the half-full directory segments before more files can be copied to the disk. However, splitting a directory segment reduces the amount of directory entry shuffling that occurs as files are added and deleted on a volume, thereby improving overall efficiency.

Refer back to Figure 1–15. Assume the empty entry between PERMANENT-5 and PERMANENT-6 represents 100_8 blocks of free disk space and that directory splitting is not done. If a program makes a directory entry for a new 25_8-block file, the directory entry for the file PERMANENT-7 must be moved to directory segment 2 to make room in segment 1 for the entry for PERMANENT-5A and an empty entry of 75_8 blocks. If the program makes another directory entry for another 25_8-block file, the file PERMANENT-6 must be moved to segment 2 to make room for the entry PERMANENT-5B and an empty entry of 50_8 blocks. In other words, each time a new directory entry occurs in segment 1, segment 2 must be updated as well, requiring an extra disk read and write.

If a directory segment is split when full, however, this problem does not occur as readily. Consider Figure 1–16 and observe what has happened. When a program creates the new 25_8-block file PERMANENT-5A, only directory segment 2 must be updated. No directory shuffling is required. If the file PERMANENT-4 is deleted and replaced with two or more files, directory segment 1 has room to accommodate the new file entries. Because directory splitting moves several directory entries from one segment to another at one time, any given directory operation is far less likely to require access to more than one directory segment. Directory splitting reduces dramatically the number of disk accesses required, on average, and improves overall directory efficiency.

1.1.6 How to Recover Data When the Directory Is Corrupted

One of the most frustrating experiences a programmer can have is to lose data on a volume because a block in the volume directory becomes bad or because another user writes over the directory. Usually, the files on the volume are intact, but the directory entries for some of the files have been destroyed. This section presents some guidelines you can follow to recover as much data as possible from a volume with a corrupted directory.

1.1.6.1 Examine Segment 1

First, determine whether segment 1 of the directory is bad. Remember, segment 1 occupies physical blocks 6 and 7 of the volume. To examine segment 1, mount the volume and issue the DIRECTORY command.

If you get an immediate *?MON-F-Directory I/O error* or *?MON-F-Dir I/O err* message, you know that segment 1 is bad. This leaves you with two alternatives: you can reformat and reinitialize the volume (the volume is reusable if a bad block scan shows no bad blocks in the directory area); or, if you are desperate to recover the data on the volume, you can open the volume in non-file-structured mode with TECO and search for data that resembles source code or other ASCII information that looks familiar. This kind of search is a tedious process; you probably shouldn't even consider it unless you have a video terminal to use with TECO. See Section 1.1.6.4 for information on removing a file from the volume.

If the DIRECTORY monitor command gives you at least a partial directory listing, you will be able to recover some of the information from the volume by issuing the DIRECTORY/SUMMARY monitor command. The /SUMMARY option lists information up to but not including the bad segment. To recover as many files as possible, you must repair the directory by linking around the bad segment.

1.1.6.2 Follow the Chain of Segments

Use SIPP to open the volume in non-file-structured mode. Look first at location 6000_8, which is the start of the header for directory segment 1. It contains the total number of segments available. Location 6002_8 contains the number of the next segment, and location 6004_8 shows the highest segment in use. (To review the directory header words and their meanings, see Table 1–2.)

To find the absolute location of the next segment, multiply the link word by 2000_8 and add 4000_8. For example, if the link word is 2, the next segment starts at location 10000_8 on the volume. Chain your way through the segments by opening the next segment and following its link word. As you go, make a worksheet for the link information, according to the format shown in Figure 1–18. Continue chaining until you have accounted for all the segments. Remember that segment 1 is always the first segment—that is, nothing links to it. The last segment has does not link and contains 0.

In Figure 1–18, segment 3 must link to 0—that is, it is the last segment since all the others have been accounted for already. To repair this directory, modify the header

Figure 1–18: Worksheet for a Directory Chain with Four Segments

Highest segment in use: 4
Number of segments available: 4

Segment:	Linked to:
1	2
2	4
4	3(bad)
3(bad)	?

Figure 1–19: Worksheet for a Directory Chain with Nine Segments

Highest segment in use: 9
Number of segments available: 9

Segment:	Linked to:
1	2
2	5
5	4
4	8
8	7(bad)
7(bad)	?
3	0
6	9
9	3

of segment 4 so that the link word contains 0 instead of 3. This eliminates segment 3 from the chain. Section 1.1.6.3 describes how to remove the files from the volume.

Figure 1–19 shows a more complicated example. In this case the bad segment is not the last one in the directory.

In a situation of the kind shown in Figure 1–19, you can follow the chain from segment 1 through segment 8, which points to the bad segment. To continue from that point, enter in the left column the lowest segment number not yet accounted

for and follow its link. Remember, if a segment links to 0, it is the last segment in the directory. Continue until all the segments are accounted for in the left column.

When you finish, the number that is missing from the right column is the segment to which the bad segment links. In Figure 1–19, this is segment 6. As in the previous example, use SIPP to link around the bad segment. In this case, change the link word in segment 8 to point to segment 6, thus removing segment 7 from the chain.

1.1.6.3 Remove the Data from the Good Segments

Once you have eliminated the bad segment by linking around it, you are ready to save the files whose entries appear in the good segments. Use the monitor COPY command to copy the files to a good volume. The following command, for example, copies all the files from one diskette to another:

```
COPY DU2:*.* DU3: RET
```

This procedure removes all the files from the volume except those whose entries appear in the bad segment.

1.1.6.4 Remove the Data from the Bad Segment

You can sometimes save files whose entries appear in the bad segment by using SIPP and DUP. If, when you open the segment with SIPP, block 1 of the segment is unreadable, you should probably stop because chances are that even if block 2 of the segment is readable, it contains old data that is not valid.

If you can read block 1, decode the header and the entries according to the diagram in Figure 1–4. Continuing with SIPP, try to locate the files on the volume. (Section 1.1.2.4 explains how to locate a file on the volume.) Once you establish the starting and ending blocks of a specific file, run DUP and use the following command sequence to transfer the file to a new volume:

```
output-filespec=input-volume:/G:startblock/E:endblock/I/F
```

You can use the following keyboard monitor command to achieve the same results:

```
COPY/DEVICE/FILES input-volume/START:startblock/END:endblock output-filespec
```

When you have finished removing files from the volume, you can return it to another user (if someone wrote over the directory); or, you can reformat and initialize it (if there was a bad block in the directory area). If reformatting does not remove the bad block, label the volume clearly so you do not accidentally use it again.

1.1.7 BUP Volumes

BUP-initialized disks are nonbootable RT–11 file-structured volumes with one directory segment and a special BUP identifier in the home block (see Section 1.1.1). Structurally, they are essentially identical to other RT–11 volumes. BUP, during restoration operations, checks for the special BUP identifier in the home block as part of its verification procedure to ensure that savesets consisting of multiple file sections are of the proper length and in the proper sequence.

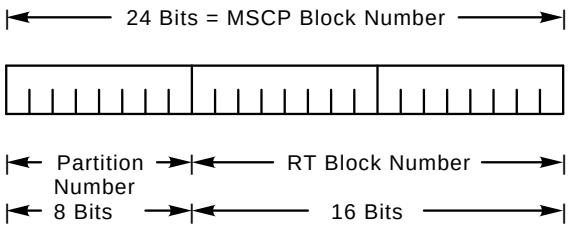
Section 2.11 describes the format of BUP savesets and file sections.

1.1.8 Large (Partitioned) Disks

RT–11's directory structure allows a block number up to 16 bits ($65{,}535_{10}$) long. Many of today's disks, however, have more than 65,535 blocks. To utilize the full capacity of these disks, RT–11 does **disk partitioning**.

Disk partition numbers allow RT–11 to use disks having more than 65,535 blocks. The disk partition number can be thought of as a high-order block number, as shown in Figure 1–20.

Figure 1–20: MSCP Disk Block Number



If a disk has more than 65,535 blocks, the RT–11 device handler for that disk divides the disk into logical partitions of 65,535 [1] blocks each. An RT–11 disk handler for a partitioned disk can, in theory, support up to 256 disk partitions. Therefore, the largest disk RT–11 can access has 256*65,535 blocks. To an RT–11 user, such a disk would appear to be 256 separate 65,535-block disks, each disk having its own directory.

Even though full MSCP supports block numbers of up to 32 bits, current RT–11 MSCP handlers for partitioned disks (for example, DU) support a block number of no more than 24 bits, because the handlers store the partition numbers as bytes. However, the partition number entries in the handler's translation table could be expanded to word entries if desired and 32-bit block numbers supported with no particular difficulty. Refer to *RT–11 System Internals Manual* for details of the format of the DU handler's translation table.

[1] Although RT–11 block numbers can be 0 through 177777_8, or a total of $65{,}536_{10}$ blocks (200000_8, or 000000 in 16 bits since the 17th bit is lost), the size of a partition is defined as $65{,}535_{10}$ blocks (177777_8), with RT–11 block numbers 0 through 177776. This avoids the problem of 16-bit overflow when dealing with the partition size. Because the partition number is added onto the left of the RT–11 block number to give the MSCP block number, one block between each partition is unused. The list below gives the block numbers of the first three partitions:

Partition	Block Numbers
0	000000–177776, block 177777 unused
1	200000–377776, block 377777 unused
2	400000–577776, block 577777 unused

1.1.9 Interchange Diskette Format

You can use the FILEX /U option (or the monitor COPY/INTERCHANGE command) to transfer data between RT–11 volumes and interchange diskettes. An interchange diskette, also known as an IBM floppy disk, consists of 77_{10} tracks. Each track contains 26_{10} sectors, and you can store one record of 128 or fewer characters per sector, using EBCDIC format.

Track 0 of the diskette is reserved for dataset labels, which are a form of directory. The functions of the sectors of track 0 are as follows.

Sectors 1 through 4 are reserved by IBM for system use. There are 80 blanks per sector.

Positions 1 through 13_{10} of sector 5 are used to record the identity of an error track. Positions 1 through 5 contain **ERMAP** to identify the sector as an error map. Sector 5 is not supported by RT–11.

Sector 6 is reserved by IBM for system use. It contains 80 blanks.

Sector 7 is the volume label. Positions 1 through 4 (bytes 0 through 3) contain **VOL1** in EBCDIC. This identifies the diskette as an IBM floppy. Within Digital, these four bytes identify the system that wrote the diskette. RT–11 stores **RT11** here. Other fields in sector 7 identify the diskette, its format, and its owner, and indicate whether or not the diskette uses standard labels. Table 1–4 describes the contents of sector 7.

Table 1–4: Interchange Diskette Sector 7

Octal	Decimal	Byte Contents
0	1	V
1	2	O
2	3	L
3	4	1
4	5	Bytes 5 through 10 contain the volume ID field. The ID consists of one to six digits or letters (left-justified); unused positions must contain blanks.
12	11	Access code. Must be blank to permit access to diskette.
13	12	Bytes 12 through 37 are reserved by IBM.
45	38	Bytes 38 through 51 contain the owner ID field. Not all systems use this field.
63	52	Bytes 52 through 76 are reserved by IBM.

Table 1–4 (Cont.): Interchange Diskette Sector 7

Octal	Decimal	Byte Contents
114	77	Bytes 77 and 78 are the record sequence field, or interleave factor.
115	78	Two blanks represent 1:1 interleave.
116	79	Reserved by IBM.
117	80	Label version field. **W** indicates standard labels.

Sectors 8 through 26 are the dataset labels. They are 40 words long and contain directory information. Table 1–5 describes the contents of sectors 8 through 26.

Table 1–5: Interchange Diskette Sectors 8 Through 26

Octal	Decimal	Byte Contents
0	1	H
1	2	D
2	3	R
3	4	1
4	5	Reserved.
5	6	Bytes 6 through 13 contain the user name for the dataset (the file label).
15	14	Bytes 14 through 22 are reserved.
26	23	Bytes 23 through 27 contain the block/record length. The default is 80, but 128 is possible.
33	28	Reserved.
34	29	Bytes 29 and 30 contain two EBCDIC characters representing the track number of the beginning of data.
36	31	EBCDIC 0 (360_8).
37	32	Bytes 32 and 33 contain two EBCDIC characters representing the sector number of the beginning of data.
41	34	Reserved.
42	35	Bytes 35 and 36 contain two EBCDIC characters representing the number of the last track reserved for this dataset.
44	37	EBCDIC 0 (360_8).
45	38	Bytes 38 and 39 contain two EBCDIC characters representing the number of the last sector reserved for this dataset.
47	40	Reserved.
50	41	Bypass indicator.

Table 1–5 (Cont.): Interchange Diskette Sectors 8 Through 26

Octal	Decimal	Byte Contents
51	42	Dataset security.
52	43	Write protect.
53	44	Blank for data interchange (100_8).
54	45	Multi-volume indicator: C=Continued, L=Last, Blank=Not continued.
55	46	Bytes 46 and 47 contain the volume sequence number.
57	48	Bytes 48 and 49 contain the creation year (such as **80**).
61	50	Bytes 50 and 51 contain the creation month.
63	52	Bytes 52 and 53 contain the creation day.
65	54	Bytes 54 through 66 are reserved.
102	67	Bytes 67 through 72 contain the expiration date (in the same format as the creation date).
110	73	Verify mark: V=Dataset verified, Blank=Not verified.
111	74	Reserved.
112	75	Bytes 75 through 79 contain the number of the next unused track and sector within this dataset.
113	76	Bytes 75 and 76 contain the track number.
114	77	EBCDIC 0 (360_8).
115	78	Bytes 78 and 79 contain the number of the next unused sector.
117	80	Reserved.

1.2 Sequential-Access Volumes

RT–11 magtape volumes are file-structured and sequentially accessed (not random-accessed). This section describes the magtape formats and file structures. At the end of this section is a procedure you can use to copy magtapes between the RT–11 and VMS operating systems.

1.2.1 Magtape Structure

RT–11 supports two types of magtape formats. They are quite similar to each other and are both based on ANSI Standard X3.27–1978, *File Structure and Labeling of Magnetic Tapes for Information Interchange*.

If you read or write files on a magtape using an RT–11 utility such as PIP, or from your own assembly-language program using .LOOKUP, .ENTER, .READ, and .WRITE, you use the file-structured magtape handler. File-structured magtape handlers contain the FSM, the file structure module. RT–11's file-structured magtape handler creates and recognizes tape labels on magtapes that it writes.

BUP, the backup utility program, bypasses the FSM in magtape handlers and uses its own file structure module. Therefore, if you write files on a magtape using BUP, the resulting tape will be somewhat different from one created using the file-structured magtape handler.

The following sections describe each tape format in detail.

1.2.1.1 RT–11 File-Structured Magtape Format

This section describes the magtape format read and written by the file-structured magtape handler.

RT–11 file-structured magtape implementation includes the following restrictions. Features and restrictions for BUP-written magtapes are described in the *RT–11 System Utilities Manual*:

- There is no EOV (end-of-volume) support. This means that no file can continue from the end of one tape volume to another volume.
- RT–11 does not ignore noise blocks on input.
- RT–11 assumes that data is written in records of 512_{10} characters per block. The logical record size equals the physical record size.

NOTE

The hardware magtape handler (as opposed to the file-structured magtape handler) can read data in any format. You can also make use of .SPFUN programmed requests and the file-structured magtape handler to read tapes with data in a nonstandard format (see the *RT–11 Device Handlers Manual* for details). The RT–11 utility programs, such as PIP, DUP, and DIR, can only read and write tapes in the standard RT–11 format of 512_{10}-character blocks.

- RT–11 does not check access fields and therefore provides no volume protection.

Figure 1–21 is a dump of tape written by the file-structured magtape handler (FSM). The VOL1 block, however, is actually written by DUP (not by FSM) when you INIT the magtape. Prior to RT–11 Version 5.5, DUP and FSM always made label records of 512 bytes, but only the first 80 bytes were used. Characters in positions 81–512 were undefined. Magtape labels created in that format were not transportable to other operating systems, such as VMS.

Since V5.5, those label blocks are physically 80 characters (40 words) long, making them transportable to a VAX processor running the VMS operating system. The procedure to transport magtapes created by RT–11 to VMS is described in Section 1.2.2.

If necessary, a customization patch for DUP.SAV allows you to make old-style 256-word labels. If you use the patch, bytes 81–512 in VOL1 records (written by DUP) are initialized to spaces. Bytes 81–512 in HDR1 and EOF1 labels (written by FSM),

however, are still undefined. Magtapes containing such labels are not transportable to VMS.

Figure 1–21: FSM Magtape Label and Header Formats

```
*** MTFMGR: DB MU0:,  256. Words, (Base  8) ***
047526 030514 052122 030461 020101 020040 020040 020040  *VOL1RT11A       *
020040 020040 020040 020040 020040 020040 020040 020040  *                *
020040 020040 042040 041045 020040 020040 020040 020040  *     D%BHAMILTON*
020040 020061 020040 020040 020040 020040 020040 020040  *  1             *
020040 020040 020040 020040 020040 020040 020040 031440  *              3*

*** MTFMGR: DB MU0:,  256. Words, (Base  8) ***
042110 030522 042532 047522 042105 055056 055132 020040  *HDR1FILE.DAT  *
020040 020040 051040 030524 040461 030040 030060 030061  *     RT11A 00010*
030060 030060 030060 030061 020060 030060 030060 020060  *001000100 00000 *
030060 030060 020060 030060 030060 030060 042504 051103  *00000 000000DECR*
030524 040461 020040 020040 020040 020040 020040 020040  *T11A            *

** Tape Mark **

DATA AREA - 256-word fixed-length blocks.  These blocks are exactly
the same as the disk files that they represent.

** Tape Mark **

*** MTFMGR: DB MU0:,  256. Words, (Base  8) ***
047505 030506 042532 047522 042105 055056 055132 020040  *EOF1FILE.DAT  *
020040 020040 051040 030524 040461 030040 030060 030061  *     RT11A 00010*
030060 030060 030060 030061 020060 030060 030060 020060  *001000100 00000 *
030060 030060 020060 030060 030060 030060 042504 051103  *00000 000007DECR*
030524 040461 020040 020040 020040 020040 020040 020040  *T11A            *

** Tape Mark **

** Tape Mark **

** Tape Mark **
```

Note that the 000007 in the EOF1 label in Figure 1–21 indicates that there are 7 data blocks in the data area.

In the following examples, an asterisk (*) represents a tape mark. The structure of the actual tape mark depends on the encoding scheme that the hardware uses. A typical nine-channel NRZ tape mark consists of one tape character (23_8) followed by seven blank spaces and an LRCC (23_8). Consult the hardware manual for your own tape volume if the format of the tape mark is important to you.

A file stored on magtape has the following format:

```
HDR1
 *
data
 *
EOF1
 *
```

A volume containing a single file has the following format:

```
VOL1
HDR1
 *
data
 *
EOF1
 *
 *
 *
```

A volume containing two files has the following format:

```
VOL1
HDR1
 *
data
 *
EOF1
 *
HDR1
 *
data
 *
EOF1
 *
 *
 *
```

A double tape mark following an EOF1 * label indicates logical end of tape. (Note that the EOF1 label is considered to consist of the actual EOF1 information plus a single tape mark.)

A magtape that has been initialized has the following format:

```
VOL1
HDR1
 *
 *
EOF1
 *
 *
 *
```

A bootable magtape is a multifile volume that has the following format:

```
VOL1
BOOT
HDR1
 *
data
 *
EOF1
 *
 *
 *
```

To create an RT–11 bootable magtape, you must copy the primary bootstrap by using the INITIALIZE/FILE:primboot command, where *primboot* is MBOOT for 8

bpi and TMSCP magtapes or MBOT16 for 16 bpi magtapes. The primary bootstrap is represented by **BOOT** in the format given for a bootable magtape. It occupies a 256_{10}-word physical block. The first real file on the tape must be the secondary bootstrap, the file MSBOOT.BOT. If the tape is designed to allow another user to create another bootable magtape, you should copy the primary bootstrap file (MBOOT.BOT or MBOT16.BOT) to the tape, as a file. (This is in addition to copying it into the boot block at the beginning of the tape.) More detailed instructions for building bootable magtapes are in the *RT–11 System Internals Manual*.

The primary and secondary bootstraps inspect the I/O page for the presence of standard magtape volume registers to determine which type of magtape controller is available. Make sure that other peripheral volumes on your system do not use addresses in the I/O page normally used by another magtape controller. The bootstraps might attempt to use the magtape controller assigned to those addresses rather than the magtape controller actually installed on your system.

Each label on the tape (as shown in the formats of the various magtape structures) is a separate record, 80 bytes long, and each byte in the label contains an ASCII character. (That is, if the content of a byte is listed as 1, the byte contains the ASCII code for 1, not the octal value 1.) Table 1–6 shows the contents of the 80 bytes in the three labels. Note that with the older version of RT–11, the VOL1, HDR1, and EOF1 labels occupied a full 256_{10}-word block each, of which only the first 80 bytes were meaningful.

Table 1–6: RT–11 File-Structured Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
Volume Header Label (VOL1)			
1–3	Label identifier	3	VOL
4	Label number	1	1
5–10	Volume identifier	6	Volume label. If you do not specify a volume ID at initialization time, the default is RT11A[SP]
11	Accessibility	1	[SP]
12–37	Reserved	26	(Spaces)
38–50	Owner identifier	13	Positions 38–40 = D%B (This means tape was written by DEC PDP–11.) Positions 41–50 = Owner name. Maximum is 10 characters; default is (spaces).
51	DEC standard version	1	1

Table 1–6 (Cont.): RT–11 File-Structured Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
Volume Header Label (VOL1)			
52–79	Reserved	28	(Spaces)
80	Label standard version	1	3
File Header Label (HDR1)			
1–3	Label identifier	3	HDR
4	Label number	1	1
5–21	File identifier	17	The six-character ASCII file name, dot, three-character file type. This field is left justified and followed by spaces. Do not pad the file name or type.
22–27	File set identifier	6	RT11A[SP]
28–31	File section number	4	0001
32–35	File sequence number	4	First file on tape has 0001. This value is incremented by 1 for each succeeding file. On a newly initialized tape, this value is 0000.
36–39	Generation number	4	0001
40–41	Generation version	2	00
42–47	Creation date	6	[SP] followed by (year*1000) + day in ASCII; [SP] followed by 00000 if no date. For example, 2/1/90 is stored as [SP]90032.
48–53	Expiration date	6	[SP] followed by 00000 indicates an expired file.
54	Accessibility	1	[SP]
55–60	Block count	6	000000
61–73	System code	13	DECRT11A[SP] followed by spaces.
74–80	Reserved	7	(Spaces)

Table 1–6 (Cont.): RT–11 File-Structured Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
First End-of-File Label (EOF1). This label is the same as the HDR1 label, with the following exceptions:			
1–3	Label identifier	3	EOF
55–60	Block count	6	Number of data blocks since the preceding HDR1 label, unless you issue an .SPFUN programmed request. If you issue .SPFUNs, the block count is 0. However, if the only special function operations you do are 256_{10}-word .SPFUN writes, the block count is accurate.

1.2.1.2 BUP Magtape Format

The magtape format used by BUP, the Backup Utility Program, is not quite the same as the RT–11 file-structured magtape format used by PIP, DUP, and so on. BUP bypasses the file-structured magtape handler and writes its own tape labels. You can mount a magtape written by BUP on a VAX or other system that supports ANSI-format magtapes, and read the BUP saveset from the tape. Note, however, that VMS will read the entire BUP saveset as a single file. You have to extract individual RT–11 files from the saveset file with the VMS EXCHANGE utility.

Figure 1–22 and Table 1–7 illustrate the format of labels on a BUP magtape. Some of the contents in these records is fixed, while other parts are variable. For example, DECRT11BUP is always recorded in a BUP tape VOL1 label, but BACKUP.BUP is an arbitrary saveset name given by the user. Similarly, 88159 in the HDR1 record is a date code indicating that the file was made on the 159th day of 1988.

Figure 1–22: BUP Magtape Label and Header Formats

```
VOLUME LABEL:

  047526 030514 052122 052502 020120 020040 020040 020040  *VOL1RTBUP       *
  020040 020040 020040 020040 042504 051103 030524 041061  *        DECRT11B*
  050125 033065 020060 020040 020040 020040 020040 020040  *UP560           *
  020040 020040 020040 020040 020040 020040 020040 020040  *                *
  020040 020040 020040 020040 020040 020040 020040 032040  *               4*

HEADER_1 LABEL:

  042110 030522 040502 045503 050125 041056 050125 020040  *HDR1BACKUP.BUP  *
  020040 020040 051040 041124 050125 030040 030060 030061  *     RTBUP 00010*
  030060 030061 030060 030061 020060 034070 032461 020071  *001000100 88159 *
  030060 030060 020060 030060 030060 030060 042504 051103  *00000 000000DECR*
  030524 041061 050125 033065 020060 020040 020040 020040  *T11BUP560       *

HEADER_2 LABEL:

  042110 031122 030106 030064 033071 030060 030465 020062  *HDR2F0409600512 *
  030060 032460 033470 031064 030060 030060 030060 030060  *0005874200000000*
  020040 020040 020040 020040 020040 020040 020040 020040  *                *
  020040 030060 020040 020040 020040 020040 020040 020040  *  00            *
  020040 020040 020040 020040 020040 020040 020040 020040  *                *

 ** Tape Mark **

  FILE SECTION DATA AREA CONTAINING CONSECUTIVE DISK BLOCKS, BLOCKING
  FACTOR=8 (PHYSICAL RECORD SIZE=2048 WORDS, EACH RECORD CONTAINS
  EIGHT 256-WORD BLOCKS)

 ** Tape Mark **

END_OF_FILE_1 (OR END_OF_VOLUME_2) LABEL:

  047505 030506 040502 045503 050125 041056 050125 020040  *EOF1BACKUP.BUP  *
  020040 020040 051040 041124 050125 030040 030060 030061  *     RTBUP 00010*
  030060 030061 030060 030061 020060 034070 032461 020071  *001000100 88159 *
  030060 030060 020060 030060 031467 031464 042504 051103  *00000 007343DECR*
  030524 041061 050125 033065 020060 020040 020040 020040  *T11BUP560       *

END_OF_FILE_2 (OR END_OF_VOLUME_2) LABEL:

  047505 031106 030106 030064 033071 030060 030465 020062  *EOF2F0409600512 *
  030060 032460 033470 031064 030060 032460 033470 031064  *0005874200058742*
  020040 020040 020040 020040 020040 020040 020040 020040  *                *
  020040 030060 020040 020040 020040 020040 020040 020040  *  00            *
  020040 020040 020040 020040 020040 020040 020040 020040  *                *

 ** Tape Mark **
[ ** Tape Mark ** ] if end of tape
[ ** Tape Mark ** ] "   "  "   "

OR HDR1, if another file section begins.
```

Table 1–7: BUP Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
Volume Header Label (VOL1)			
1–3	Label identifier	3	VOL
4	Label number	1	1
5–10	Volume identifier	6	Volume label. The label of the first volume of a BUP saveset is RTBUP; the label on the second volume is RTBU02, the third is RTBU03, etc.
11	Accessibility	1	SP
12–24	Reserved	13	(Spaces)
25–37	Implementation identifier	13	DECRT11BUPnnn, where nnn is a 3-digit version number. This identifier also appears in HDR1, EOF1, and EOV1 labels.
38–51	Owner identifier	14	(Spaces)
52–79	Reserved	28	(Spaces)
80	Label standard version	1	4
File Header Label (HDR1)			
1–3	Label identifier	3	HDR
4	Label number	1	1
5–21	File identifier	17	BACKUP.BUP plus seven spaces
22–27	File set identifier	6	RTBUP SP
28–31	File section number	4	0001
32–35	File sequence number	4	First file on tape has 0001. This value is incremented by 1 for each succeeding file. On a newly initialized tape, this value is 0000
36–39	Generation number	4	0001
40–41	Generation version	2	00

Table 1–7 (Cont.): BUP Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
File Header Label (HDR1)			
42–47	Creation date	6	SP followed by (year*1000) + number of the day in ASCII; SP followed by 00000 if no date. For example, SP88032 represents February 1, 1988.
48–53	Expiration date	6	SP followed by 00000 indicates no date.
54	Accessibility	1	SP
55–60	Block count	6	000000
61–73	Implementation identifier	13	DECRT11BUP560
74–80	Reserved	7	(Spaces)
Second File Header Label (HDR2)			
1–3	Label identifier	3	HDR
4	Label number	1	2
5	Record format	1	F (indicates fixed-length records)
6–10	Block length	5	04096
11–15	Record length	5	00512
16		1	Reserved
17–24	Saveset size in blocks	8	Decimal value
25–32	Next block number	8	Decimal value; in the final EOF2 label, this number equals the saveset size.
33–50	Reserved	17	Spaces
51–52	Offset length	2	00
53–80	Reserved	27	Spaces

Table 1–7 (Cont.): BUP Magtape Labels

Character Position in Label	Field Name	Length of Field	Contents
First End-of-File Label (EOF1). This label is the same as the HDR1 label, with the following exceptions:			
1–3	Label identifier	3	EOF
55–60	Block count	6	Number of data blocks since the preceding HDR1 label, unless you issue an .SPFUN programmed request. If you issue .SPFUNs, the block count is 0. However, if the only special function operations you do are 256_{10}-word .SPFUN writes, the block count is accurate.
Second End-of-File Label (EOF2)			
1–3	Label identifier	2	EOF
4	Label number	1	2
5	Record format	1	F
6–10	Block length	5	04096
11–15	Record length	5	00512
16	Reserved	1	SP
17–24	Saveset size	8	
25–32	Next block number	8	
33–50	Reserved	17	Spaces
51–52	Reserved	2	00
53–80	Reserved	27	Spaces

A backup saveset is recorded as one or more file sections. Only one file section for each tape volume is associated with a particular saveset.

EOV labels replace EOF labels when physical end-of-tape is encountered and a saveset continues on subsequent volume(s). In this case, the next volume always begins with the next saveset (file) section in sequence.

Label records are all 80 characters in length, both logically and physically.

VOL1 labels have the following characteristics:

- The first volume in a saveset contains a VOL1 label that identifies the tape with name RTBUP. Subsequent volumes are identified with names RTBU02, RTBU03, and so forth.
- The implementation identifier is DECRT11BUPxxx, where xxx is a 3-digit version number. This identifier is also recorded in the HDR1, EOF1 and EOV1 labels.

Notes about HDR1, EOF1 and EOV1 labels:

- Expiration date is not specified

Notes about HDR2, EOF2 and EOV2 labels:

- Character position 16 is blank and reserved for future use.
- Character positions 17–24 contain an 8-digit saveset size in blocks.
- Character positions 25–32 contain an 8-digit "next" block number. This number equals the saveset size on the final EOF2 label.

1.2.2 RT–11/VMS Magtape File Interchange

Because of changes that were made to how the FSM represents file names in the HDR1 and EOF1 label name fields, RT–11 magtapes are more compatible with the VMS operating system ANSI magtape implementation. The format change is backwards compatible with RT–11 utilities and, using the information provided below, allows text files to be transferred between the RT–11 and VMS operating systems.

Text files to be transferred between RT–11 and VMS must be enclosed in an RT–11 logical disk, as the logical disk file format is compatible with both operating systems and can be conveniently transferred in either direction by using magtape media.

The VMS EXCHANGE utility is used for format translation when files are transferred in either direction. Use EXCHANGE to create an RT–11 logical disk on a VAX computer when transferring files from the VMS system to RT–11. When transferring files from the RT–11 system to VMS, use EXCHANGE to convert files on the RT–11 logical disk from RT–11 format to VMS format. Because EXCHANGE utility characteristics could change with a future VMS release, you should consult the VMS documentation if any questions arise with using EXCHANGE.

The following example illustrates transferring files from a computer running RT–11 to a VAX computer running the VMS operating system:

1. Create and mount a logical disk of sufficient size to contain those files you want to transfer. Then, copy the files to the logical disk. In the following commands, *LOGDSK.DSK* is the logical disk, *LD0* is logical disk unit, and *dev:files.ext* are those files you want to transfer. You then copy the logical disk to an initialized magtape volume, *MU0*:

```
.CREATE LOGDSK.DSK/ALLOCATE:nnnnnn
.MOUNT LD0: LOGDSK.DSK
.INITIALIZE LD0:
.COPY dev:files.ext LD0:
.INITIALIZE MU0:
.COPY LOGDSK.DSK MU0:
```

As an alternative, you could use the following BACKUP command to achieve the same result:

```
.BACKUP/INITIALIZE/NOQUERY dev:files.ext MU0:LOGDSK.DSK
```

2. Carry the magtape volume to the computer running VMS, mount the volume, log on to the VMS system, and enter the following:

```
$ ALLOCATE MUA0:
```

If the files were placed in LD0 by the COPY command, issue the following command:

```
$ MOUNT/OVERRIDE=OWNER/BLOCKSIZE=512 MUA0: RT11A
```

If the files were placed in LD0 by the BACKUP command, issue the following command:

```
$ MOUNT/OVERRIDE=OWNER MUA0: RTBUP
```

3. After mounting MUA0, issue the following commands:

```
$ COPY MUA0:LOGDSK.DSK *
$ EXCHANGE
EXCHANGE> MOUNT/VIRTUAL LD: LOGDSK.DSK
EXCHANGE> DIR LD:
EXCHANGE> COPY LD: *.* *
EXCHANGE> CTRL/Z
$ DISMOUNT MUA0:
```

The following example illustrates transferring files from a VAX computer running the VMS operating system to a PDP–11 computer running RT–11:

1. Use EXCHANGE to convert and copy VMS format files (*vmsfiles.**) to an RT–11 format logical disk file (*LOGDSK.DSK*):

```
$ EXCHANGE
EXCHANGE> INITIALIZE/CREATE LOGDSK.DSK/ALLOC:nnnnnn
EXCHANGE> MOUNT/VIRTUAL LD: LOGDSK.DSK
EXCHANGE> COPY vmsfiles.* LD:
EXCHANGE> CTRL/Z
$
```

2. Initialize and mount a magtape (*MUA0*), and specify a 512-byte blocksize:

```
$ ALLOCATE MUA0:
$ INITIALIZE/OVERRIDE=OWNER MUA0: RTTAPE
$ MOUNT/BLOCKSIZE=512 MUA0: RTTAPE
```

3. Copy the logical disk file, *LOGDSK.DSK*, to the magtape and dismount the magtape:

```
$ COPY LOGDSK.DSK MUA0:
$ DISMOUNT MUA0:
```

4. Carry the magtape MUA0 to the RT–11 system and mount it. Transfer the logical disk file, *LOGDSK.DSK*, to a disk with sufficient space, and then perform regular RT–11 operations to retrieve the files it contains.

Chapter 2
File Formats

This chapter describes the formats of various RT–11 files. It contains information on the following file types:

- Logical disk files (.DSK)
- Object files (.OBJ)
- Symbol table definition files (.STB)
- Library files (.OBJ and .MLB)
- Absolute binary files (.LDA)
- Standard save image files (.SAV)
- Extended (I–D space) save image files (.SAV)
- Relocatable files (.REL)
- Stream ASCII files (such as .MAC, .FOR, and so on)
- CREF files
- BUP saveset files (.BUP)
- Error log files

2.1 Logical Disk File Format (.DSK)

A logical disk file looks exactly like a random-access volume image (Figure 1–1); block 0 of a logical disk file corresponds to block 0 of a random-access volume, block 1 of a logical disk file corresponds to the home block (block 1) of a random-access volume, and so on. If a logical disk contains a bootable device image, blocks 0 and 2 through 5 contain the primary and secondary bootstrap for LD as the system device.

A logical disk file contains a standard RT–11 directory, as described in Chapter 1, beginning in block 6 of the logical disk file. The directory entries point to files stored in the space between the end of the logical disk's last directory segment and the end of the logical disk file. All files stored in a logical disk file (.SAV, .OBJ, .LST, and so on) have the same format as they would if they were stored directly on a random-access volume.

In effect, a logical disk file is an image of a random-access volume; all descriptions of random-access volumes in Section 1.1 apply to logical disk files.

2.2 Object File Format (.OBJ)

An object module is a file containing a program or routine in a binary, relocatable form. Object files normally have a .OBJ file type. In a MACRO–11 program, one module is defined as the unit of code enclosed by the .TITLE and .END pair of directives. MACRO–11 takes the module name from the .TITLE statement. Language processors, such as MACRO–11 and FORTRAN IV, produce object modules; the Linker processes object modules to make runnable programs in .SAV, .LDA, or .REL format. The librarian can also process object files to produce library files, which the Linker can then use. Figure 2–1 illustrates object module processing.

Although you can combine many different object modules to form one file, each object module remains complete and independent. However, when the librarian combines object modules into a library, the modules are no longer independent. Instead, they are concatenated and become part of the library's structure.

The librarian concatenates modules by byte rather than by word in order to save space. For example, suppose a library is to consist of two modules and the first module contains an odd number of bytes. The librarian adds the second module to the library behind the first module and positions the first byte of the second module as the high-order byte of the last word of the first module. As a result of this procedure, one byte is saved in the library.

To understand byte concatenation, think of the modules as a stream of bytes, rather than as a stream of two-byte words. Figure 2–2 shows how two five-byte modules would be concatenated. Module 1 and module 2 are shown both as bytes and as words.

Figure 2–1: Object Module Processing

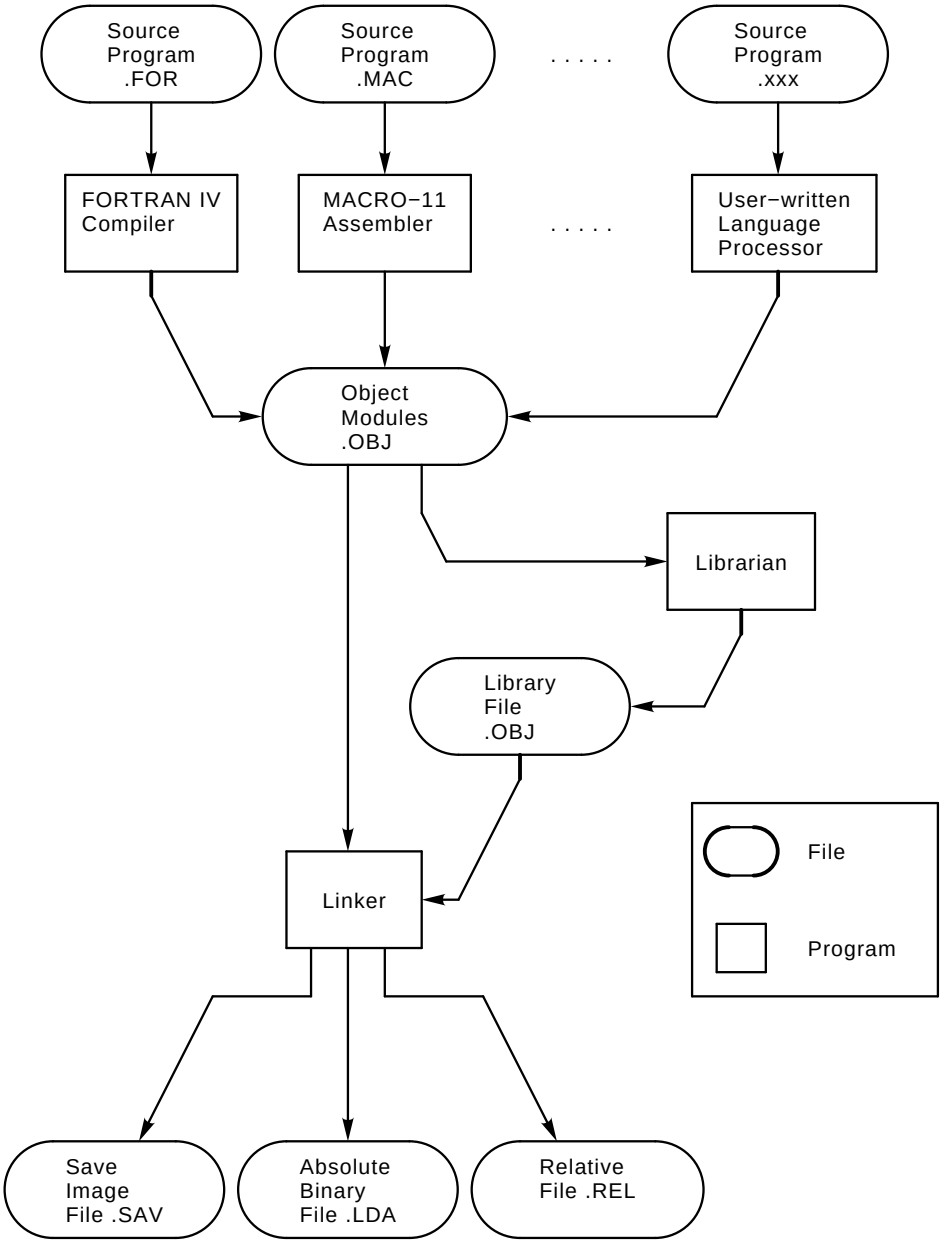
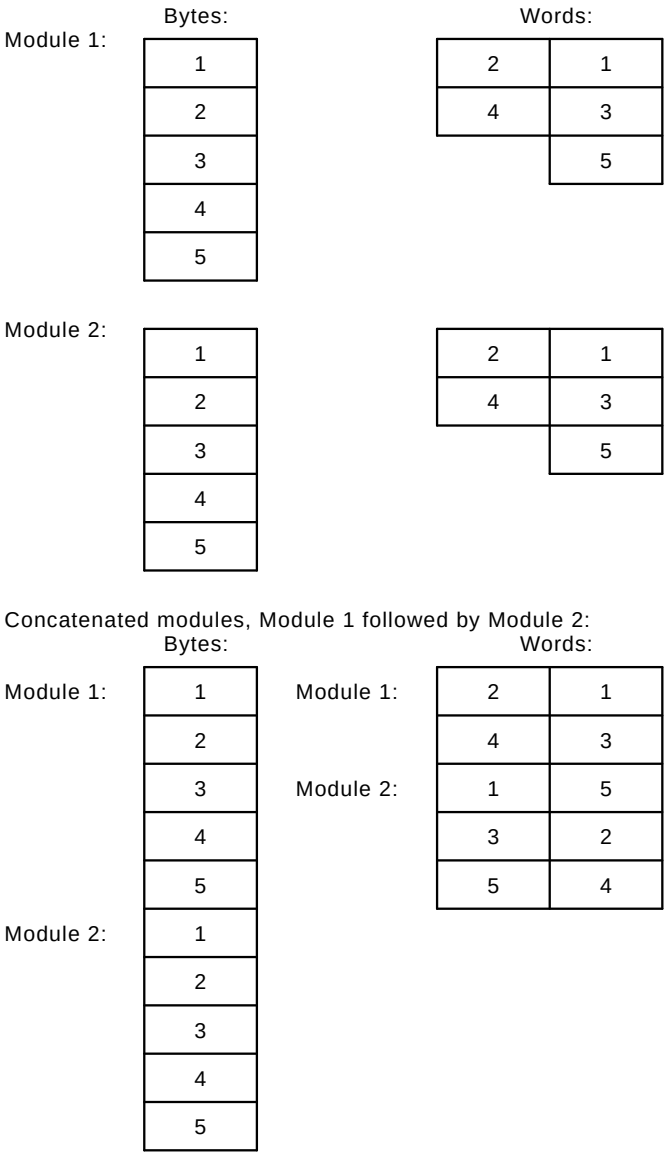


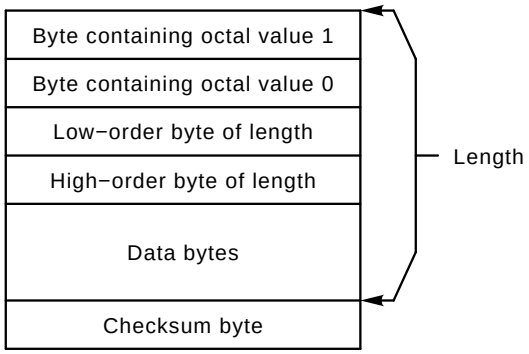
Figure 2–2: Modules Concatenated by Byte



The rest of Section 2.2 contains information on the composition of object modules that is more detailed than most programmers require. However, if you intend to write a language processor, a Linker program, or a program to dump and interpret object modules, you should read this material carefully. If you are writing a language processor and want its output to be processed by the RT–11 Linker, be sure that the processor produces object modules compatible with those described here. Since this section documents the object modules produced by MACRO–11 and FORTRAN IV, you could also use this information to write your own Linker program.

Object modules are made up of formatted binary blocks. A formatted binary block is a sequence of 8-bit bytes (stored in an RT–11 file or by some other means) that is arranged as shown in Figure 2–3.

Figure 2–3: Formatted Binary Format



Each formatted binary block has its length stored within it. The length includes all bytes of the block except the checksum byte. The checksum byte is the negative of the sum of all preceding bytes. Formatted binary blocks may be separated by a variable number of null (0) bytes.

The "data bytes" portion of each formatted binary block contains the actual object module information. RT–11 uses and recognizes eight types of data blocks. The information in these blocks guides the Linker as it translates object code into a runnable program. Table 2–1 lists the eight types of data blocks.

Table 2–1: RT–11 Data Blocks

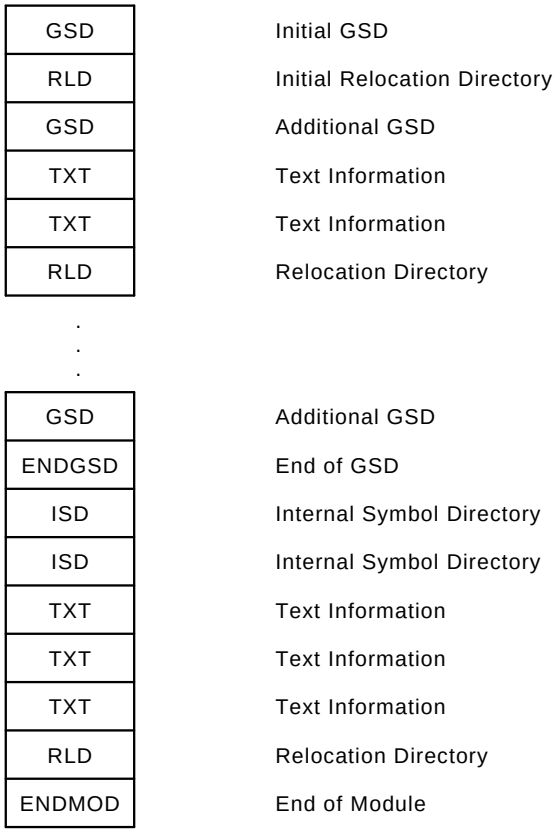
Identification Code	Block Type	Function
1	GSD	Holds the Global Symbol Directory information.
2	ENDGSD	Signals the end of Global Symbol Directory blocks in a module.
3	TXT	Holds the actual binary text of the program.
4	RLD	Holds Relocation Directory information.
5	ISD	Holds the Internal Symbol Directory information (not supported by RT–11).
6	ENDMOD	Signals the end of the object module.
7	Librarian header	Holds the status of the library file (see Section 2.4.1).

Table 2–1 (Cont.): RT–11 Data Blocks

Identification Code	Block Type	Function
10	Librarian end	Signals the end of the library file (see Section 2.4.3).

An object module must begin with a Global Symbol Directory (GSD) block and end with an End of Module (ENDMOD) block. Additional GSD blocks can occur anywhere in the file, but must appear before an End of Global Symbol Directory (ENDGSD) block. An ENDGSD block must appear before the ENDMOD block, and at least one Relocation Directory (RLD) block must appear before the first Text Information (TXT) block. Additional RLD and TXT blocks can appear anywhere in the file. The Internal Symbol Directory (ISD) block can appear anywhere in the file between the initial GSD and ENDMOD blocks. Figure 2–4 shows a general scheme for an object module.

Figure 2–4: General Object Module Format



You must declare all program sections (p-sects, v-sects, and c-sects) defined in a module in GSD items. The size word of each program section definition should contain the size in bytes to be reserved for the section. If you declare a program section more than once in a single object module, the Linker uses the largest declared size for that section. All global symbols that are defined in a given program section must appear in the GSD items immediately following the definition item of that program section.

A special program section, called the absolute section . ABS., is allocated by the Linker, beginning at location 0 of memory. Immediately after the GSD item that defines the absolute section, declare all global symbols that contain absolute (non-relocatable) values. If you do not want to allocate any memory space for the absolute section, specify zero as its size word. You can do this even if absolute global symbol definitions occur after it.

You must declare in GSD items those global symbols that are referenced but not defined in the current object module. These global references may appear in any GSD item except the very first, which contains the module name. In MACRO, referenced globals are listed in a GSD block under the . ABS. p-sect. They always have the p-sect definition preceding them.

Note that when a 16-bit word is stored as part of the information in a data block, it is always stored as two consecutive 8-bit bytes, with the low-order byte first.

Object module data blocks vary in length. The first byte in a data block is a code that identifies the type of data block. The codes range from 0 through 10_8, as Table 2–1 shows. The format of the rest of the information in the data block depends on the type of data block.

The following sections describe in detail the format of the data blocks.

2.2.1 Global Symbol Directory Block (GSD)

Global Symbol Directory blocks contain all the information the Linker needs to assign addresses to global symbols and to allocate the memory a job requires. Table 2–2 shows the eight types of entries that GSD blocks can contain.

Table 2–2: Entries in GSD Blocks

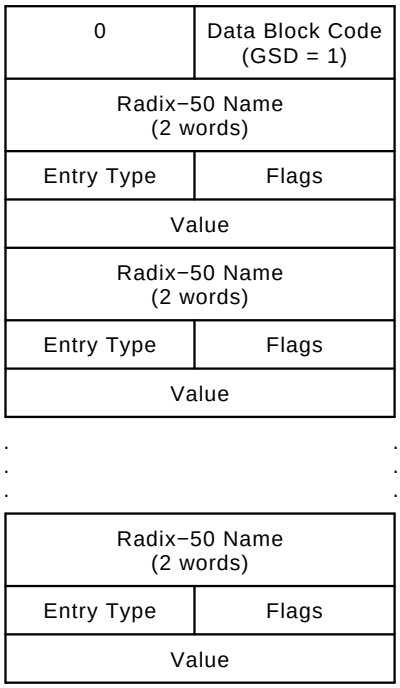
Entry Type	Description
0	Module Name
1	Control Section Name (c-sect)
2	Internal Symbol Name
3	Transfer Address
4	Global Symbol Name
5	Program Section Name (p-sect)

Table 2–2 (Cont.): Entries in GSD Blocks

Entry Type	Description
6	Program Version Identification (IDENT)
7	Mapped Array Declaration (v-sect)

Each entry type is represented by four words in the GSD data block. The first two words contain six Radix–50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. Figure 2–5 illustrates the format of the GSD data block and the entry types.

Figure 2–5: Global Symbol Directory Data Block

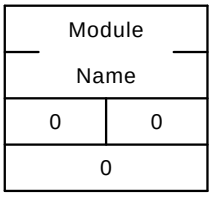


The following sections describe the entry types for GSD data blocks.

2.2.1.1 Module Name (Entry Type 0)

The module name entry, illustrated in Figure 2–6, declares the name of the object module. The name need not be unique with respect to other object modules because modules are identified by file, not module name. However, only one module name declaration can occur in a single object module.

Figure 2–6: Module Name Entry Format (Entry Type 0)



2.2.1.2 Control Section Name (Entry Type 1)

The control section name entry (see Figure 2–7) declares the name of a control section. The Linker converts control sections—which include a-sects, blank c-sects, and named c-sects—to p-sects. For compatibility with other systems, the Linker processes a-sects and both forms of c-sects. See Section 2.2.1.6 for the entry the Linker generates for a .PSECT statement.

You can define .ASECT and .CSECT statements in terms of .PSECT statements, as follows:

For a blank c-sect, define a p-sect with the following attributes:

```
.PSECT     ,RW,I,LCL,REL,CON
```

For a named c-sect, the p-sect definition is:

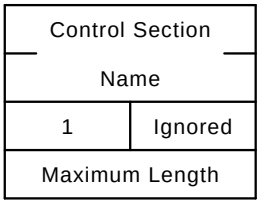
```
.PSECT   name,RW,I,GBL,REL,OVR
```

For an a-sect, the p-sect definition is:

```
.PSECT   . ABS.,RW,I,GBL,ABS,OVR
```

The Linker processes a-sects and c-sects as p-sects with the fixed attributes defined above.

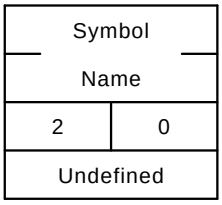
Figure 2–7: Control Section Name Entry Format (Entry Type 1)



2.2.1.3 Internal Symbol Name (Entry Type 2)

The internal symbol name entry (see Figure 2–8) declares the name of an internal symbol with respect to the module. Because the Linker does not support internal symbol tables, the detailed format of this entry is not defined. If the Linker encounters an internal symbol entry while reading the GSD, it ignores it.

Figure 2–8: Internal Symbol Name Entry Format (Entry Type 2)

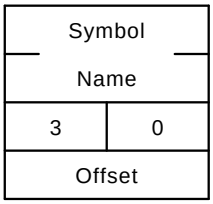


2.2.1.4 Transfer Address (Entry Type 3)

The transfer address entry (see Figure 2–9) declares the transfer address of a module relative to a p-sect. The first two words of the entry define the name of the p-sect. The fourth word indicates the relative offset from the beginning of that p-sect. If no transfer address is declared in a module, do not specify the transfer address entry in the GSD, or specify a transfer address 000001 relative to the default absolute (`. ABS.`) psect.

To begin execution of a program within a particular object module of a program, specify the starting address to the Linker as the transfer address. The Linker passes the first even transfer address it encounters to RT–11 as the program's starting address. Whenever the resulting program executes, the start address indicates the first executable instruction. If there is no transfer address (if, for example, you did not specify one with the .END directive in a MACRO–11 program), or if all transfer addresses are odd, the resulting program does not self-start when you run it.

Figure 2–9: Transfer Address Entry Format (Entry Type 3)



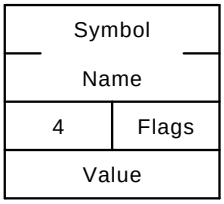
NOTE

When the p-sect is absolute, **Offset** is the actual transfer address if it is not equal to 000001.

2.2.1.5 Global Symbol Name (Entry Type 4)

The global symbol name entry (see Figure 2–10) declares either a global reference or a definition. All definition entries must appear after the declaration of the p-sect under which they are defined, and before the declaration of another p-sect. Global references can appear anywhere within the GSD.

Figure 2–10: Global Symbol Name Entry Format (Entry Type 4)



The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol. The fourth word contains the value of the symbol relative to the p-sect under which it is defined.

The flag byte of the symbol declaration entry has the bit assignments shown in Table 2–3. Bits 1, 2, 4, 6, and 7 are not used by the RT–11 Linker.

Table 2–3: Flag Bits for Global Symbol Name Entry

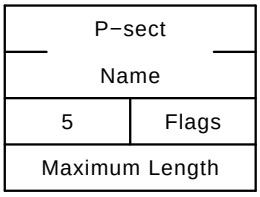
Bit	Meaning
0	Weak Qualifier 0 = Strong (normal) symbol 1 = Weak symbol
3	Definition 0 = Global symbol reference 1 = Global symbol definition
5	Relocation 0 = Absolute symbol value 1 = Relative symbol value

2.2.1.6 Program Section Name (Entry Type 5)

The p-sect name entry (see Figure 2–11) declares the name of a p-sect and its maximum length in the module. It also uses the flag byte to declare the attributes of the p-sect. The default attributes of the p-sect (blank or named with no attributes specified) are as follows:

```
.PSECT      ,RW,I,LCL,REL,CON
```

Figure 2–11: P-sect Name Entry Format (Entry Type 5)



NOTE

The length of all absolute sections is zero.

GSD records must be constructed so that after a p-sect name has been declared, all global symbol definitions pertaining to it must appear before another p-sect name is declared. Global symbols are declared by symbol declaration entries. Thus, the normal format is a series of p-sect names, each followed by optional symbol declarations.

Table 2–4 shows the bit assignments of the flag byte. Bits 1 and 3 are not used by the RT–11 Linker.

Table 2–4: Flag Bits for P-sect Name Entry

Bit	Meaning
0	Save (Root Allocation) 0 = P-sect scope is determined by the value of bit 6. 1 = P-sect is allocated in the root and its scope is global regardless of the value of bit 6.
2	Allocation 0 = P-sect references are to be concatenated with other references to the same p-sect to form the total amount of memory allocated to the section. 1 = P-sect references are to be overlaid. The total amount of memory allocated to the p-sect is the size of the largest request made by individual references to the same p-sect.

Table 2–4 (Cont.): Flag Bits for P-sect Name Entry

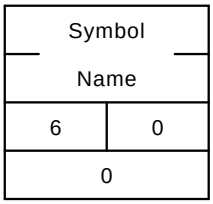
Bit	Meaning
4	Access (not supported by RT–11 monitors) 0 = P-sect has read/write access. 1 = P-sect has read-only access.
5	Relocation 0 = P-sect is absolute and requires no relocation. 1 = P-sect is relocatable and references to the control section must have a relocation bias added before they become absolute.
6	Scope 0 = The scope of the p-sect is local. References to the same p-sect will be collected only within the overlay segment in which the p-sect is defined. 1 = The scope of the p-sect is global. References to the p-sect are collected across overlay segment boundaries.
7	Type 0 = The p-sect contains instruction (I) references. Concatenation of this p-sect will be by word boundary. Globals will be given overlay control blocks. 1 = The p-sect contains data (D) references. Concatenation of this p-sect will be by byte boundary. Globals will not go through the overlay handler.

2.2.1.7 Program Version Identification (Entry Type 6)

The program version identification entry (see Figure 2–12) declares the version of the module. The Linker saves the version identification, or IDENT, of the first module that defines a nonblank version. It then includes this identification on the memory allocation map.

The first two words of the entry contain the version identification. The Linker does not use either the flag byte or the fourth word because they contain no meaningful information.

Figure 2–12: Program Version Identification Entry Format (Entry Type 6)



2.2.1.8 Mapped Array Declaration (Entry Type 7)

The mapped array declaration (see Figure 2–13) allocates space within the mapped array area of the job's memory. The Linker adds the array name to the list of p-sect names, and subsequent RLD blocks can reference it. The Linker adds the length (in units of 32-word blocks) to the job's mapped array allocation. It rounds up the total amount of memory allocated to each mapped array to the nearest 256-word boundary. The contents of the flag byte are reserved and assumed to be zero. (Only the FORTRAN IV compiler produces this v-sect.)

The Linker processes a v-sect as a p-sect with the following attributes:

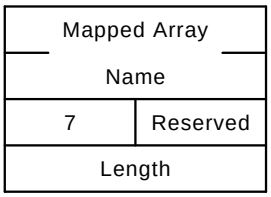
```
.PSECT    . VIR.,RW,D,GBL,REL,CON
```

The size is equal to the number of 32_{10}-word blocks required. If the length is zero, the segment is the root. There must never be any globals under this section, which starts at a base of 0.

NOTE

One additional address window is allocated whenever a mapped array is declared.

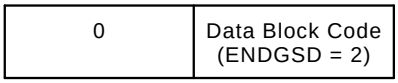
Figure 2–13: Mapped Array Declaration Entry Format (Entry Type 7)



2.2.2 End of Global Symbol Directory Block (ENDGSD)

The end of global symbol directory block (see Figure 2–14) declares that no other GSD blocks are contained in this module. Exactly one end of GSD block must appear in every object module. The length of the data block is one word.

Figure 2–14: End of GSD Data Block



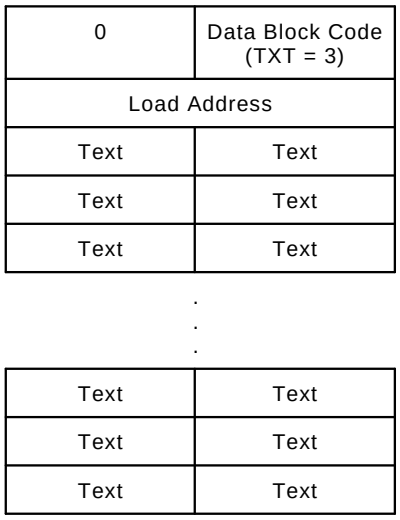
2.2.3 Text Information Block (TXT)

The text information block (see Figure 2–15) contains a byte string of information that the Linker writes directly into the output file. The block consists of a load address followed by the byte string.

Text records can contain words or bytes of information whose final contents have not yet been determined. This information is bound by a relocation directory block that immediately follows the text block. If the text block does not need modification, then a relocation directory block is not needed. Thus, multiple text blocks can appear in sequence before encountering a relocation directory block.

The load address of the text block is specified as an offset from the current p-sect base. At least one relocation directory block must precede the first text block. This RLD block must declare the current p-sect.

Figure 2–15: Text Information Data Block



2.2.4 Relocation Directory Block (RLD)

Relocation directory blocks (see Figure 2–16) contain the information the Linker needs to relocate and link the preceding text information block. Every module must have at least one relocation directory block that precedes the first text information block. The first block does not modify a preceding text block. Instead, it defines the current p-sect and location.

Relocation directory blocks can contain 14 types of entries. These entries are classified as relocation, or location modification entries. Table 2–5 lists the valid entry types.

Figure 2–16: Relocation Directory Data Block

0	Data Block Code (RLD = 4)
Displacement	Command
Information	Information
Information	Information
Information	Information

. .
. .
. .

Command	Information
Information	Displacement
Information	Information
Information	Information
Information	Information
Displacement	Command
Information	Information
Information	Information
Information	Information

Table 2–5: Valid Entry Types for RLD Blocks

Entry Type	Description
1	Internal Relocation
2	Global Relocation
3	Internal Displaced Relocation
4	Global Displaced Relocation
5	Global Additive Relocation
6	Global Additive Displaced Relocation
7	Location Counter Definition
10	Location Counter Modification
11	Program Limits (.LIMIT)
12	P-sect Relocation
13	Not used
14	P-sect Displaced Relocation
15	P-sect Additive Relocation
16	P-sect Additive Displaced Relocation
17	Complex Relocation

Each type of entry is represented by a command byte, which specifies the type of entry and the word or byte modification. This byte is followed by a displacement byte and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding text information block, yields the virtual address in the image that is to be modified. The command byte of each entry has the bit assignments shown in Table 2–6. The following sections describe the valid entry types for the RLD data block.

Table 2–6: Bit Assignments for the RLD Command Byte

Bit	Meaning
0–6	Specify the type of entry. Although there is room to specify 128 command types, only 14_{10} are currently implemented in the RT–11 Linker.
7	Modification (the B bit in Figures 2–17 through 2–30). This feature is not supported by RT–11, and the bit is ignored if set. The RT–11 Linker supports word relocation, not byte relocation. 0 = The command modifies an entire word. 1 = The command modifies only one byte.

2.2.4.1 Internal Relocation (Entry Type 1)

This type of entry (see Figure 2–17) relocates a direct pointer to an address within a module. The Linker adds the current p-sect base address to a specified constant and writes the result into the output file at the calculated address—that is, it adds a displacement byte to the value calculated from the load address of the preceding text block.

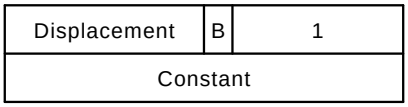
For example:

```
A:      MOV     #A,R0
```

or

```
        .WORD   A
```

Figure 2–17: Internal Relocation (Entry Type 1)



2.2.4.2 Global Relocation (Entry Type 2)

This type of entry (see Figure 2–18) relocates a direct pointer to a global symbol. The Linker obtains the definition of the global symbol and writes the result into the output file at the calculated address.

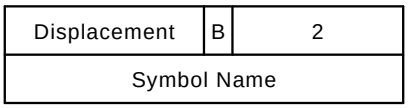
For example:

```
        MOV     #GLOBAL,R0
```

or

```
        .WORD   GLOBAL
```

Figure 2–18: Global Relocation (Entry Type 2)



2.2.4.3 Internal Displaced Relocation (Entry Type 3)

This type of entry (see Figure 2–19) relocates a relative reference to an absolute address from within a relocatable control section. The Linker subtracts from the specified constant the address plus 2 into which the relocated value is to be written. The Linker then writes the result into the output file at the calculated address.

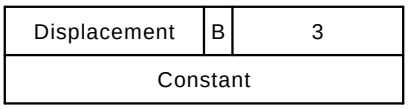
For example:

```
CLR     177550
```

or

```
MOV     177550,R0
```

Figure 2–19: Internal Displaced Relocation (Entry Type 3)



2.2.4.4 Global Displaced Relocation (Entry Type 4)

This type of entry (see Figure 2–20) relocates a relative reference to a global symbol. The Linker obtains the definition of the global symbol and subtracts from the definition value the address plus 2 into which the relocated value is to be written. It then writes the result into the output file at the calculated address.

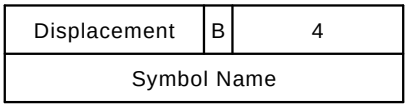
For example:

```
CLR     GLOBAL
```

or

```
MOV     GLOBAL,R0
```

Figure 2–20: Global Displaced Relocation (Entry Type 4)



2.2.4.5 Global Additive Relocation (Entry Type 5)

This type of entry (see Figure 2–21) relocates a direct pointer to a global symbol with an additive constant. The Linker obtains the definition of the global, adds the specified constant, and then writes the resultant value into the output file at the calculated address.

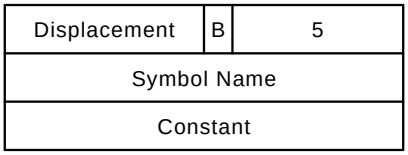
For example:

```
MOV     #GLOBAL+2,R0
```

or

```
.WORD   GLOBAL-4
```

Figure 2–21: Global Additive Relocation (Entry Type 5)



2.2.4.6 Global Additive Displaced Relocation (Entry Type 6)

This type of entry (see Figure 2–22) relocates a reference to a global symbol with an additive constant. The Linker obtains the definition of the global symbol and adds the specified constant to the definition value. The Linker subtracts from the resultant additive value the address plus 2 into which the relocated value is to be written. It then writes the result into the output file at the calculated address.

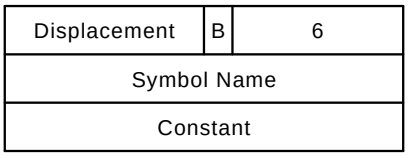
For example:

```
CLR     GLOBAL+2
```

or

```
MOV     GLOBAL-5,R0
```

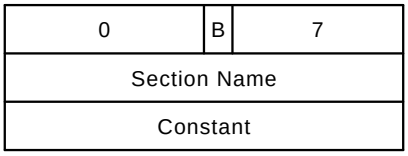
Figure 2–22: Global Additive Displaced Relocation (Entry Type 6)



2.2.4.7 Location Counter Definition (Entry Type 7)

This type of entry (see Figure 2–23) declares a current p-sect and location counter value. The Linker stores the control base as the current control section. It adds the current control section base to the specified constant and stores the result as the current location counter value.

Figure 2–23: Location Counter Definition (Entry Type 7)



2.2.4.8 Location Counter Modification (Entry Type 10)

This type of entry (see Figure 2–24) modifies the current location counter. The Linker adds the current p-sect base to the specified constant and stores the result as the current location counter.

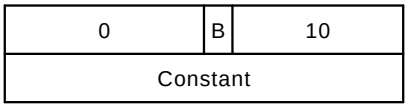
For example:

```
.=.+N
```

or

```
.BLKB N
```

Figure 2–24: Location Counter Modification (Entry Type 10)



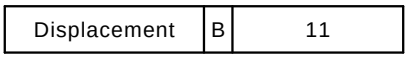
2.2.4.9 Program Limits (Entry Type 11)

This type of entry (see Figure 2–25) is generated by the .LIMIT assembler directive. The Linker obtains the first address above the header, which is normally the beginning of the stack, and the highest address allocated to the job. It then writes these addresses into the output file at the calculated address and the following location, respectively.

For example:

```
.LIMIT
```

Figure 2–25: Program Limits (Entry Type 11)



2.2.4.10 P-sect Relocation (Entry Type 12)

This type of entry (see Figure 2–26) relocates a direct pointer to the beginning address of another p-sect (other than the p-sect in which the reference is made) within a module. The Linker obtains the current base address of the specified p-sect and writes it into the output file at the calculated address.

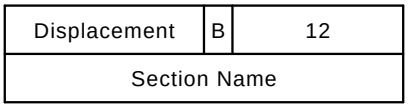
For example:

```
        .PSECT  A
B:
        .
        .
        .
        .PSECT  C
        MOV     #B,R0
```

or

```
        .WORD B
```

Figure 2–26: P-sect Relocation (Entry Type 12)



2.2.4.11 P-sect Displaced Relocation (Entry Type 14)

This type of entry (see Figure 2–27) relocates a relative reference to the beginning address of another p-sect within a module. The Linker obtains the current base address of the specified p-sect. It then subtracts from the base value the address plus 2 into which the relocated value is to be written and writes the result into the output file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
        .PSECT  C
        MOV     B,R0
```

Figure 2–27: P-sect Displaced Relocation (Entry Type 14)

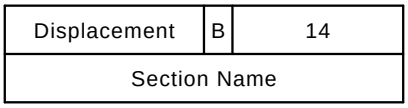
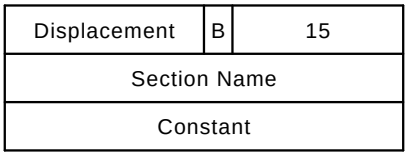


Figure 2–28: P-sect Additive Relocation (Entry Type 15)



2.2.4.12 P-sect Additive Relocation (Entry Type 15)

This type of entry (see Figure 2–28) relocates a direct pointer to an address in another p-sect within a module. The Linker obtains the current base address of the specified p-sect. It adds the base to the specified constant and then writes the result into the output file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT  D
        MOV     #B+10,R0
        MOV     #C,R0
```

or

```
        .WORD   B+10
        .WORD   C
```

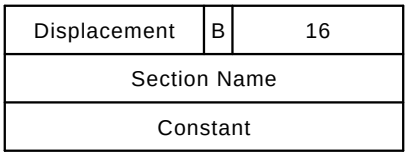
2.2.4.13 P-sect Additive Displaced Relocation (Entry Type 16)

This type of entry (see Figure 2–29) relocates a relative reference to an address in another p-sect within a module. The Linker obtains the current base address of the specified p-sect and adds it to the specified constant. Next, it subtracts from the resultant additive value the address plus 2 into which the relocated value is to be written. It writes the final result into the output file at the calculated address.

For example:

```
        .PSECT  A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT  D
        MOV     B+10,R0
        MOV     C,R0
```

Figure 2–29: P-sect Additive Displaced Relocation (Entry Type 16)



2.2.4.14 Complex Relocation (Entry Type 17)

This type of entry (see Figure 2–30) resolves a complex relocation expression. A complex relocation expression is one in which any of the MACRO–11 binary or unary operations are permitted with any type of argument, regardless of whether the argument is unresolved global, relocatable to any p-sect base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically specified operation codes and arguments. The operation codes each occupy one byte and the entire RLD command must fit in a single data block. Table 2–7 shows the list of valid operation codes. Note that complex relocation on foreground links causes a warning message from the Linker. The results of complex relocation will be correct if no relocatable symbols are involved.

Table 2–7: Operation Codes for Complex Relocation

Code	Description
0	No operation
1	Addition (+)
2	Subtraction (–)
3	Multiplication (*)

Table 2–7 (Cont.): Operation Codes for Complex Relocation

Code	Description
4	Division (/)
5	Logical AND (&)
6	Logical inclusive OR (!)
7	Exclusive OR
10	Negation (–)
11	Complement (^C)
12	Store result (command termination)
13	Store result with displaced relocation (command termination)
16	Fetch global symbol. It is followed by four bytes containing the symbol name in Radix–50 representation.
17	Fetch relocatable value. It is followed by one byte containing the section number, and two bytes containing the offset within the section.
20	Fetch constant. It is followed by two bytes containing the constant.

The STORE commands (codes 12 and 13) indicate that the value is to be written into the output file at the calculated address.

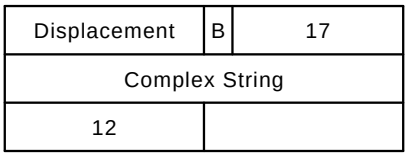
The Linker evaluates all operands as 16-bit signed quantities using two's complement arithmetic. The results are equivalent to expressions that are evaluated internally by the assembler. Note the following rules:

1. An attempt to divide by 0 yields a 0 result. The Linker issues a warning message.
2. All results are truncated from the left in order to fit into 16 bits. No diagnostic is issued if the number was too large. If the result modifies a byte, the Linker checks for truncation errors. (Byte operations are not allowed.)
3. All operations are perfomed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
        .PSECT  ALPHA
A:
        .
        .
        .
        .PSECT  BETA
B:
        .
        .
        .
        MOV     #A+B-<G1/G2&^C<177120!G3>>,R1
```

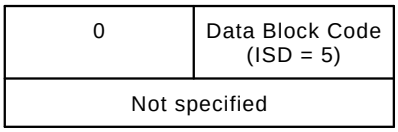
Figure 2–30: Complex Relocation (Entry Type 17)



2.2.5 Internal Symbol Directory Block (ISD)

Internal symbol directory blocks (see Figure 2–31) declare definitions of symbols that are local to a module. The Linker does not support this feature; therefore, a detailed data block format is not documented here. If the Linker encounters this type of data block, it ignores it.

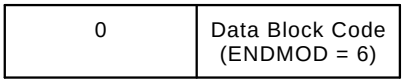
Figure 2–31: Internal Symbol Directory Data Block



2.2.6 End Of Module Block (ENDMOD)

The end of module block (see Figure 2–32) declares the end of an object module. Exactly one end of module record must appear in each object module. It is one word long.

Figure 2–32: End of Module Data Block



2.3 Symbol Table Definition File Format (.STB)

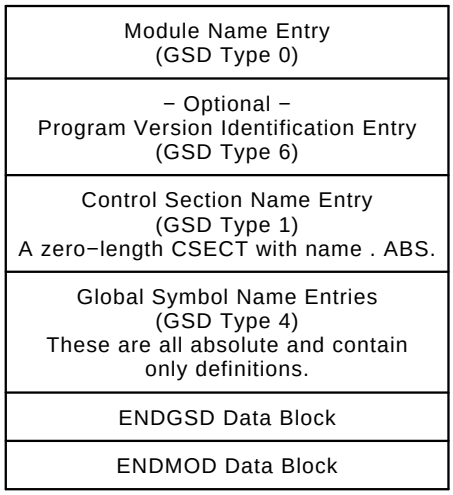
The RT–11 Linker can produce a symbol table (.STB) file as its third output file. The text of the .STB file consists of global symbol table definitions. For example, if the source file contains X == 10, the .STB file contains X = 10. Or, if the source file contains A = FOO, the .STB file contains the address of FOO.

The .STB file can serve as a communication link between a background and a foreground job. This communication comes about when you link the background job and obtain a .STB file as output. Then, when you link the foreground job, include the .STB file as one of the input files. The foreground job is then able to reference

symbols used by the background job. Similarly, you can use the .STB file to create a communication link between a program and a symbolic debugger.

The internal format of the .STB file consists entirely of Global Symbol Directory (GSD) data blocks followed by one End of Global Symbol Directory (ENDGSD) data block and one End of Module (ENDMOD) data block. Figure 2–33 illustrates the .STB file format.

Figure 2–33: .STB File Format



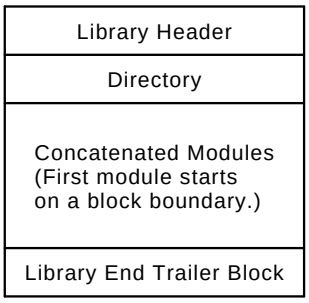
2.4 Library File Format (.OBJ and .MLB)

A library file contains concatenated modules and some additional information. RT–11 supports object and macro libraries. Object libraries usually have an .OBJ file type; macro libraries usually have a .MLB file type.

The modules in an object or macro library file are preceded by a Library Header Block and Library Directory, and are followed by the Library End Block, or trailer. Figure 2–34 shows the format of an object or macro library file.

Diagrams of each component in the library file structure are included in the sections that follow. See the *RT–11 System Utilities Manual* for information on using the librarian.

Figure 2–34: Library File Format (.OBJ and .MAC)



2.4.1 Library Header Format

The library header describes the status of the file. Of the two tables that follow, Table 2–8 shows the contents of the object library header and Table 2–9 shows the contents of the macro library header.

All numeric values shown are octal. The date and time, which are in standard RT–11 format, are the date and time the library was created. This information is displayed when the library is listed.

Table 2–8: Object Library Header Format

Offset	Contents	Description
0	1	Library header block code
2	42	
4	7	Librarian code
6	500	Library version number
10	0	1 if library created with /X option
12		Date in RT–11 format (0 if none)
14		Time expressed in two words
16		
20	0	Reserved
22	0	Reserved
24	0	Reserved
26	10	Directory relative start address
30		Number of bytes in directory and header
32	0	Reserved
34		Next insert relative block number
36		Next byte within block
40		Directory starts here

Table 2–9: Macro Library Header Format

Offset	Contents	Description
0	1001	Library type and ID code
2	500	Library version number
4	0	Reserved
6		Date in RT–11 format (0 if none)
10		Time expressed in two words
12		
14	0	Reserved
16	0	Reserved
20	0	Reserved
22	0	Reserved

Table 2–9 (Cont.): Macro Library Header Format

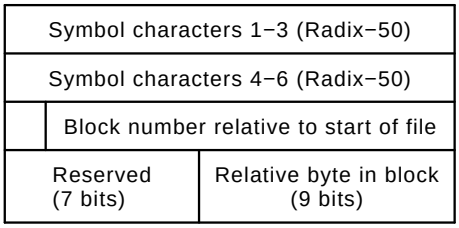
Offset	Contents	Description
24	0	Reserved
26	0	Reserved
30	0	Reserved
32	10	Size of directory entries
34		Directory starting relative block number
36		Number of directory entries allocated; default is 200
40		Number of directory entries available

2.4.2 Library Directories

There are two kinds of library directories: for object libraries, the directory is an Entry Point Table (EPT); for macro libraries, the directory is a Macro Name Table (MNT). The EPT directory (see Table 2–8 for the header format) consists of 4-word entries that contain information related to all modules in the library file.

Note that if you use the librarian /N option for object libraries to include module names, bit 15 of the relative block number word is set to 1. If you invoke the librarian with the monitor LIBRARY command, module names are never included.

Figure 2–35: Library Directory Format (.OBJ)



In the library directory, the symbol characters represent the entry point or the macro name. The relative byte maximum is 777_8.

The object library directory starts on the first word after the library header, word 40_8. The object library directory is only long enough to accommodate the exact number of modules in the library and space for this directory is not preallocated. The directory is kept in memory during librarian operations, and the amount of available memory is the only limiting factor on the maximum size of the directory. Reserved locations in the header and at the end of the directory—those not used by the directory—are zero filled. Modules follow the directory and they are stored beginning in the next block after the directory.

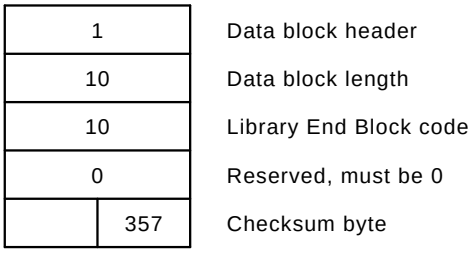
The macro library directory starts on a block boundary, relative block 1 of the library file. (See Table 2–9 for the header format) Its size is preallocated. The default size is two blocks, but you can change this by using the librarian /M option. Unused entries in the directory are filled with –1. Macro files are stored starting on the block boundary after the directory. This is relative block 3 of the library file if you use the default directory size.

Modules in libraries are concatenated by byte. (See Figure 2–2 for an example of byte concatenation.) This means that a module can start on an odd address. When this occurs, the Linker shifts the module to an even address at link time.

2.4.3 Library End Block Format

Following all modules in an object or macro library is a specially coded Library End Block, or trailer, which signifies the end of the file (see Figure 2–36).

Figure 2–36: Library End Block Format



2.5 Absolute Binary File Format (.LDA)

Both the Linker /L option and the keyboard monitor LINK command /LDA option produce output files in an absolute binary format. Such a format is suitable for down-line loading of programs, for loading stand-alone application programs, and as input to special programs that put code into ROM (Read-Only Memory).

Absolute binary format, shown in Figure 2–37, consists of a sequence of data blocks, where each block represents the data to be loaded into a specific portion of memory. The data portion of each block consists of the absolute load address of the block, followed by the absolute data bytes to be loaded into memory beginning at the load address. There can be as many data blocks as necessary in an .LDA file. The last block of the file is special because it contains only the program start address, or transfer address, in its data portion. If this address is even, the Absolute Loader passes control to the loaded program at this address. If the address is odd (that is, if the program has no transfer address, or the transfer address was specified as a byte boundary), the loader halts after loading. The final block of the .LDA file is recognized by the fact that its length is six bytes.

The general procedure for loading a program that will execute in a stand-alone environment is as follows:

1. Using the console microcode, load the Bootstrap Loader into memory. If the processor does not contain console microcode, you must toggle the Bootstrap Loader into memory.
2. Using the Bootstrap Loader, load the Absolute Loader into memory.
3. Using the Absolute Loader, load the .LDA file into memory and begin execution.

Most PDP–11 processors contain console microcode that make step 1 unnecessary. The procedure for loading stand-alone programs is described in the *Microcomputer Processor Handbook*. LSI–11/23 computer systems do not have bootstrap loader microcode and require the use of steps 1 and 2.

The Bootstrap Loader is printed on the PDP–11 Programming Card, which is part of every RT–11 distribution kit.

The load module's data blocks contain only absolute binary load data and absolute load addresses. All global references have been resolved and the Linker has performed the appropriate relocation.

2.6 Standard Save Image File Format (.SAV)

Standard save images do not support separated I–D space. Standard save images can be run in the background environment under any monitor and as virtual jobs in the foreground or system job environment under mapped monitors. Save image files normally have a .SAV file type. This format is an image of the program exactly as it would appear in memory. (Block 0—the first 256-word unit—of the file corresponds to memory locations 0–776, block 1 to locations 1000–1776, and so on.) See Table 2–10 for the contents of block 0 of a standard .SAV file. (Note that not all locations are used for each link; for example, whether the job is for a mapped environment or whether it is overlaid affect block 0.) See also the *RT–11 System Utilities Manual* for more information on the load modules created by the Linker.

Figure 2–37: Absolute Binary Format (.LDA)

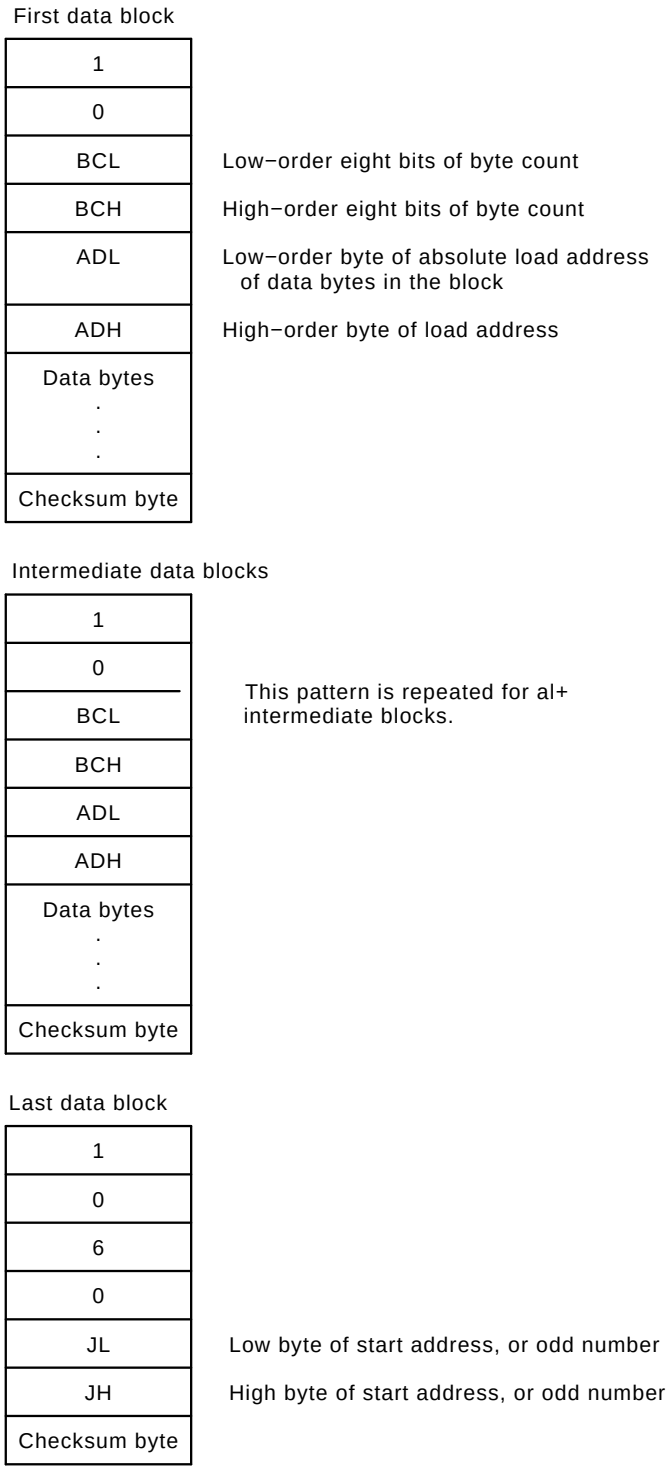


Table 2–10: Information in Block 0 of a .SAV Image

Offset	Contents
0	VIR in Radix–50 if the Linker /V option was used.
2	Virtual high limit if Linker /V option was used.
4	Job definition word ($JSX) bits. See Table 2–11 for bit definitions.
6	Reserved
10	Reserved
12	Reserved
14	BPT trap PC(mapped monitors only)
16	BPT trap PSW (mapped monitors only)
20	IOT trap PC (mapped monitors only)
22	IOT trap PSW (mapped monitors only)
24	Reserved
26	Reserved
30	Reserved
32	Overlay definition word (SV.CVH) bits. See tables 2–12 and 2–13 for bit definitions.
34	Trap vector PC (TRAP)
36	Trap vector PSW (TRAP)
40	Program's relative start address
42	Initial location of stack pointer (changed by /M option)
44	Job Status Word
46	USR swap address
50	Program's high limit
52-62	Reserved
64	Address of overlay handler table for overlaid files
66	Address of start of window definition blocks (if /V used)
70–356	Reserved
360–377	Bitmap area

Location 4 in block 0 contains the job definition word, $JSX. The low three bits in $JSX define certain characteristics about how the program can be loaded and whether the program can be run under VBGEXE. The other bits in $JSX pertain only to programs that support separated I–D space addressing and Supervisor mode. See Table 2–11 for a description of the $JSX bits.

In overlaid programs, location 32 in absolute block 0 contains the overlay definition word, SV.CVH. The low 5 bits in SV.CVH define the overlay handler and describe the overlay handler characteristics. The high 3 bits in the low byte are reserved and the high byte contains the edit level for the overlay handler source file (and are of no concern to the user). See Table 2–12 for the bit mask that defines each overlay handler and Table 2–13 for a description of the SV.CVH bits.

Locations 360–377 are the CCB and are are restricted for use by the system. The Linker stores the program memory usage bits in these eight words, which are called a bitmap. Each bit represents one 256-word block of memory and is set if the program occupies any part of that block of memory. Bit 7 of byte 360 corresponds to locations 0 through 777; bit 6 of byte 360 corresponds to locations 1000 through 1777, and so on. The monitor uses this information when it loads the program.

The monitor commands R and RUN load and start a program stored in a .SAV file. (The RUN command is actually a combination of the GET and START commands.) First, the Keyboard Monitor reads block 0 of the .SAV file into an internal USR buffer. It extracts information from locations 40–64 and 360–377 (the bitmap, described above). Using the protection bitmap (called LOWMAP), which resides in RMON, KMON checks each word in block 0 of the file. It does not load locations that are protected, such as location 54 and the device interrupt vectors. It loads unprotected locations into memory from the USR buffer. Next, KMON sets location 50 to the top of usable memory, or to the top of the user program, whichever is greater.

If the RUN command (or the GET command) was issued, KMON checks the bitmap from locations 360–377 of the .SAV file. For each bit that is set, it loads the corresponding block of the .SAV file into memory. However, if KMON is in memory space that the program needs to use, KMON puts the block of the .SAV file into a USR buffer and then moves it to the file SWAP.SYS.

Finally, when it is time to begin execution of the program, KMON transfers control to RMON. RMON reads the parts of the program, if any, that are stored in SWAP.SYS into memory, where they overlay KMON and possibly the USR. The monitor keeps track of the fact that KMON (and perhaps the USR) are swapped out, and execution of the program begins.

Table 2–11: Job Definition Word ($JSX) Bit Definitions

Bit Mask	Symbol	Meaning
000001	REQID$	Job requires separated I–D space.
000002	USEID$	Job uses separated I–D space, if possible.
000004	REQSM$	Job requires Supervisor mode support.
000010	USESM$	Job uses Supervisor mode, if possible.
000020	ALL64$	.SETTOP under VBGEXE supported to top of memory (-2).
000040	IOPAG$	IOPAGE region under VBGEXE is mapped to PAR7.

Table 2–11 (Cont.): Job Definition Word ($JSX) Bit Definitions

Bit Mask	Symbol	Meaning
000100	NOVBG$	Job cannot be run under VBGEXE.
000200	VBGEX$	Job can be run under VBGEXE.
000400-100000		Reserved.

Table 2–12: Identifying Overlay Handlers in SV.CVH

Bit Mask	Overlay Handler
000001	OHANDL
000003	VHANDL
000012	XHANDL
000027	ZHANDL

Table 2–13: Overlay Definition Word (SV.CVH) Bit Definitions

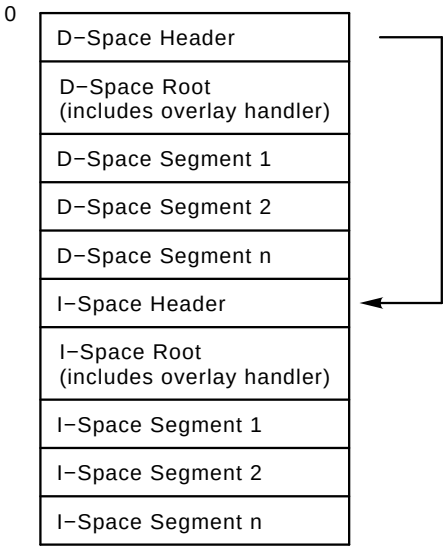
Bit Mask	Symbol	Meaning
000001	CVH.DK	Supports disk-type (/O) overlays
000002	CVH.MP	Supports virtual-type (/V) overlays
000004	CVH.ID	Supports separated I–D space and Supervisor Mode
000010	CVH.XH	Supports single virtual-type overlay only
000020	CVH.LO	Program loader (VBGEXE) initializes /O overlay area
000040-000200		Reserved
000400-100000	CVH.ED	The edit level of the overlay handlers source file (for Digital internal use only).

2.7 Extended Save Image File Format (.SAV)

Support for separated I–D space addressing produces the extended save image file format. Such a job has an odd transfer address that stops it from being run as a standard save image.

The extended save image file format differs from the standard in a number of ways, as illustrated in the following diagram. There is a header for the D-space segments followed by one for the I-space segments. There is a root for each address space and each root contains an overlay handler for that address space. Note also that the D-space header is loaded at physical block 0, while the I-space header is loaded at virtual block 0 (defined at location 40).

Figure 2–38: Extended Save Image File Format



The contents of the D-space header (absolute block 0 of the save image) are described in Table 2–14. The contents of the I-space header are described in Table 2–15. In Table 2–14 note particularly locations 4, 30, 40, and 60.

LINK couples each I-space segment with a corresponding D-space segment (if there is a corresponding D-space segment). The coupling is identified by a segment number (*segnum*) that is stored in the following 3-word routine at the beginning of each I-space segment:

```
MOV    (PC)+,R0
.WORD  #12.*segnum
RETURN
```

As mentioned, there need not be a corresponding D-space segment for each I-space segment. Further, there is no segment identifier in D-space segments; they are identified only by an entry in the overlay handler table and the corresponding I-space segment.

Table 2–14: Information in Absolute Block 0 (D-Space Header) of Extended .SAV Image

Offset	Contents
0	VIR in Radix–50 if the Linker /V option was used.
2	Virtual high limit if Linker /V option was used.
4	Job definition word ($JSX) bits. See Table 2–11 for bit definitions.
6	Reserved

Table 2–14 (Cont.): Information in Absolute Block 0 (D-Space Header) of Extended .SAV Image

Offset	Contents
10	Reserved
12	Reserved
14	BPT trap PC
16	BPT trap PSW
20	IOT trap PC
22	IOT trap PSW
24	Reserved
26	Reserved
30	Contains 000000 or 100000. If 000000, all system subroutines linked into program support separated I–D space. Program can be run. If 100000, at least one system subroutine which cannot support separated I–D space has been linked into the program. Program cannot be run; VBGEXE returns an error at attempt to run program.
32	Overlay definition word (SV.CVH) bits. See tables 2–12 and 2–13 for bit definitions.
34	Trap vector PC (TRAP)
36	Trap vector PSW (TRAP)
40	Value of *2*blknum+1*, where *blknum* is the relative file block number of the I-space CCB (first block after D-space blocks).
42	Initial location of stack pointer (changed by /M option)
44	Job Status Word
46	USR swap address
50	Program's high limit
52	Reserved.
54	Reserved.
56	Reserved.

Table 2–14 (Cont.): Information in Absolute Block 0 (D-Space Header) of Extended .SAV Image

Offset	Contents
60	Contains <IDS> in Radix–50 to indicate separated I–D space job.
62	Reserved.
64	Address of overlay handler table for overlaid files
66	Address of start of window definition blocks (if /V used)
70–356	Reserved
360–377	CCB (core control block) bitmap area

Table 2–15: Information in Relative Block 0 (I-Space Header) of Extended .SAV Image

Offset	Contents
0–36	Contain 000000
40	Program's relative start address (interpreted as a User mode I-space virtual address.)
42–46	Contain 000000
50	Program's I-space high limit
52–356	Contain 000000
360–377	I-space CCB (core control block) bitmap area

2.8 Relocatable File Format (.REL)

To link a foreground job, use the Linker /R option or the keyboard monitor LINK command with the /FOREGROUND option. This causes the Linker to produce output in a linked, relocatable format, with a .REL file type. Note that system files are also stored in relocatable format. The only difference is that system files use a file type of .SYS instead of .REL.

Table 2–16: Information in Block 0 of a .REL Image

Offset	Contents
0	VIR in Radix–50 if the Linker /V option was used.
2	Virtual high limit if Linker /V option was used.
4	Job definition word ($JSX) bits. See Table 2–11 for bit definitions.
6	Reserved

Table 2–16 (Cont.): Information in Block 0 of a .REL Image

Offset	Contents
10	Reserved
12	Reserved
14	BPT trap PC(mapped monitors only)
16	BPT trap PSW (mapped monitors only)
20	IOT trap PC (mapped monitors only)
22	IOT trap PSW (mapped monitors only)
24	Reserved
26	Reserved
30	Reserved
32	Overlay definition word (SV.CVH) bits. See tables 2–12 and 2–13 for bit definitions.
34	Trap vector PC (TRAP)
36	Trap vector PSW (TRAP)
40	Program's relative start address
42	Initial location of stack pointer (changed by /M option)
44	Job Status Word
46	USR swap address
50	Program's high limit
52	Size of program's root segment, in bytes.
54	Stack size, in bytes (changed by /R:n option).
56	Size of overlay region, in bytes (0 if not overlaid)
60	.REL file ID (REL in Radix–50)
62	Relative block number for start of relocation information
64	Address of overlay handler table for overlaid files
66	Address of start of window definition blocks (if /V used)
70–356	Reserved
360–377	Bitmap area

The object modules used to create a .REL file are linked as if they were a background .SAV image, with a base of 1000. This permits you to use .ASECT directives to store information in locations 0 through 777 in .REL files. All global references have been resolved. The Linker does not relocate the .REL file at link time; it merely includes relocation information to be used at FRUN time. The relocation information in the

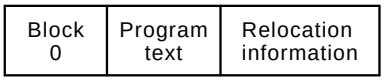
file is used to determine which words in the program must be relocated when the job is installed in memory.

There are two types of .REL files to consider: those programs with overlay segments, and those without them.

2.8.1 .REL Files Without Overlays

A .REL file for a program without overlays appears as shown in Figure 2–39.

Figure 2–39: .REL File Without Overlays



Block 0 (relative to the start of the file) contains the information shown in Table 2–10. Some of this information is used by the FRUN processor.

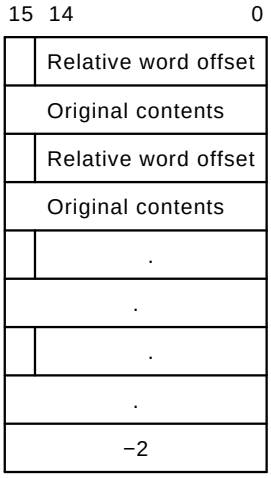
In the case of a program without overlays, the FRUN processor performs the following general steps to install a foreground job.

1. It reads block 0 of the file into an internal monitor buffer.
2. It obtains the amount of memory required for the job from location 52_8 of block 0 of the file, and allocates the space in memory by moving KMON and the USR down.
3. It reads the program text into the allocated space.
4. It reads the relocation information into an internal buffer.
5. It relocates the locations indicated in the relocation information area by adding or subtracting the relocation quantity. This quantity is the starting address the job occupies in memory, adjusted by the relocation base of the file. .REL files are linked with a base of 1000.

The relocation information consists of a list of addresses relative to the start of the user's program. The monitor scans the list, and for each relative address computes an actual address. That address is then loaded with its original contents plus or minus the relocation constant. The relocation information is shown in Figure 2–40.

In Figure 2–40, bits 0 through 14 represent the relative address to relocate divided by 2. This implies that relocation is always done on a word boundary, which is indeed the case. Bit 15 indicates the type of relocation to perform—positive or negative. The relocation constant (which is the load address of the program) is added to or subtracted from the indicated location depending on the sense of bit 15; 0 implies addition, while 1 implies subtraction. A full 16-bit word is the original contents. The

Figure 2–40: Root Relocation Information Format



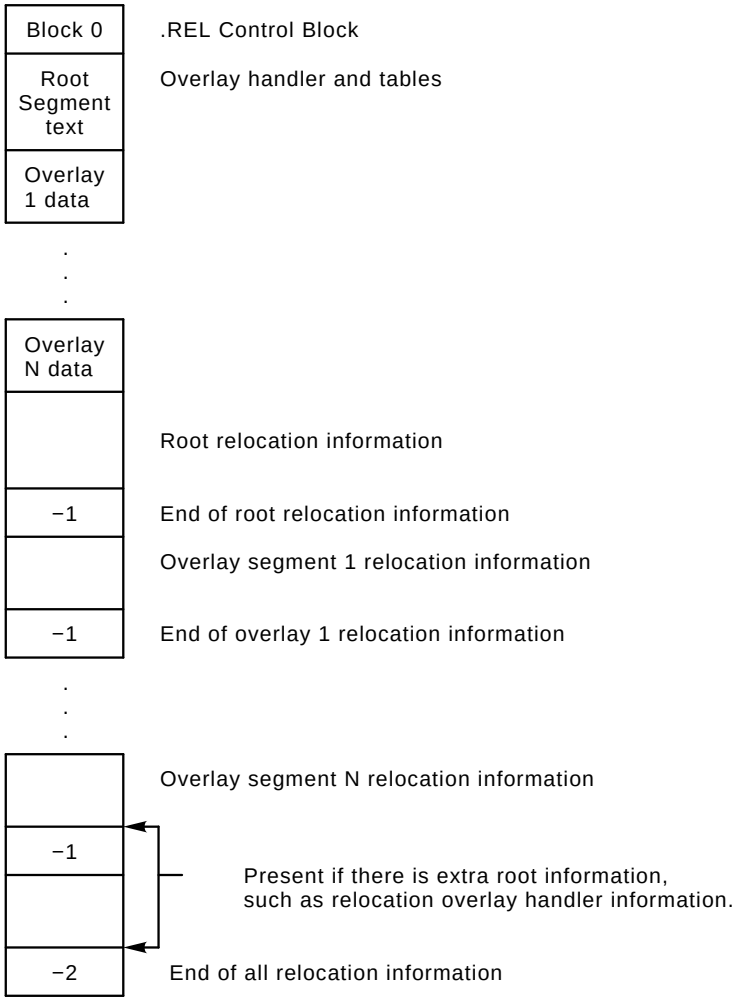
value 177776, or –2, terminates the list of relocation information for a file without overlays.

2.8.2 .REL Files with Overlays

When you include overlays in a program, in addition to relocating the root segment, the FRUN processor must also relocate the overlay segments. Since overlays are not permanently memory resident but are read in from the file as needed, they require an additional operation. FRUN relocates each overlay segment and rewrites it into the file before the program begins execution. (Therefore, the volume containing the file must be write-enabled.) Thus, when the overlay is called into memory during program execution, it is correct. This process takes place each time you run an overlaid file with FRUN or SRUN. The relocation information for overlaid files contains both the list of addresses to be modified and the original contents of each location. This allows the file to be executed again after the first usage. It is necessary to preserve the original contents in case some change has occurred in the operating environment. Examples of these changes include using a different monitor version, running on a system with a different amount of memory, and having a different set of device handlers or system jobs resident in memory. Figure 2–41 shows a .REL file with overlays. Refer to Figure 2–40 and Figure 2–42 for more detail of the relocation information.

In the case of a .REL file with overlays, location 56 of block 0 of the .REL file contains the size in bytes of all the overlay regions. FRUN adds this size to the size of the program base segment (in location 52) to allocate space for the job.

Figure 2–41: .REL File with Overlays

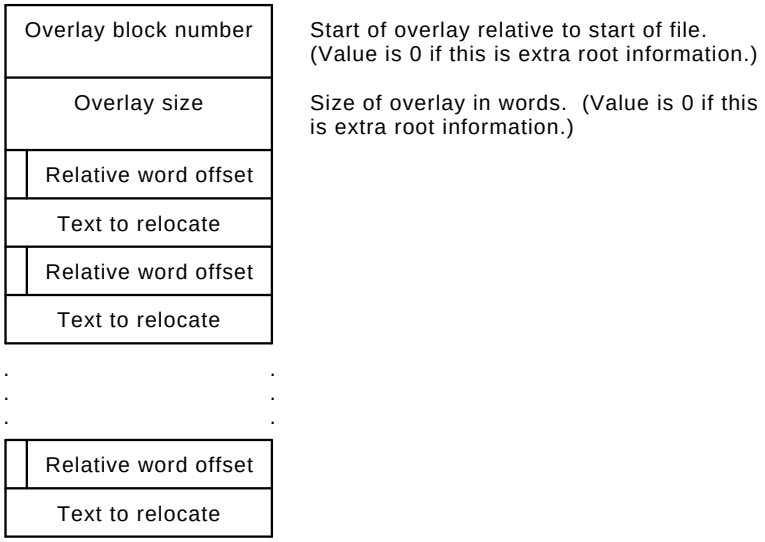


After FRUN relocates the program base (root) code, it reads each existing overlay into the program overlay region in memory, relocates it using the overlay relocation information, and then writes it back into the file.

The root relocation information section is terminated with a –1. This –1 is also an indication that an overlay segment relocation block follows.

The relocation is relative to the start of the program and is interpreted as if it were in a file without overlays (that is, bit 15 indicates the type of relocation, and the displacement is the true displacement divided by 2). Encountering –1 indicates that a new overlay region begins here; a –2 indicates the termination of all relocation information.

Figure 2–42: Overlay Segment Relocation Block



2.9 Stream ASCII File Format

Source files, such as MACRO–11 and FORTRAN IV programs, and text files that you create with an editor are in stream ASCII format. These files consist of a series of bytes, each byte representing an ASCII character. Stream ASCII files have no special headers or end blocks, nor do they include any formatted binary blocks.

2.9.1 Defining a Line or Record

RT–11 typically defines a line of input as a string of characters terminated by a linefeed (012_8) or formfeed (014_8). Some programs also define vertical tab (013_8) as a line terminator. In RT–11, linefeeds generally come as part of a carriage return/linefeed pair (the RETURN key), but treating carriage-return (015_8) as a line terminator can cause complications in some programming applications and should be avoided. For example, treating carriage return as a line terminator makes it more difficult to do overprinting. Instead, consider the linefeed to be the line or record terminator.

2.9.2 End-of-File

Although not required, an 032_8, or CTRL/Z character, may terminate a stream ASCII file. When you use PIP with the /A option (or when you use the monitor COPY/ASCII command) to copy an ASCII file, PIP expects to find a CTRL/Z at the end of the file. If there is an embedded CTRL/Z character within the file, PIP considers the CTRL/Z to mark the file's end, and any characters following the CTRL/Z are ignored. When you use PIP in its default mode (image) or when you use the monitor COPY command without any option to copy an ASCII file, PIP does not look for a CTRL/Z character. It simply copies blocks until it reaches the end of the file.

Programs that process stream ASCII files should ignore nulls (000_8). Many programs that write ASCII files fill the remainder of the last block of the file with nulls instead of appending a CTRL/Z. When reading such a file, a program can read all the data in the file, ignore the trailing nulls, then receive an end-of-file error from RT–11 when it tries to read beyond the last block of the file.

2.10 CREF File Format

The RT–11 CREF program produces a cross-reference listing. You can only run CREF indirectly—that is, by chaining to it from another program, such as your own language processor. CREF appends its cross-reference table to the listing file your calling program creates.

To chain to CREF, you must first store some information in the chain communication area (absolute locations 500 through 776) of the calling program. Table 2–17 lists the information that CREF requires.

Table 2–17: CREF Chain Interface Specification

Location	Contents		Description
500	.RAD50	/SY /	The file specification to call CREF:
502	.RAD50	/CRE/	
504	.RAD50	/F /	
506	.RAD50	/SAV/	
510			RT–11 channel number of output file
512			Radix–50 name of output device
514			Highest output block number written, plus 1
516			RT–11 channel number of input file
520			Radix–50 name of input device
522			Highest input block number written, plus 1
524			Listing format: 0 = 80 columns, –1 = 132
526	.RAD50	/dev/	Program to chain back to. (If this value is zero, CREF closes the listing file and exits.)
530	.RAD50	/fil/	
532	.RAD50	/nam/	
534	.RAD50	/typ/	
536–776			.ASCIZ string for CREF to use as title line (no page number)

The input file you supply to CREF must consist of 12_{10}-byte entries, one entry for each reference to a symbol. Table 2–18 shows the format of the entries.

Table 2–18: Entry Format for CREF Input File

Octal Byte	Offset Value
0	Section descriptor: Bits 0 through 4 contain an alphabetic character for CREF to use as the section name. The ASCII value is stripped to 5 bits. Bits 5 through 7 contain the section number. This number controls the order of the sections.
1–6	The ASCII name of the symbol.
7–10	The page number, in binary. Put –1 here if you are not using page numbers.
11–12	The line number, in binary.
13	A one-character identifier for CREF to print next to this reference. Typically, this character is used to identify a destructive reference or a definitional reference.

2.11 BUP Saveset Section Definition Block Format

A BUP saveset consists of at least one section; more that one section is created if the saveset spans more than one output volume. Each section is a complete, protected RT–11 file. Block 0 of each section contains information for that section that BUP uses in directory and restoration operations. Table 2–19 describes the contents of a BUP saveset block 0.

Table 2–19: Contents of Block 0 of a BUP Saveset Section

Starting Offset	Contents
0–1	Number of definition blocks (currently = 1)
2–5	Radix–50 "BUP" and three spaces
6–7	Major version number of BUP that created the file section
10–11	Minor version number of BUP that created the file section
12–17	Reserved (zero)
20–21	First three characters of filename, in Radix–50
22–23	Last three characters of filename, in Radix–50
24–25	File type, in Radix–50
26–27	Reserved (zero)
30–31	First file section length
32–33	Intermediate file section lengths
34–35	Last file section length

Table 2–19 (Cont.): Contents of Block 0 of a BUP Saveset Section

Starting Offset	Contents
36–37	Total saveset length (without definition blocks)
40–41	Number of file sections in saveset
42–43	Section number of this file section
44–45	Reserved (zero)
46–47	Date of backup (RT–11 date word format)
50–57	Time of backup (four words: HOUR, MINUTES, SECONDS, TICKS)
60–177	CSI command line that created the saveset
200–777	Reserved (zero)

2.12 Error Log File Formats

Device handlers that support error logging call the error logger through a monitor pointer on each successful I/O transfer as well as on each error. The **copy code** in the error logger retrieves, or copies, the appropriate information from the handler, storing it in the error log input buffer in the error logger's memory area. The error log job is suspended until the copy routine puts some data into the input buffer, at which point the monitor resumes the error log job so it can process the new data.

The error log job remains suspended until the error log input buffer has filled to 200_{10} or more words of the 256_{10}-word total buffer size. The copy code portion of the EL job informs the monitor of this by setting the carry bit on return. Thus the error log job is resumed by the monitor only when the error log input buffer contains a sizeable amount of information to be processed.

For device errors, cache errors, and memory parity errors, the error logger first creates or updates the unit statistics information in the copy of the disk file header that is in memory. The EL job (disk output code) stores error records in the disk output buffer (one of two buffers, since double-buffering is used) until a 256_{10}-word block is full. That is, it stores records until the buffer cannot contain the next record. Then it writes the updated header record and the accumulated error records to a disk file called ERRLOG.DAT. Figure 2–43 describes the error logging subsystem.

For successful I/O transfers, the error logger first creates or updates the unit statistics information in the copy of the disk file header that is in memory, as it does with device and memory errors. It writes the updated header to disk only after 10_{10} good I/O transfers have been logged.

Figure 2–43: Error Logging Subsystem

2.12.1 Error Log Disk File Format

The error log disk file is called ERRLOG.DAT. Figure 2–44 shows its format.

The first part of the figure describes the contents of block 0, the header block. For each device on which error logging is done, there is a statistics record (a 7-word entry) in block 0. Since the number of records is a variable SYSGEN parameter, the pointer at the beginning of block 0 points to the end of these device-specific records in block 0.

The second part of the figure, beginning with block 1, describes the 256-word blocks containing error records. The data in these records will vary depending on the situation. The blocks can contain statistics on device errors, memory errors, or both,

depending on the error logging you have enabled and the order in which error-logging events happen.

The low byte of the first word for both Device Error and Memory Error Reports contains the record size. Records are contiguous within blocks and, therefore, the *size of this error record* field is an offset to the next record in the block. The final record in each block contains a -1 in the size field, indicating that the next record in the file begins at the first word of the next block.

The *Unit Number* field in the Device Error Record and the *Parity ID* field in the Memory Error Record indicate the kind of error (Device or Memory) which is being reported. If the field contains a value equal to or greater than zero, the report is for a device error and the field contains the device unit number reporting the error. If the value in the field is less than zero, the report is for a memory error and the values have the following meaning:

–2 for a memory error

–3 for a cache error

–4 for both memory and cache error

Figure 2–44: ERRLOG.DAT Format

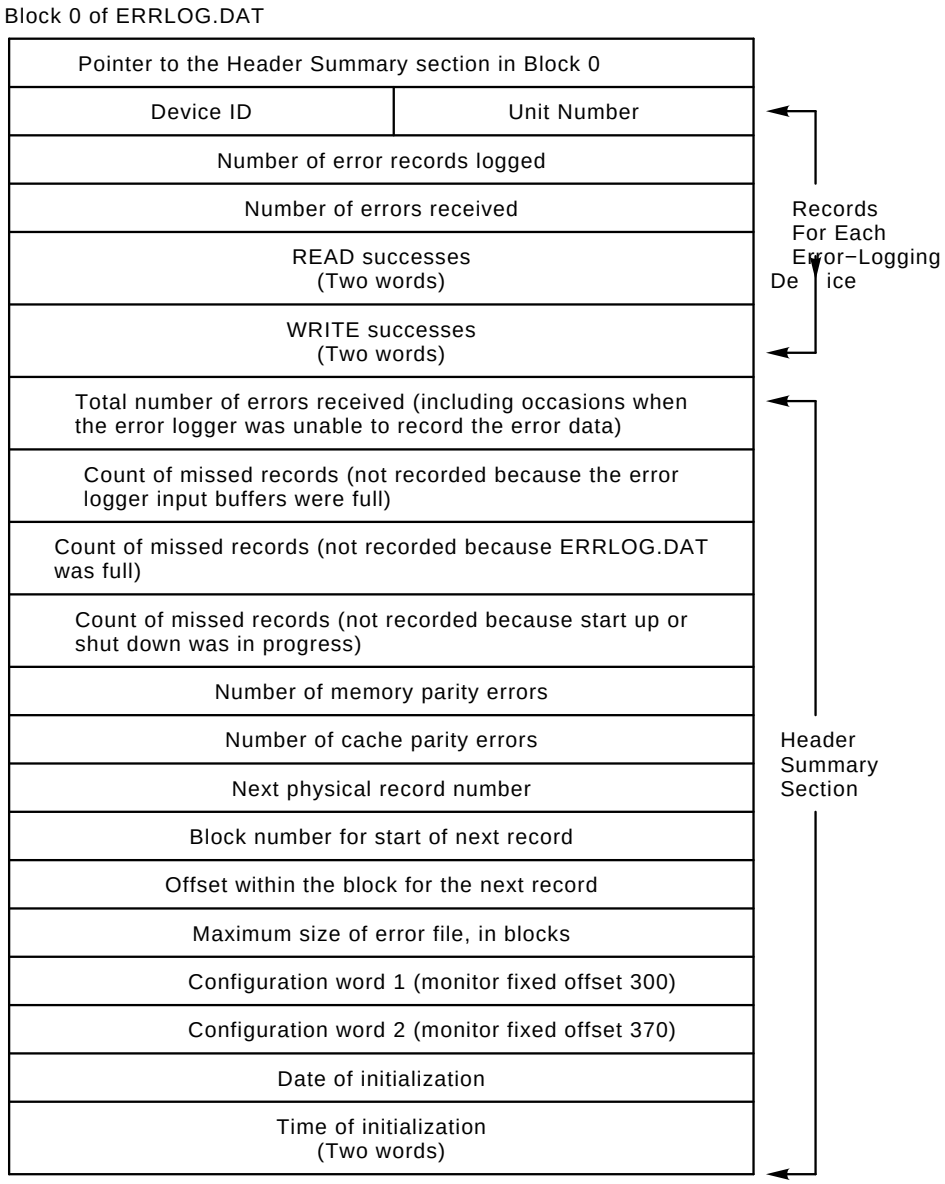


Figure 2–44 (continued on next page)

Figure 2–44 (Cont.): ERRLOG.DAT Format